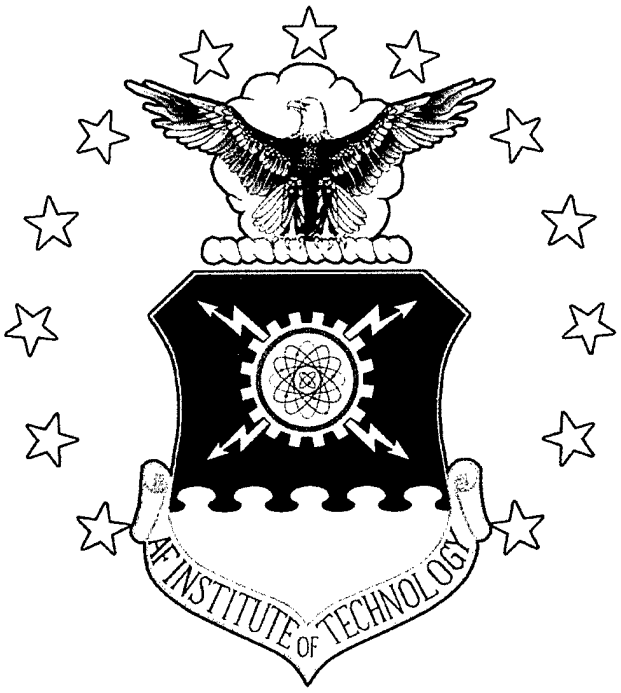

Multiagent Systems Engineering: A Methodology for Analysis and Design of Multiagent Systems

THESIS

Mark F. Wood, Captain, USAF

AFIT/GCS/ENG/00M-26

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the US Government.

AFIT/GCS/ENG/00M-26

MULTIAGENT SYSTEMS ENGINEERING: A METHODOLOGY FOR ANALYSIS AND DESIGN OF MULTIAGENT SYSTEMS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Science

Mark Wood, B. S. Computer Science

Captain, USAF

March 2000

AFIT/GCS/ENG/00M-26

MULTIAGENT SYSTEMS ENGINEERING: A METHODOLOGY FOR ANALYSIS AND DESIGN OF MULTIAGENT SYSTEMS

THESIS

Mark Wood, B. S.
Captain, USAF

Approved:

Maj. Scott A. DeLoach (Chairman)	8 Mar 00 date
Dr. Thomas C. Hartrum (Member)	8 mar 2000 date
LtCol. Timothy M. Jacobs (Member)	8 Mar 2000 date

ACKNOWLEDGMENTS

I wish to thank the members of my thesis committee, and in particular my advisor Major Scott DeLoach, for their guidance and assistance throughout the thesis process.

I would also like to thank my peers and fellow graduating officers of the Agent Research Group for their suggestions, critiques, humor, and dedication: Tim Lacey, Dave Robinson, and Marc Raphael. Additional thanks as well to the lady who makes agentTool go: Jennifer Mifflin.

I thank my friends and family members who helped me by offering advice, love, support, and proof-reading throughout the past eighteen months. Finally, special thanks to Lisa Walsh for her patience and understanding.

Mark Wood

TABLE OF CONTENTS

TABLE OF FIGURES

ABSTRACT

This thesis defines a methodology for the creation of multiagent systems, the Multiagent Systems Engineering (MaSE) methodology. The methodology is a key issue in the development of any complex system and there is currently no standard or widely used methodology in the realm of multiagent systems. MaSE to covers the entire software lifecycle, starting from an initial prose specification, and creating a set of formal design documents in a graphical style based on a formal syntax. The final product of MaSE is a diagram describing the deployment of a system of intelligent agents that communicate through structured conversations. MaSE was created with the intention of being supported an automated design tool. The tool built to support MaSE, *agentTool*, is a multiagent system development tool for designing and synthesizing complex multiagent systems.

MULTIAGENT SYSTEMS ENGINEERING: A METHODOLOGY FOR ANALYSIS AND DESIGN OF MULTIAGENT SYSTEMS

I. Introduction

The Air Force is quickly moving towards distributed C^3I applications. This is clearly delineated by visionary documents such as *Joint Vision 2010* (Shalikashvili 1999), and *Air Force 2025* (Kelley 1996). In each of these documents, *information superiority* is seen as the key factor to success in the 21st century. The *ABIS Grid* is defined as "an information environment, including communications, processing, information repositories, and valued-added services that provide the users with an ability to find information, to obtain processing services, and to exchange information." If the warfighters are going to trust such a system, we must ensure that the system and its information sources are robust, reliable, and secure.

Working in joint environments suggests the need for distributed processing systems. As these systems get larger and more complex, users are presented with new problems such as information overload and the inability to find information. An agent is an abstraction that encompasses many desirable characteristics of such a system, and helps solve the associated problems. Agents are autonomous, goal-directed, intelligent, and inherently distributable in a multiagent system (MAS).

1.1 Background

Agents have created enthusiasm recently as powerful new abstractions for designing and creating software systems. Of particular interest is the inherent ability of agents to thrive in distributed domains such as the Internet. Much recent agent research has focused on the internal structures of individual agents, leading to a concept of agents as individuals rather than as merely parts of a distributed system. However, collections of agents must work cooperatively in order to solve complex problems, which is the principle advantage of agents. There are several motivations for using collections of agents as a collaborative problem-solving tool (Nwana 1996). They can solve problems too large for a single agent. They can also solve inherently distributed problems. Finally, they can handle distributed information sources and expertise. The most powerful tools for handling the complexity of these large systems and problems are modularity and abstraction (Sycara 1998). MASs provide the modularity of individual agents that are specialized to perform particular tasks. MASs are concerned with the coordinated behavior of a collection of individual agents to achieve system-level goals.

Constructing a MAS is difficult. They have all the problems of traditional distributed and concurrent systems, plus the additional difficulties that arise from flexibility requirements and sophisticated interactions. Sycara (Sycara 1998) states that there are two technical hurdles to the extensive use of multiagent systems. First, there is no proven methodology that enables designers to clearly structure applications as MASs. Second, there are no general case industrial-strength toolkits that are flexible enough to specify the numerous characteristics of agents.

This thesis addressed the first technical hurdle by proposing a methodology for the design of MAS. The focus was on the construction of a MAS through an entire software development lifecycle from problem description to implementation. For the most part, studies of this sort have

focused more on high-level descriptions and concepts than on an actual design methodology. Other design systems do exist as general-case solutions, but these are neither tuned for nor particularly useful in creating a system that is intended to take full advantage of agent capabilities. Object-oriented design has achieved some maturity and therefore provides a stable foundation to build upon. However, object-oriented methodologies are not directly applicable to agent systems – typical agents are significantly more complex in both design and behavior than objects.

1.2 Problem

This research focused on developing a design methodology to create a multiagent system starting from an initial prose specification, and producing a set of formal design documents in a graphically-based style. This methodology is the basis of AFIT's agentTool development system, which also serves as a demonstration and validation platform, and a proof of concept. The formal design is also further manipulated within agentTool through work in concurrent thesis research to specify individual agents (Robinson 2000), verify agent communications (Lacey 2000), and store design components in a knowledge base (Raphael 2000).

There are two principal strengths of the methodology developed through this research: it is focused toward the specific capabilities of multiagent systems and it is based on a formal language and rules. The methodology supports distributed multiagent systems by helping the designer create structured communications between agents. This is possible because the methodology is focused on the domain of agents, meaning that a system designer using this system has already made the decision that the system will be agent-based. This domain restriction also supports the creation of a graphical language, which is the basis of the graphical user interface (GUI) used by the agentTool development system. The graphical diagrams give an

informative view to the designer of the system dimension currently being developed, and also hide the formal constructs.

1.3 Goal

The goal of this thesis was to produce the methodology described above, a full-lifecycle methodology for creating multiagent systems. This goal was supported by research examining the different ways that agents have been used in the creation of software systems. The research focus was toward design and analysis abstractions, looking at the system lifecycles, and creating design processes. The result of this research was the construction of a methodology for creating software systems based on multiple software agents.

1.4 Assumptions

Prior to conducting research, it was necessary to make a few assumptions in order to narrow the scope of the methodology. The first assumption was that every system designed using the methodology would be a closed system. Any contact external to the system is performed by an agent created as part of the system that participates in the system communication protocols. This eliminates consideration of introduction of conflicting goals. Since external entities can be interfaced with via an agent, this assumption does not greatly affect the domain of possible systems that can be designed.

The second assumption was that the scale of systems designed through this methodology would not be very large; the target is ten or less software agent classes. There is not a hard limit set at ten, but simply no verification or validation is done of larger systems, and no thought is given to potential problems of such a system. The practical effects of this assumption were that the methodology did not have to consider any potential effects of a system with a large number of

agents and the resulting complex inter-agent communications. Furthermore, in consideration of the eventual implementation of agentTool, the diagrams used by this methodology are much easier to reproduce graphically with a small number of agents.

The third assumption was that there is no requirement for agent mobility. A mobile agent is one that can move between computers hosting the MAS. One method of accomplishing this mobility is for the agent to start a new version of itself at another site, send it the state information from the old version, then terminate and delete the old version. This creates a new copy of the agent that continues where the old one left off. This produces several problems for the rest of the system such as updating all other agents with the new location of the mobile agent. Also, what happens if an agent transfer goes awry and there are multiple or no copies of an agent? The inclusion of mobile agents may have added much complexity to the methodology, and not added much functionality. Most of the benefits of mobile agents can be designed into a system by simply using multiple agents.

Fourth, the methodology does not consider dynamic systems where agents can be created and destroyed during execution. This would lead to many of the same problems as mobile agents. An agent can be added to a system through a process such as registration that is a user-initiated event, but not added and deleted continuously during ordinary operation.

The final assumption was that inter-agent conversations are assumed to be one-to-one, as opposed to multicast. This assumption was made after investigation of conversation representation, and acceptance of a graphical dual-state table representation. Substituting a series of point-to-point messages will fulfill a requirement for a multicast message.

1.5 Areas of Collaboration

This research was conducted in conjunction with three other thesis efforts. All four projects were aimed at constructing a tool, named agentTool, which takes a multiagent system through the entire lifecycle from analysis to implementation utilizing formal constructs and transformations. The goal of agentTool is to take a formal agent system specification and produce working code. The other three projects were concerned with validating agent communications (Lacey 2000), agent-level design (Robinson 2000), and the design and operation of a knowledge base for agentTool (Raphael 2000).

Furthermore, the starting point of the methodology and focus for the remainder of the research is the Multiagent Systems Engineering (MaSE) prototype developed by Maj. DeLoach (DeLoach 1999), and described in Section 2.3.4. The methodology developed in this thesis is named the "MaSE methodology", and uses most of DeLoach's MaSE diagrams as well as the same engineering focus. The first MaSE paper described diagrams and concepts of MAS design. The MaSE methodology extends those ideas, adds more concepts from the analysis and design phases, and orders them into a methodology that covers the whole software lifecycle. For the remainder of this thesis, the acronym "MaSE" refers to the original paper by DeLoach only when used in Chapter 2, and otherwise refers to the methodology developed through the thesis. If used together, the DeLoach work will be called the "MaSE prototype", and the thesis result, the "MaSE methodology".

1.6 Overview of Thesis

The remainder of this document is organized as follows. Chapter 2 provides the background material about agents and multiagent system design. Chapter 3 describes the process of constructing a comprehensive methodology for multiagent system design that covers the entire

lifecycle. Chapter 4 presents the methodology and takes an example system through the design process. Chapter 5 describes implementation of a graphically based multiagent system design application based on the methodology. Chapter 6 discusses conclusions reached during this study and possible future research.

II. Background

2.1 Introduction

The field of artificial intelligence (AI) is an atypical discipline, as any discussion of AI must start by asking *what is AI*? Consider a discussion of biology or baking beginning the same way and the point is quite clear. The recent popularity of *agents* as an abstraction tool for AI faces a similar definitional dilemma. There are many differing views on what an agent is or is not. Perhaps the only agreed upon characteristic of an agent is the one already mentioned; that an agent is an *abstraction*, useful for solving problems in particular domains. In general, most characterizations of agents involve a few specific terms that directly map into a discussion of multiagent systems. Agents are inherently distributed. They are autonomous and goal-oriented as well as social, and they share information with other agents interactively.

Multiagent systems (MASs) do not suffer from the same definition problem as both AI and agents. At the macro level at least, MASs are accepted to be collections of agents that interact to solve a problem. The general idea is that the problems they solve are too complex for a single problem-solver. Of course with multiple agents as components, the micro level of a MAS may be pictured quite differently in the minds of various researchers. Many principles are the same though.

Before launching into specifics, there are many general characteristics of MASs that appear frequently in contemporary research and in the papers discussed here. There are many different concepts of what makes up the internals of an agent, and many share the idea of agents fulfilling sets of *roles*. Roles are analogous to roles played by actors in a play or by members of a typical company structure. The company has roles such as "president", "vice-president", and

"mail clerk" arranged in a hierarchy. The instantiation of these roles is not necessarily static. The mail clerk may one day become the president. A role has certain responsibilities associated with it as well, such as the "mail clerk" role that must deliver the mail. These responsibilities are associated with particular goals, so a role can be thought of as an abstraction that encompasses a particular goal or set of goals. An agent that plays a particular role is fulfilling those goals.

The idea of an agent "goal" is an anthropomorphism to reflect some internal purpose to an agent's actions. A goal may also be system-level, which is the more commonly understood context.

MASs must utilize some method of communication that links agents. These lines of communication are comprised of conversations. A conversation is a sequence of messages between two (or more) classes of agents that fulfills a particular goal. For example, an agent may need to register with a registration agent when entering a system. The "registration" conversation would involve a series of messages passing domain-specific information to complete the registration. Typically, conversations support a particular goal or objective. It is the patterns of conversations that determine the shape of a multiagent system and tap the power of distributed agents.

There are several motivations for using agents as a collaborative problem-solving tool (Nwana 1996). They can solve problems too large for a single agent. They can also solve inherently distributed problems. Finally, they can handle distributed information sources and expertise. In general, a distributed system has enhanced speed, reliability, and flexibility over a centralized system. Plus, the use of multiple agents adds reusability and modularity.

MAS design is a fledgling discipline. Certainly no standard or widely used methodology exists. Current methodologies are either entirely research-oriented, or highly specialized. Much

of the current agent-oriented design research has focused on agents as individuals, rather than as parts of a distributed system. However, there are some recently emerging ideas about how to model certain MAS facets such as conversations and "patterns" of agents, and a few attempts to merge these various facets into a cohesive design methodology. This chapter discusses and analyzes some of these emerging ideas that form the current state of the art in multiagent systems creation.

This chapter is divided into three phases of the software lifecycle: analysis, design, and implementation. The first part will cover some modeling abstractions that have been presented to assist MAS design. These will be presented from first the analysis (Section 2.2) and then the design phases (Section 2.3) of the software lifecycle. Three design methodologies that "flesh out" the process fairly well are included. The next part will cover the sparse field of implementation (Section 2.4) where a formal design of a MAS is transformed into code. Finally a summary of the whole process and an attempt to draw some conclusions is presented.

2.2 Abstract Methodologies – Analysis phase

The objective of the analysis phase of the MAS software lifecycle is to transform the system requirements into a representation of the system that can be forwarded to the design phase. Additionally, it is important that the analyst understand the domain of the system being built. System analyses in other contexts tend to concentrate on how a particular system will function, perhaps using an interaction model. Kendall and Zhao suggest four analysis patterns (discussed in Section 2.2.1 below) that focus more on *what* the system is trying to achieve (Kendall & Zhao 1998). They suggest that the nature of goals create a less variable product than a classical analysis based on tasks and activities.

The basic purpose of goal-based analysis patterns is to map requirements to implementation. More specifically, a method for determining what types of agents are required for the design phase is needed. Since agents are goal-driven entities, the product of system analysis will consist of a collection of goals, which can then be mapped to agents or (even better for the systems engineer) patterns of agents. Finally, constructing agent conversations to support goals is valuable since, as mentioned above, conversations form the backbone of any MAS.

2.2.1 Capturing and Structuring Goals: Analysis Patterns

The four analysis patterns described by Kendall and Zhao (Kendall & Zhao 1998) for focus on what a MAS should do are Capturing Goals, Goal Cases, Structuring Goal Cases, and Goal Cases as Objects. The patterns are designed in a waterfall model as shown in Figure 1. They are intended to be applied in series. The output of each phase is the input for the next. In practice, the process is iterative and incremental as depicted by the arrows in the diagram. The overall objective of this approach is to take a prose specification of a system and construct a Goal Hierarchy Diagram, which can be used as a foundation for future modeling and "passed on" to the design phase.

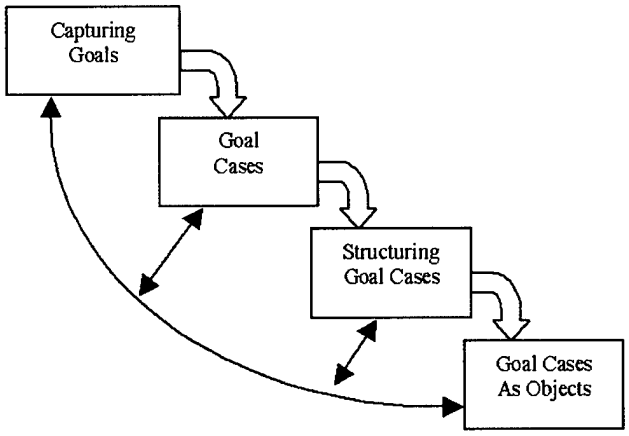

Figure 1: Analysis Pattern Map

The first pattern, Capturing Goals, takes the initial context of the system as input. An initial context is an informal description of requirements, perhaps structured as bullet statements. An example might be "The system is responsible for making a class schedule for AFIT which considers conflicts between instructors, student, and room availability." The pattern attempts to capture what is important in an extensive set of such statements. The resulting context is an explicit list of goals and objectives. These goals and objectives are much less likely to change than the more detailed steps that follow in the lifecycle and as such, provide a foundation for analysis. Example goals from the first example would be: "To schedule classes" and "to deconflict schedules"

Goal cases are essentially goals plus the source of the goal (a use case). The context resulting from this pattern is a combination of the goal and the text from which it was derived. The point of this second pattern is to maintain traceability in case the requirements are revisited.

The third pattern creates a Structured Goal Case that can be pictured in the form of a tree diagram. This adds a hierarchy to the goals. Figure 2 shows a Goal Case that is in keeping with the example above.

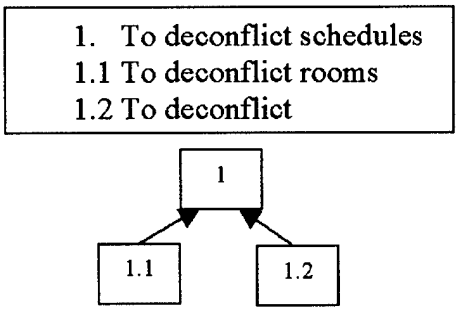

Figure 2: Structured Goal Case

The final pattern removes redundancy from the tree in case a goal appears in more than one place. The tree would no longer actually be a tree, but would still be a structured hierarchy. The Goal Cases in this structure are modeled as objects. They inherit or multi-inherit from parents in the neo-tree. Additional goals and actions can be added to subordinate goals as needed, just as is done with objects. The advantage of applying these four patterns is that the resulting solution captures the goals of a system. This leads to a goal-driven analysis rather than a task-based or activity-based one. The resulting goal hierarchy diagram is more stable than a structure based on activities or tasks.

Potential problems with this method are similar to those with other methods. The scope of the system being modeled must be clearly defined. Furthermore, this procedure can result in a very large number of goals, which would then have to be separated by sub-function in order to maintain clarity (Kendall & Zhao 1998).

2.2.2 A Methodology for Agent-Oriented Analysis and Design

A more recent methodology combines some aspects of both the analysis and design phases into a single framework. Wooldrige, Jennings, and Kinny propose taking a statement of requirements and producing a design in sufficient detail that it can be implemented directly using "traditional techniques". They make several assumptions up front that are worth repeating, as they seem appropriate for any MAS design procedure (Wooldrige, Jennings & Kinny 1999).

- Agents require a significant amount of computational resources, roughly equivalent to a process in UNIX
- The goal of the MAS is to maximize some global quality measure that may be sub-optimal from the point of view of the individual system components

- Agents are heterogeneous, meaning that they can be implemented using any number of different languages and programming techniques
- The entire MAS contains a relatively small number of agents (less than 100)

This methodology (MAAD, for short) works in two familiar stages: Analysis, followed by Design. The whole process is based around the idea of building agent based systems as a process of organizational design, with each system as a collection of agent roles. The roles are analogous to a typical company structure. The company has roles such as "president", "vice-president", and "mail clerk" arranged in a hierarchy. The instantiation of these roles is not necessarily static. The mail clerk may one day become the president.

A role is defined in MAAD similarly to how it was done earlier in this thesis, by *responsibilities, and permissions.* Additionally, roles here have *protocols*, which define the different ways an agent class can interact with other agent classes.

The objective of the analysis phase is to develop an understanding of the system without concern for any implementation detail. This understanding is captured in the *organization* of the system, which is considered the highest level abstract analysis concept (shown in Figure 3).

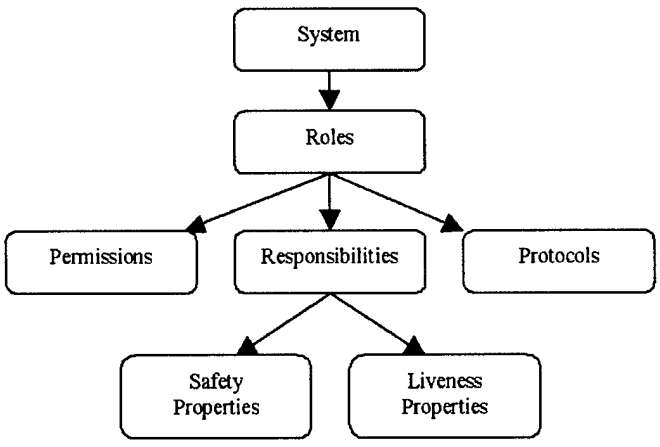

Figure 3: Abstract Analysis Hierarchy

The organization is comprised of two models, the *Roles Model* and the *Interaction Model.* They capture essentially the same information as the Agent and Interaction models in the BDI method, except there is a much more detailed description of the responsibilities associated with a particular role.

A responsibility can be divided into one of two categories: *liveness* and *safety* responsibilities. Liveness responsibilities are those that state "something good happens". An example liveness responsibility is given for a CoffeeFiller role: "fill up the coffee when it is empty". By contrast, a safety responsibility is an invariant constraint such as "coffeeStock ≥ 0", which makes sense because a CoffeeFiller can't add more coffee to the pot than that available in stock.

2.2.3 Transitioning from Analysis to Design

The two MAS analysis models described above are essentially based respectively on goals and roles. The first model by Kendall and Zhao ends system analysis with a structured hierarchy of goals, but goes no further into the software lifecycle. The MAAD analysis model begins with constructing roles that are mapped to agent types in the design phase, but offers no guidance as to their construction.

In order to proceed to agent design, goals must be transformed into agents. In general, each goal maps to an agent role. Duplicate goals may be combined into single roles, though use of the fourth pattern described by Kendall and Zhao to remove redundancy from the hierarchy should alleviate that. Goals that are completely defined by sub-goals are not mapped to roles. The sub-goals can simply be mapped to roles in their place.

Agents may then be defined through roles. Multiple roles may be combined into a single agent. This process of role-mapping can be very complex, and will be covered more in the next section of this paper. So far, we have mapped system requirements to goals, mapped goals to roles, and mapped roles to agents. There has not been much research into this analysis process, and certainly no actual engineered solutions. The conceptual framework is slowly being laid, however.

One final (and perhaps separate) phase in MAS analysis is determining conversations between agents. It is through conversations that the power of the distributed system is realized. Conversations are as important to defining a MAS as the agents themselves. To be effective, multiple agents must coordinate and communicate their actions. At the analysis level, conversations should be defined as the minimum messages that must be passed. Additional messages may be added for robustness and completeness. Any communication between agents requires at least one conversation type. Conversations are defined as between either agent classes or roles, depending on the method used. The following section on the design phase of MAS development elaborates beyond the simple need for having conversations.

2.3 Abstract Methodologies – Design phase

The design phase of the software lifecycle has been the focus of much of the non-applications research on MASs and agents to this point. This section will cover several different methods of classifying and patterning agents, including three design methodologies that are, in some respects, successive refinements of each other. Each is based on previously existing object oriented design processes. Object oriented design has achieved some maturity and therefore provides a stable foundation to build upon. However, object oriented methodologies are not

directly applicable to agent systems – typical agents are significantly more complex in both design and behavior than objects.

The idea of using roles to define an agent has already been introduced. In a bit more detail, a role is an abstraction that encompasses a particular goal or set of goals. An agent *plays* a role in a system much like an actor in a play. Someone gets to play the "Hamlet" agent and receive the applause, someone else is the "Horatio" agent and gets to live, but every role has to be filled so someone has to play the "Rosencrantz" agent and get neither applause nor life. "Alas, poor yorickAgent17, I new()'d him."

2.3.1 Roles and Role Models

Kendall suggests a new way of organizing agents into reusable patterns. She proposes that agent roles be collected into patterns called *role models* (Kendall 1998). A role model is a new abstraction for modeling and designing agent systems. The basic idea is that patterns of agent roles are constructed, labeled, and archived. When a new system is designed, the patterns of the systems are analyzed. If patterns are recognized that are supported by existing role models, then the role model can be added to the system, resulting in a slew of agent roles that satisfy the subset of the system goals involved in the pattern that was recognized.

Kendall also expands on the definition of roles, describing how a role in a role model can be played not only by an agent, but also by objects, processes, organizations, and even people. The underlying idea is that as long as every role in a role model is covered, the role model will function as intended.

Figure 4 depicts a sample role model named the "mediator" pattern that represents a typical organization. The mediator is a central role, like a plant manager, that is responsible for interacting with clients.

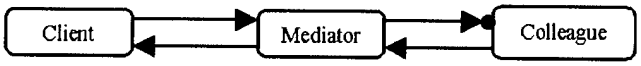

Figure 4: The Mediator Pattern Role Model

The set of colleagues (note the "•" in Figure 4 represents "one to many") could represent other factory sections like manufacturing, QA, and a repair shop. All interactions with the client are done through the mediator who then delegates responsibilities to the appropriate colleagues. In terms of roles, the mediator has the *responsibilities* of handling all client communications, delegating tasks to colleagues, and passing results back to the client.

Upon instantiation, each role must be played by some entity. Continuing with the pattern in Figure 4 as an example, the mediator could be a webserver process that handles requests from websurfing human (or agent) clients. Depending on its particulars, a request could be forwarded to one of several colleagues: a search-bot, a database query, or a live connection to a system administrator.

There are obviously many situations where the mediator pattern or another role model would effectively map a to particular system or piece of a system. Once a pattern is recognized, a role model that matches that pattern is retrieved from a library. It can be combined with any other patterns in a variety of ways. A role model may be an aggregate of others. For example, one object can play both a colleague role from the mediator pattern and a client role if it must go collect information from an external source, which is also follows the mediator pattern. A particular pattern may also be derived from a base model, which is similar to object-oriented

inheritance. In that case, the derived role must play all the roles of the base role. Furthermore, roles can form role sequences where an entity iterates through a sequence of roles. A *producer* of one item may become a *consumer* of another under an additional common role model pattern called the "supply chain" pattern.

Role *Synergy* is what happens when a composite role model, utilizing the combination methods mentioned above, is more than the sum of its parts. An example of this is the "bureaucracy" pattern shown in Figure 5. There are many details involved with the construction of such a pattern that are beyond the scope of this discussion, but looking closely, it is clear that the mediator pattern roles of mediator and client being played in this pattern by the "Manager" and "Subordinate." This pattern actually combines four separate patterns together into a powerful composite (as anyone who has played a role in a bureaucracy can tell you).

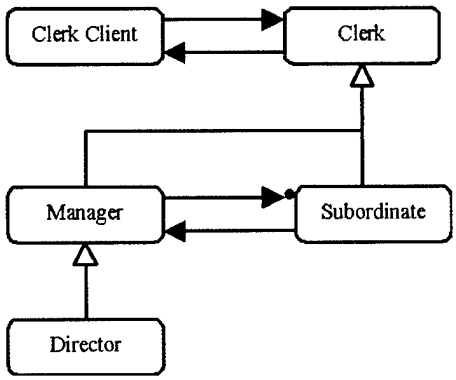

Figure 5: Bureaucracy Pattern

Role Modeling is appropriate for MAS design for several reasons. First, it emphasizes interactive behavior by structuring patterns around communications (all of the arrows on the above diagrams are communication paths). Second, all of the roles in a role model work toward a common set of goals. Finally, roles and role models provide a new abstraction that can unify

diverse system aspects from intelligent agents to people. Role models do not by themselves ever fully describe a system design, but they provide an excellent tool for any of the following methodologies, or a new composite one.

2.3.2 A Methodology and Modelling Technique for Systems of BDI Agents

In a relatively early foray into a system-level design methodology for MASs, the authors proposed adapting and extending current object-oriented design approaches into the agent world, specifically applying them to Belief-Desire-Intention (BDI) agents (Kinny, Georgeff & Rao 1996).

A general-case BDI agent is defined primarily by its "mental state", which is formed from three facets appropriately called *beliefs*, *desires*, and *intentions*. Each facet is implemented by a set of appropriate statements. Beliefs are a set of statements that reflect the agent's knowledge about the environment. The beliefs could be imbedded into the agent when it is created, or implied though trial and error. As far as the agent "knows", the statements in its belief set are true. The set of BDI agent desires are what the agent *wants* to become or remain true. It is a representation of the goals of the agent. The intentions set is a collection of actions that the agent will take in support of its desires. In many BDI agents, intentions are created through some form of agent planning. In this context, planning would consist of determining which actions or sequences of actions would take the world from the state described in the belief set to the state in the desire set. The resulting actions are then placed into the intentions set.

This methodology by Kinny, Georgeff, and Rao is aimed at creating a set of models that define an agent system based on BDI agents. There are two sets of models, the *external viewpoints* and *internal.* The external models present a system-level view made up of agents characterized by purpose, and is primarily concerned with the interactions between those agents,

including both inheritance relationships and communications. The internal models are related to a specific agent class, and are concerned with the internal workings of that class, specifically their beliefs, goals, and plans.

The external viewpoint decomposes a system into agents based upon the key roles of the application. Agents are instantiations of classes within an agent class hierarchy. The details of the agent classes are covered by two models: the *Agent Model*, which describes a number of actual agents and the hierarchical relationships between agent classes, and the *Interaction Model*, which describes communications and control relationships between agent classes.

The methodology for specifying these models consists of four steps. The steps are performed in sequence, and repeated for refinement. These steps and the external viewpoint are independent of the architecture of a particular agent.

1. Identify the roles required
2. Identify responsibilities and services for each role.
3. Identify interactions (communications) for each service.
4. Refine the hierarchy by introducing new agent classes, composing classes, and creating new agent instances.

The abstraction hierarchy is straightforward. Roles are sets of responsibilities, and responsibilities are sets of services. Revisiting a previous example, if a role of an agent was "to de-conflict schedules", an associated responsibility could be to "maintain a master schedule" and a service would be to "provide a master schedule upon request."

The internal viewpoint of this methodology is strongly tied to the BDI paradigm. It contains three models which are direct extensions of either object oriented models (beliefs and

goals) or dynamic models (plans). A *Belief Model* holds an agent's *belief set*, which is information about both the environment and the agent's current state and whether or not they may change over time. The *Goal Model* consists of a *goal set*, which is a set of possible goals that an agent may adopt and events to which it can respond. These goals are the "desires" piece of BDI. Finally, the *Plan Model* holds the creatively named *plan set,* which is a set of possible plans an agent may adopt to achieve its goals. The plan set contains possible intentions of a BDI agent.

There are two steps in the methodology for modeling the internals of an agent. Again, they are designed to be continuously iterated and the models refined as the design process is performed.

1. Analyze the means of achieving goals. That is, in what order and under what conditions can a goal be achieved.
2. Build the beliefs of the system.

In summary, The BDI methodology presents a structured framework for designing MASs at both the system and agent level. Goals are used as the focus of design because, as stated previously, they are more stable than behaviors or plans in any application domain. Plans are highly context-sensitive. Many different plans may be created for a particular goal in different situations. The focus of the BDI methodology is on the end-point that needs to be reached rather that on the types of behaviors that lead to that end-point. This idea of designing from goals is the primary difference between the BDI methodology and object oriented methods.

2.3.3 A Methodology for Agent-Oriented Analysis and Design

Returning to the MAAD methodology introduced in Section 2.2.2 (Wooldridge, Jennings & Kinny 1998), the design phase generates three more models. The *agent model* identifies agent types for the system and agent instances that are created from those types. The *services model*

describes the services associated with each agent type. Finally, the *acquaintance model* describes acquaintances, which are possible communication pathways, in order to identify bottlenecks.

The methodology connects its analysis and design portions using the roles defined in the roles model produced in the analysis phase. It offers a small amount of guidance for making the transition. A designer can choose to package a number of closely related roles in the same agent type for the purposes of convenience. Efficiency will also be a concern, and is another reason to combine roles. As with the role models described above, a fair amount of expertise is required to make these decisions.

MAAD is similar to the BDI agent methodology described earlier. Both methods work at two levels: system and agent. They both use agent roles as key concepts and associate them with responsibilities. Furthermore, both communication processes are designed around conversations between roles. Each process moves from the abstract concepts toward the concrete, with each move shrinking the space of possible systems that could be implemented. Finally, each process is designed to be iterative, with successive iterations refining the system at various levels. The MAAD authors take note of these similarities and even provide a comparison between the two, highlighting the more detailed definition of role responsibilities in MAAD.

2.3.4 Multiagent Systems Engineering Prototype

The most recent research into MAS design covered in this research is Multiagent Systems Engineering (MaSE) (DeLoach 1999). MaSE builds upon other research, including the previously discussed MAAD, but re-focuses upon the actual engineering process. In other words, it is nice to talk about creating MASs, but how does one actually *do* it?

MaSE has two primary goals:

- To develop a complete methodology and language for designing framework and architecture independent MASs
- To structure the methodology specifically to support formal automated software development

Furthermore, MaSE attempts to make its component languages (described below) graphically-based to hide formalisms as much as possible and to support automatic generation of agent systems either through code synthesis, or component library reuse.

In a similar fashion to the simultaneous macro and micro approaches of the previous two methodologies, there are two component languages for describing MASs in MaSE. The Agent Modeling Language (AgML), and the Agent Definition Language (AgDL). AgML is the system-level language, distinguished by being primarily graphical in nature while maintaining a formal syntax. AgDL is used to completely describe the internal workings of an agent. AgML is reviewed in further detail later, under the first section of the methodology. AgDL and the corresponding details of agent-level design are currently under development. For more information see parallel thesis work by Robinson (Robinson 2000).

MaSE is a four-step waterfall process, as depicted in Figure 6. It is not explicitly shown to be intended as an iterative refinement process, though it could certainly operate that way,

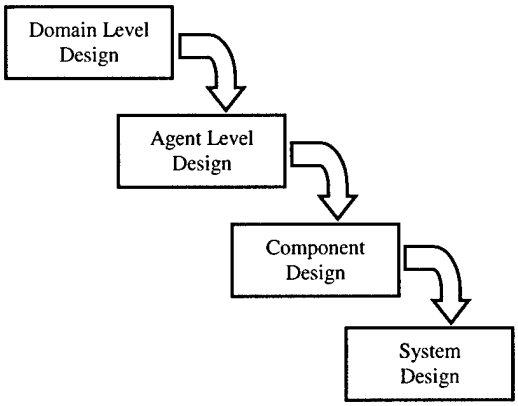

Figure 6: MaSE Prototype

similar to previous models.

The first step, *domain level design*, uses AgML to capture the basic types of agents in the system, and interactions between those types. It does so in three steps:

1. Identify agent types
2. Identify possible interactions between types
3. Define coordination protocols for each type or interaction

Through the use of AgML, the process described above is conducted using a graphical format, which still enforces the underlying formalisms. AgML uses four diagrams to define the macro level features of MASs. The *Agent Diagram* is dissimilar to like-named diagrams from other methods. In this case, the "connections" between agent types are not hierarchical relationships, but actual conversation paths. The *Communication Hierarchy Diagram* defines exactly what its name suggests. A *Communication Class Diagram* (Figure 7) is actually a state-

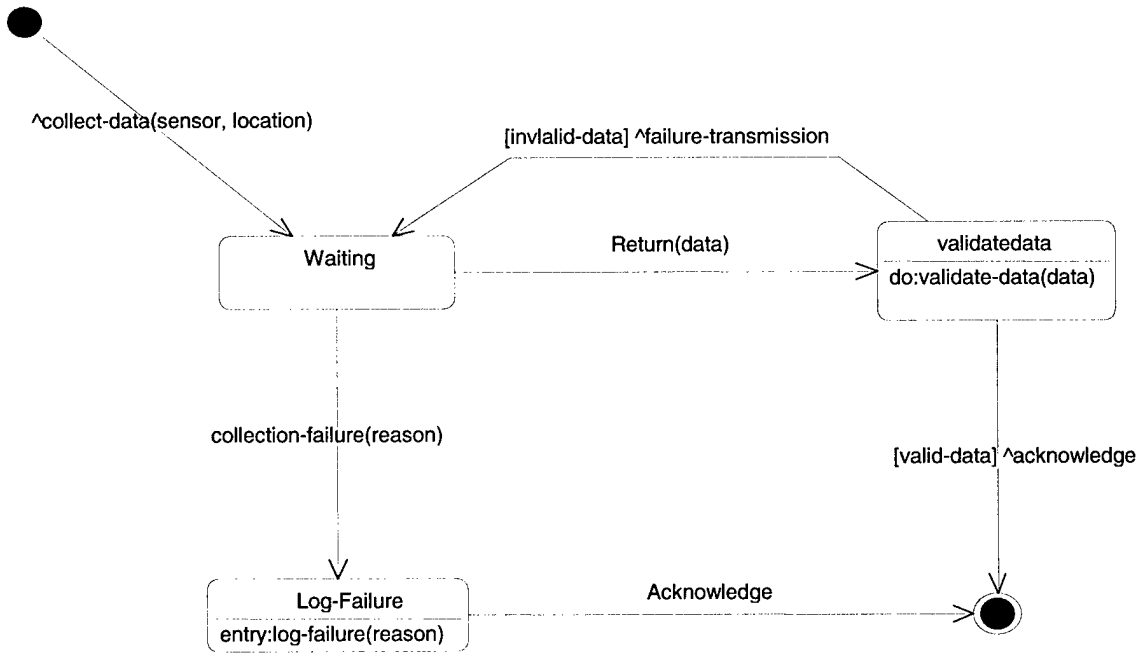

Figure 7: Communication Class Diagram

machine representation of a conversation. It is within this diagram that the focus on engineering proposed by this methodology is evident. While the details are not evident in this example, the state-machine characteristics of the diagram should be. The diagram consists of a start state (the single circle) and an end state (double circle). There are several other states (boxes) connected by transitions (arrows). Each state can optionally have an action associated with it, and each transition can include guard conditions, sent and received messages, and further actions. The conversation logically proceeds from the start state to the end state by "taking" the transitions to the intervening states, as allowed by the conditions on the transitions.

The second step in MaSE is *Agent Level Design*, where each agent type is described using AgDL. Again, there are three steps to this phase of design:

1. Map actions in conversations to internal agent components
2. Define data structures identified in conversations as input and output
3. Define additional internal data structures

Next is *Component Level Design.* This step details the components introduced in preceding steps. Components can either be created from scratch using AgDL, or hopefully taken from a re-use library given a common construction method such as JavaBeans. Some possible reusable components are planners, search algorithms, and learning algorithms.

The final step of MaSE, *system design*, is the assembly point for all previously designed agents and other components. It includes picking the type and number of components and assigning physical parameters such as addresses.

Delaying definition of the complete system until the final stage of development may seem backward compared to other methodologies. The only process really delayed though is the actual construction of "physical" entities (it seems strange describing software objects that way),

from abstract types. This delayed system design appears to better support reuse, since any predefined components may impose constraints such as other required resources. These kinds of constraints may then be folded into the design without having to rearrange an existing system layout.

One final important characteristic of MaSE is the significance of conversation design in the scope of the entire methodology. Conversations are the first piece of the system to be "engineered" when they are described in state diagrams in the first stage of the design process. This reinforces what was mentioned above, that it is the conversations that truly provide the power to a distributed system.

2.3.5 Design Summary

There is significant progress along the lines of agent role models and similar work that provide helpful abstractions to MAS design. There have also been several methodologies proposed that have classified discrete steps in a design process leading to some sort of system design. These methodologies all include extended iterative OO design procedures, and their progression follows a few patterns including a governing focus on *engineering* systems rather than just *describing* them.

2.4 Implementation

The final lifecycle step covered in this research is implementation. This is the process of transforming some sort of agent system specification to actual working code. While progress is being made toward this area, none is actually being made *in* it. While there are examples from general software engineering at AFIT (KIDS, SPECWARE, AFITTOOL), there are no currently published reports or products that actually implement code generation as a final step in a formal

MAS creation process beginning with analysis and design. All of the current methods get up to a point - some set of formal and/or informal diagrams, tables, and descriptions – and then say "use traditional methods from here." The use of the methodologies described will certainly assist in the system creation process and during coding, but there will still be a break where the agent-oriented approach ends and a "traditional" approach such as OO takes over.

2.5 Fielded products

Reticular Systems' AgentBuilder is advertised as a toolkit for building intelligent agents. It provides a GUI where a user can add agents to a system, and define communications between them. The communications are not described to the level of detail as conversations are in MaSE. Instead, there is a protocol-editing tool that allows specification of a particular response to a particular message. There is no concept of a collection of messages forming a conversation.

AgentBuilder was excluded from consideration in the previous section because it does not apply a formal methodology to the process. Indeed, with a significant learning curve, you can build an agent from the ground-up. Agents can be placed into a particular system, called projects. Agents are added to projects as "skeletons" and basic interactions with other agents can be defined at that point.

The problem is that there is no process that helps a designer to decide why to add or not add a particular agent or conversation. In short, AgentBuilder does provide a toolkit (as advertised) to assist in creating agents, but does not combine it with a methodology of the sort described above to enforce a particular, validated design process. In other words, there is no associated formalism that will force a developer to NOT do something wrong.

The product seems quite solid and the imbedded run-time environment is a nice addition. However, this is not the industry-quality product needed to push MAS development into the mainstream and unlock the potential of distributed agents.

2.6 Summary

The idea of agent roles is a common thread throughout agent analysis and design discussions. Most recent MAS design methodologies begin with identifying roles as the focus. An often-repeated definition of a role is an abstraction representing a collection of goals. Goals also seem accepted as an appropriate abstraction for agents. Many systems focus on goals as the basis for system analysis, in a break from OO techniques that focus on methods or other "abilities". The goal provides a more stable framework for agent systems as it less unlikely to change over time.

All MAS design methodologies create a collection of models at different levels of abstraction. Concepts typically modeled are hierarchies of agents or (more abstractly) agent roles and conversations/interactions between agent types. The levels of abstraction are typically limited to two: a system / macro level, and agent / micro level models.

The research of the primary MAS software lifecycle phases from analysis and design through implementation are focused mostly in the design phase. Abstractions are the key to complex design, and there are some good ideas out there.

Constructing MASs is difficult. They have all the problems of traditional distributed, concurrent systems, plus additional difficulties that arise from flexibility requirements and sophisticated interactions. Sycara (Sycara 1998) states, and this chapter has shown to some degree, that there are two big technical hurdles to the extensive use of multiagent system

development. First, there is a lack of a proven methodology that enables designers to clearly structure applications as MASs. Second, there are no general case industrial-strength toolkits that are flexible enough to specify the numerous characteristics of agents. Another remaining hurdle is a social one. Before we will design systems run by agents, we must first trust those agents to run the system.

Recent methodologies have begun to focus more on the engineering of agents than on the science of them. Formalisms existing a few years ago for describing and reasoning about agents did not provide adequate support for the actual process of agent design (Kinny, Georgeff & Rao 1996).

There are several specific areas in MAS technology still needing more work. One area is further translation of the "fuzzy" to the "concrete". In other words, abstractions are nice, but a working application is what it's all about. Finally, agent systems that do exist are mostly created for specific cases. Future system-development aids must be able to design for the general case multiagent system.

III. Problem Approach

This chapter describes the process used to create a new multiagent system design methodology called Multiagent Systems Engineering (MaSE). Section 3.1 defines general characteristics desired of a MAS methodology. Section 3.2 describes how the scope of the methodology changed during the research cycle. Section 3.3 explains how the expanded scope caused holes in the methodology and how those holes were filled. Section 3.4 describes the creation of a Java application which partially implements the methodology. Finally, Section 3.5 briefly overviews the procedures used to test and validate the methodology.

3.1 Requirements

The first step in constructing a methodology must be to define exactly what a methodology is. When constructing a methodology for the creation of multiagent systems, additional effort must go toward consideration of the characteristics of such systems and what that means for the methodology. This section describes the consideration given to the general characteristics of the MaSE methodology.

3.1.1 Definition of a Methodology

All software engineering processes require the use of a methodology whose main role is to identify the requisite steps that permit us to proceed from the project requirements to implementation. In other words, a methodology is a guide through the software lifecycle. A methodology provides tools and abstract concepts for transforming a subjective vision of a system (a set of requirements) into an objective formal implementation. It provides software engineers with a map from the original blueprint to final code.

Drogoul and Collinot discuss general traits of methodologies and propose required characteristics of MAS methodologies in their case study of robotic soccer (Drogoul & Collinot 1998). They state that in general, a methodology contains:

1. A structured set of guidelines, steps, advice for the steps, and how to proceed from one step to the next
2. A unified documentation procedure
3. Consistent use of terminology which has a meaning at each step in the cycle
4. The use of conceptual abstractions
5. A comprehensive history of the project for backtracking purposes

The creation of MaSE focused on all of the listed items, with an emphasis on the backtracking mentioned in number five. However, by only considering the listed items, the particulars of multiagent systems would not be figured into the methodology.

3.1.2 Definition of a Multiagent Methodology

The traits described in the previous section are consistent with many design methodologies such as the object-oriented methodologies described by Pressman (Pressman 1992). Agents are in many respects extensions of objects, therefore, object-oriented languages are the languages of choice for agents. However, MASs created using object-oriented methodologies will not take full advantage of agent characteristics. The additional characteristics proposed by Drogoul and Collinot as required for any MAS methodological framework are:

1. Integration of the descriptive and operational aspects of the agent organization during the analysis phase.
2. The possibility of combining a bottom-up approach (designing agents before organization) with a top-down (organization before agents).

Other sources provide similar ideas about the characteristics of a MAS methodology. The first point above is echoed by Wooldridge, Jennings, and Kinny who say that current

software development methods (such as object-oriented) will be unsuitable for MAS design as they fail to capture an agent's flexible autonomous behavior, rich interactions, and complex organizational structure (Wooldridge, Jennings & Kinny 1998). Sycara (Sycara 1998) and DeLoach (DeLoach 1999) also echo the importance of a system-level or organizational concept.

Drogoul and Collinot's second point, combining top-down with bottom-up, is important in a methodology to ensure a designer has freedom to iterate through different phases of the methodology. For example, the details of a particular system could be specified either before or after a system-level organization exists.

MaSE considers both of the above listed points. The structure of the organization is reflected in the ways that roles and agent classes interact through paths of communication and conversations respectively. Furthermore, the phases of MaSE are designed to be iterative in support of the second point, and the transformations from goals to roles to agent classes can be followed backward as well as forward. Both of these points are discussed further in Chapter 4.

3.2 Scope of Research

As stated in Chapter 1, the goal of this research was to create a methodology for developing multiagent systems. Initially, the intended scope of the methodology was much smaller than the full-lifecycle process that resulted. The original scope was to create a methodology that covered the design phase of the MAS software lifecycle; specifically, a collection of agent roles and create agent classes. The following subsections will describe how the scope of this research changed, and what decisions caused it to do so.

3.2.1 Starting Point

The first decision made in the course of this research was to use an existing prototype MAS design methodology as a starting point. The Multiagent Systems Engineering (MaSE) paper by DeLoach (DeLoach 1999), described in Section 2.3.4, was selected based on its strong engineering focus. The methodology created in this thesis actually shares the MaSE name, as discussed in Section 1.5.

The MaSE prototype provided several design diagrams that assist in designing a MAS from a collection of agent classes connected by conversations. The conversations, in particular, were well defined as a state-diagram controlled sequence of messages between agent classes. Agent classes and conversations are the two design objects utilized in the MaSE methodology from the MaSE prototype.

The *Agent Diagram* from the prototype is used in the same manner as described in Section 2.3.4, with agent classes connected by conversations. It is renamed the *Agent Class Diagram* to better reflect its contents. Conversations are still defined by a pair of *Communication Class Diagrams,* but now the two diagrams are labeled for the *initiator* and *responder* halves of the conversation. The *Communication Hierarchy Diagram* is no longer used in the MaSE methodology, as it does not add anything to the design process.

Figure 8 summarizes the state of the MaSE methodology at this early state in its development. Currently, it includes only the three abstractions defined in the MaSE prototype.

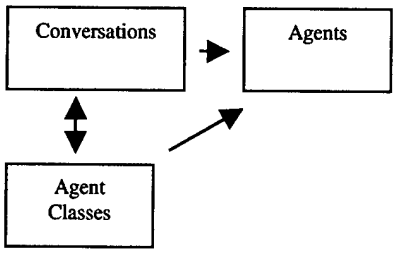

Figure 8: MaSE Creation – First Step

In this and subsequent figures, the darkened boxes represent abstractions used in system design. The flow of the process goes from left to right, as indicated by the arrows. The arrows indicate an influence between design objects. For example, in Figure 8 both Conversations and Agent Classes have an influence upon the creation of agents.

3.2.2 Roles as a Foundation

The decision to base the methodology on the MaSE prototype was made with the understanding that the methodology would cover more of the software lifecycle than the prototype. Specifically, the design portion of the methodology would be based around agent roles. Roles have been used frequently in recent agent design research, such as that described in Sections 2.3.1 (Kendall 1998) and 2.3.3 (Wooldrige, Jennings & Kinny 1999). They are an abstraction that associates a particular "job" with an agent.

Roles were chosen as the foundation of the design process because they are a modular component of agent classes. As such, they provide for easy reorganization during the design process. In this model, an agent class is built by selecting the roles it will play. Additionally, roles can be grouped together to form patterns called role models (Kendall 1998), as described in Section 2.3.1. Role models provide reuse to the design process, which became a very attractive characteristic of the methodology. The addition of roles to MaSE is reflected in Figure 9.

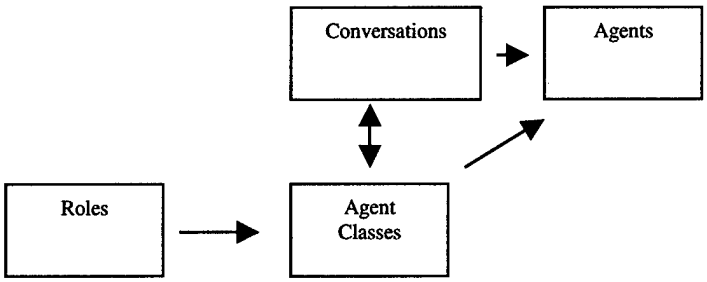

Figure 9: MaSE with Roles

3.2.3 Expanding the Methodology

The addition of roles to the methodology expanded its coverage of the MAS design phase. At this point, research into the analysis phase revealed the possibility of going "back" further into the MAS software lifecycle into the analysis of the system. The research by Kendall and Zhao covered in Section 2.2.1 (Kendall & Zhao 1998) describes how an initial system context can be analyzed to create a structured set of system goals. By integrating this capturing of goals, the methodology would then cover all phases of the MAS software lifecycle from an initial specification to code, as shown in Figure 10.

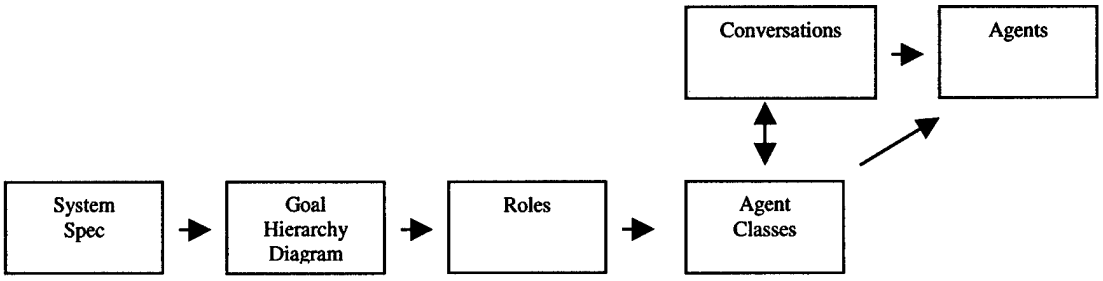

Figure 10: MaSE with Goal Analysis

3.2.4 Creating Phases

A methodology is comprised of a sequence of steps. At this point in development, MaSE was just a progression of system representations, as shown in Figure 10 above. The addition of

"phases" to break up the analysis and design work brought MaSE more in line with what a complete methodology is expected to be. Figure 11 depicts the addition of phases to MaSE, indicated by white boxes.

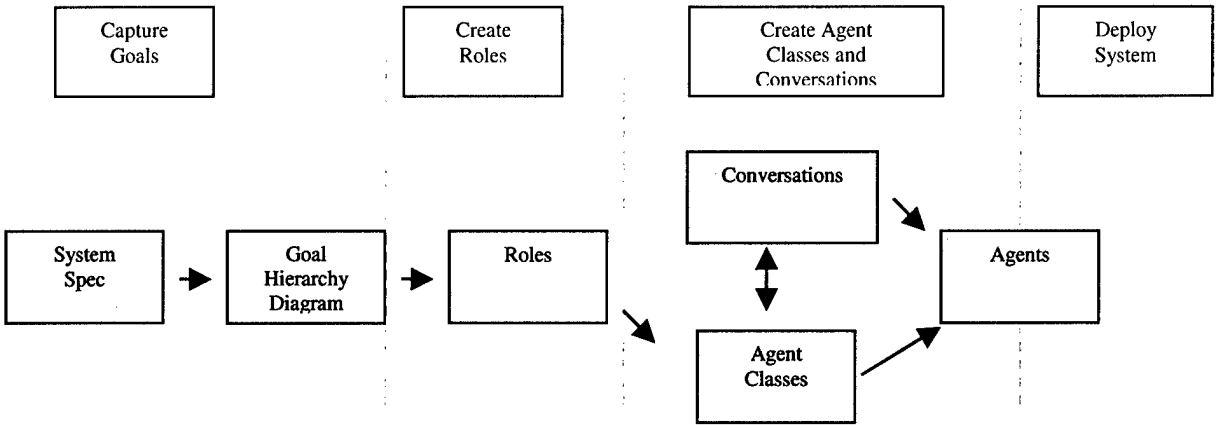

Figure 11: MaSE with Phases

3.3 Bridging Gaps

The version of MaSE shown above in Figure 11 has a scope that spans from an initial MAS specification to coded implementation. In software engineering terms, it covers the software lifecycle from analysis through to implementation. At this point though, there are some missing pieces in the middle. The joining of several existing models left gaps between the joints. In particular, there needed to be connections that transformed goals into roles and roles into agent classes. This section describes those connections in the order that they were considered and added to the MaSE methodology.

3.3.1 Transforming Roles to Agent Classes

A methodology must show how to proceed from one analysis or design object to the next, and give guidelines on how to do so, as discussed in Section 3.1.1. Research by Wooldridge, Jennings, and Kinny (Wooldridge, Jennings & Kinny 1998) has already addressed this problem, and provided some suggestions on how to solve it. Their MAAD methodology (introduced in Sections 2.2.2 and 2.3.3) contains roles as elements of analysis and agent types as elements of design, and discusses how to map one to the other.

This transitional step from one form of representation to another is important enough to a system designer that MaSE has an entire phase that describes how to do so. In brief, roles are combined and mapped to agent classes. An agent class can have one or more agent roles. A one to one mapping is the simplest case, but many combinations of roles may be performed for efficiencies sake. The "Creating Agent Classes" phase of MaSE is described later in Section 4.4.

3.3.2 Transforming Goals to Roles

After a MAS specification is analyzed to produce a Goal Hierarchy Diagram, MaSE must provide some rules for transforming the goals into the roles that will be the foundation of the system design. This is very similar to the transformation from roles to agent classes that appears later in the methodology and is discussed in the previous section.

The first attempts to solve this problem in MaSE focused on the role models discussed in Section 2.3.1 (Kendall 1998). The intent was that role models would be the target of this transformation from goals. This would actually be an indirect method of mapping, since roles come out of role models. The pattern described by a role model has an associated purpose, such as the "mediator pattern" example in Section 2.3.1. The intent was that a goal could map to the "purpose" associated with a role model and an appropriate model would then be found in a

catalog of role models, again, as described in Section 2.3.1. This approach was partially abandoned when no method of indexing such a catalog was found. Furthermore, the notion of a role model "purpose", and how to create a catalog index out of one, was not discussed in the role model paper (Kendall 1998) or the work it referenced.

Therefore, MaSE handles the goals to roles transformation directly, and in a similar manner as roles are transformed to agent classes. In general each goal maps to a role, and may be combined for efficiency. The details of this process are again associated with an entire phase of the methodology. "Transforming Goals to Roles" is the second phase of MaSE, and is covered in Section 4.2.

The state of MaSE after adding the changes discussed in this and the previous section are reflected in Figure 12.

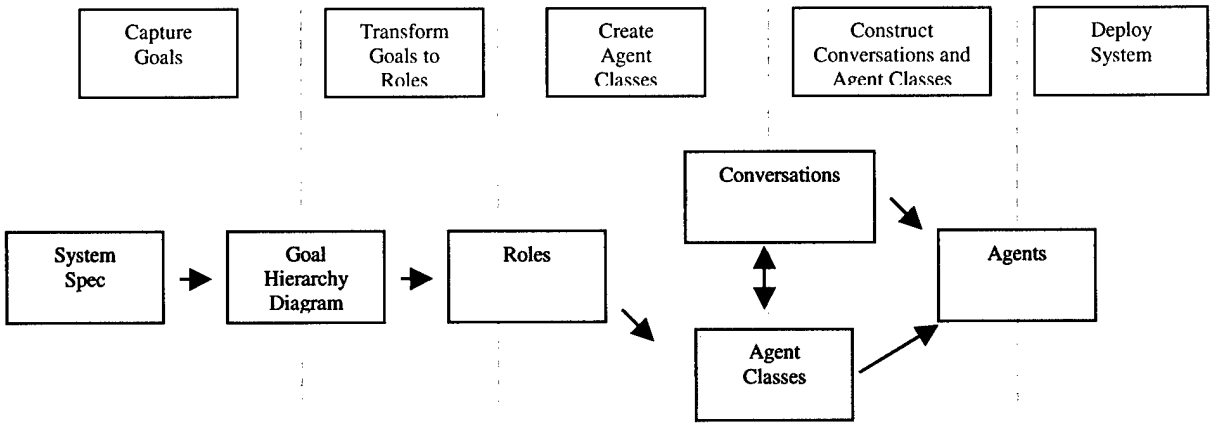

Figure 12: MaSE with Transformations

3.3.3 Designing Agents

There are many significant details of the internal workings of an agent that are not covered in MaSE directly. These details are important in MAS creation since without them, nothing would function. They are avoided in MaSE for two reasons. The first reason is that the depth of scope required to adequately consider such a topic is too much for this effort. The second reason is that those details are being considered apart from this system-wide development methodology in a parallel Agent Research Group thesis by Robinson (Robinson 2000).

MaSE must allow for such details, however, so there is a phase specifically set aside for filling in the details of agent classes. A broad overview of the methods used and guidelines for assembling agents from components is covered in Section 4.6. The phase is called "Assembling Agent Classes" because of the component architecture approach used by Robinson.

Due to the interconnected nature of agent classes and conversations, the MaSE phases that are concerned with filling in their details occur in parallel. The reasoning behind this decision is elaborated in Section 4.6.4. Figure 13 shows this parallel operation.

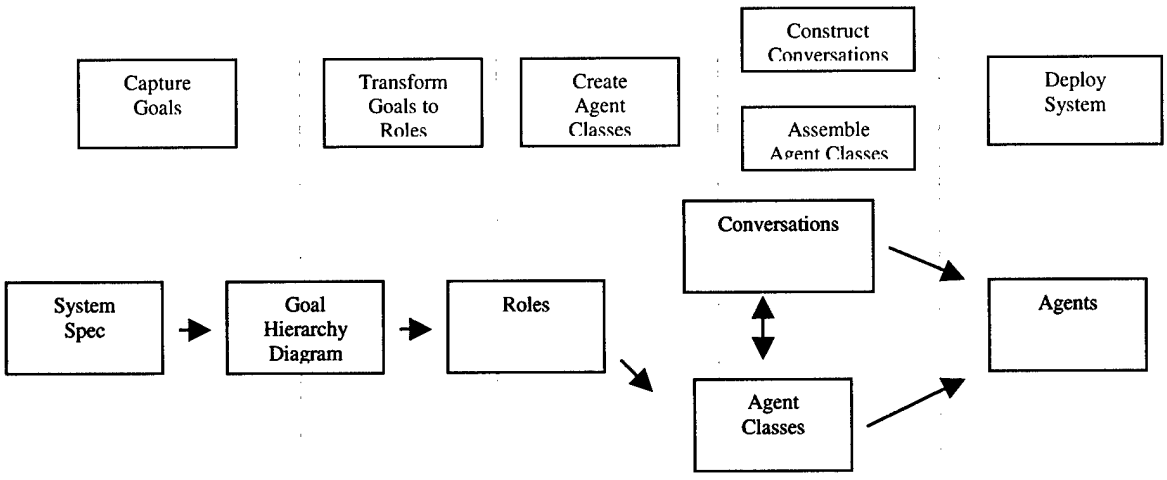

Figure 13: MaSE with Parallel Construction of Agent Classes and Conversations

3.3.4 Constructing Conversations

MaSE conversations are complicated. The representation inherited from the MaSE prototype (DeLoach 1999) is a pair of possible message sequences represented by a state table, as shown in Section 2.3.4. As complex objects, conversations proved very difficult to construct at times. As the purpose of MaSE is to assist a system designer, it was clear that some rules and guidelines for building conversations needed to be a part of the methodology.

At the point in the methodology that conversations are constructed, there were only three other constructs available to assist in their creation: goals, roles, and agent classes. The problem was, that sequence of three abstractions is intended to hold information on *what* the system is supposed to accomplish. Conversation creation was the first point in the methodology that dealt with *how* something needed to be done. Simply adding guidelines on constructing conversations would have enhanced MaSE, but it would be more helpful to the designer if some other type of abstraction supported those guidelines.

3.3.5 Use Cases and Sequence Diagrams

Another way of envisioning conversations are as *actions* that a MAS takes in support of its goals. Any addition to MaSE for the purpose of creating conversations must also be concerned with actions.

Use cases are descriptions of a sequence of actions within a system. They are an example of one way the system is supposed to behave. A typical system would contain several use cases. Collection of use cases during system analysis is included in the research by Kendall and Zhao discussed in Section 2.2.1 (Kendall & Zhao 1998). Use cases are also described in more detail in Section 4.1. The inclusion of use cases during MaSE system analysis captured the notion of *how* a system would work.

Since they are a description of a sequence of events, use cases can also be represented in a manner more appropriate for system design: a Sequence Diagram (shown in Figure 14). A sequence diagram is a step toward the eventual construction of a conversation. It depicts events (arrows) being passed in a particular order between several participants (boxes across the top).

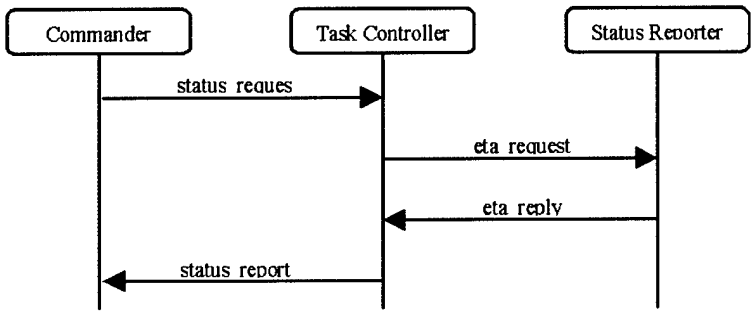

Figure 14: Sequence Diagram

Since the goals captured during analysis are being transformed into agent roles, it was appropriate to have the use cases that were also created during analysis transformed into sequence diagrams that passed messages between those same roles. If a use case is represented as a Sequence Diagram, it is clear that all communication passed between roles in the diagram should eventually become messages in conversations between agent classes in the Agent Diagram.

3.3.6 Tasks as a Design Aid

The final addition to MaSE in support of conversation construction is the concept of a *task*. A task is not a fully defined concept, but it is clear that it has worth in MaSE. A task is a detailed depiction of how a role goes about fulfilling a goal. It consists of a state machine as shown in Figure 15. A task includes communication with other roles, and allows for local state variables. A task also is clearly similar to a conversation.

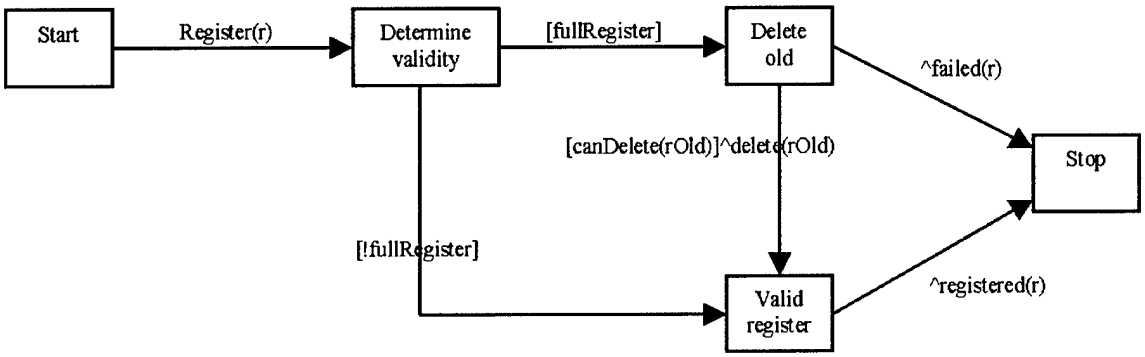

Figure 15: A Task

Agent classes are connected via conversations, agent classes play roles, and roles are connected via paths of communication. Therefore, if a task can be constructed in support of a particular role, agent classes that play that role must pass any messages passed by that task. Also, the messages passed by the agent classes must be contained in a conversation. From a different point of view, any communication with another role in a task means that a path of communication must exist between those two roles. If those roles are parts of different agent classes then a conversation must exist between those classes. On the other hand, if roles sharing a path of communication are parts of the same agent class, then the agent class must handle the event sequences that pass between the roles. The component-based agent design by Robinson (Robinson 2000) includes state-table representations of event sequences that are similar to MaSE conversations.

While a task may not be completely defined, if it has the characteristics described above then it will fit nicely into MaSE in support of conversation construction. Tasks are listed in Section 6.2.1 as an area of future research that would increase the depth of the MaSE methodology.

The addition of use cases and tasks completed the definition of the MaSE methodology as described in Chapter 4. The entire MaSE methodology is represented in Figure 16. Phases are now connected by arrows to show the methodology progression, and are renamed from previous versions to indicate what action is taking place during the phase.

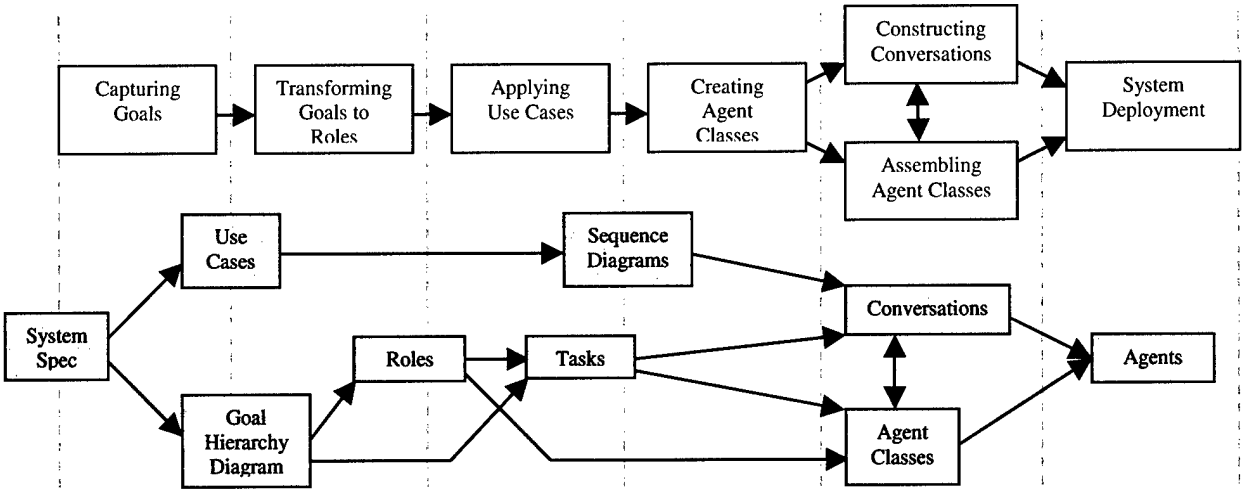

Figure 16: The MaSE Methodology

3.4 Creating a Prototype

A secondary objective of this thesis is to show how an automated tool can follow the MaSE methodology, and help a user create MASs. The agentTool project, as discussed in Chapter 1, is a collaborative effort by AFIT's Agent Research Group. The agentTool system is based on the MaSE methodology, and incorporates elements of the related thesis efforts described in Section 1.5.

An early milestone of this thesis was to create the basic Java code packages of agentTool. It was done early in the thesis process to stress that the MaSE methodology would be designed

from the ground up to support such an implementation. A further advantage was that completing this initial work gave the related thesis efforts something to hook their implementations into. This required that some work on MaSE be completed so that the basic operations of the system would follow the MaSE process. In particular, the use of "tabbed panes" described in Section 5.2 supports the notion in MaSE that all phases of the methodology are accessible, and work can easily switch between different phases.

The latest version of agentTool implements only three of the seven phases of MaSE. It is modular and expandable for future work, including full coverage of the methodology. Creating this application early in the thesis process – just after the decision to include roles as a design object and before any analysis phase work – enforced the consideration of "eventual implementation" on the rest of the creation of MaSE. In practical terms, each decision, after prototype creation, was made based on its ability to be integrated into agentTool.

3.5 Validating the methodology

The validation of MaSE as a legitimate methodology was done at two levels. Pieces were evaluated individually, and the methodology was evaluated as a whole. First, the individual pieces that became the seven phases of MaSE were created, clarified, and appraised using many small examples over the course of development. Since much of MaSE was drawn from existing work, many examples already existed.

For a validation of the entire methodology, test designs were run through the lifecycle from a specification to a collection of agents. The test group included existing system specifications as well as a contrived case for example purposes used in Chapter 4. A set of representative examples used as tests is covered in Chapter 5.

3.6 Summary

The process used to create the MaSE methodology consisted of two steps. The first step was expansion. Starting from the prototype, the scope of MaSE was incrementally expanded to cover the entire lifecycle. The second step was to fill in details, which were transformations between steps in the methodology. The result was a step-by-step methodology that assists a designer in creating a system of multiple agents, based on an initial set of requirements.

A majority of the constructs used in MaSE come from previous work in agent and multiagent analysis and design, described in Chapter 2, providing a strong foundation in established MAS design abstractions. MaSE provides a structure around these constructs, as illustrated above in Figure 16. The order of the constructs and transformations between them, allow each successive system representation to build upon those that came earlier. This eases the burden on the designer when developing complex structures, particularly conversations, that form a multiagent system.

IV. Multiagent Systems Engineering Methodology

The Multiagent System Engineering (MaSE) methodology, takes an initial system specification, and produces a set of formal design documents in a graphically based style. The primary focus of MaSE is to help a designer take an initial specification of an agent system and actually produce agents in code. This methodology forms the basis of AFIT's agentTool development system, which also serves as a validation platform and a proof of concept. The agentTool system also incorporates concurrent and future thesis research that specifies individual agents (Robinson 2000), verifies agent communications (Lacey 2000), and produces a Java implementation of a MAS. Currently, agentTool implements three of the seven MaSE phases. Section 5.1 describes agentTool in more detail.

The MaSE methodology is independent of a particular system architecture, programming language, or message-passing system. A MAS designed in MaSE could be implemented in several different ways from the same design. The future of agentTool should demonstrate this independence by enabling a MAS design to produce code in several different languages including C++ and Java.

The methodology is similar to traditional software engineering methodologies, and specialized for use in creating multiagent systems. The general operation of MaSE follows the progression of steps shown in Figure 17, with outputs from one section becoming inputs for the next. In practice though, the methodology is iterative across all phases with the intent that successive "passes" will add detail to the models described later. These phases form the next seven sections of this paper and will be detailed in order. Figure 17 is a simplification of Figure 16 from the previous chapter that also included all of the data structures involved in MaSE.

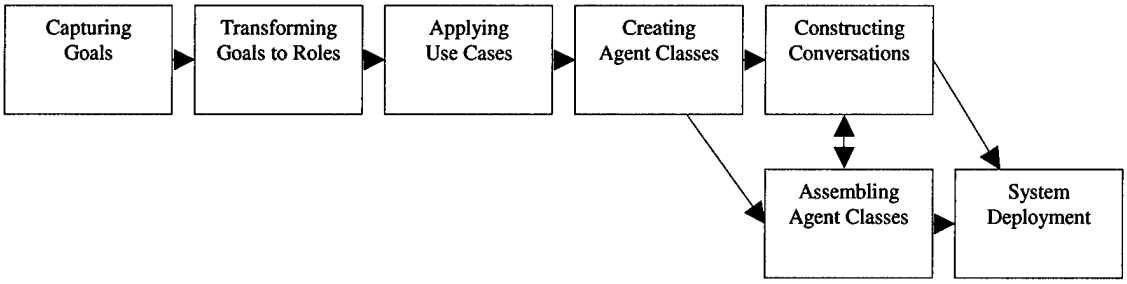

Figure 17: Phases of MaSE

A strength of MaSE is the ability to track changes throughout the process. Every design object can be traced forward or backward through the different phases of the methodology and their corresponding constructs. In this manner, backtracking can be performed to find the initial requirements that a particular agent supports. Furthermore the opposite is true; an early-phase object like a goal can be mapped to a set of later-phase objects. The purpose is to eventually be able to select a design object in agentTool and receive visual feedback on all other objects it affects.

A single example system called the Electronic Intelligence Gathering and Decision System (EGADS) will be tracked across all phases of the methodology to both illustrate details and show how the full process fits together. The requirements of this example form the initial context of the system, and are the inputs to the MaSE methodology. These requirements are attached in appendix A.

4.1 Capturing Goals

The first phase in the MaSE methodology is *Capturing Goals*, which takes the initial system specification and transforms it into a structured set of system goals. In the context of the classic software lifecycle, such as that described by Pressman (Pressman 1992), this phase is concerned with system and software analysis. This section will explain the terms and structures

used in this phase of MaSE, explain the rationale behind the use of goals, and describe the three-step process of capturing and structuring goals with an example using the EGADS system.

There are three steps in *Capturing Goals* that are described in the following subsections. First, the *goals* must be identified from the initial system specification. Next, *use cases* are drawn from the system requirements. Finally, the goals are analyzed and *structured* into a form that can be passed on and used in the design phases of the MaSE methodology.

4.1.1 Capturing Goals - Definitions

In the MaSE methodology, a *goal* is always defined as a system-level objective. Lower-level constructs may inherit or be responsible for goals, but goals always have a system-level context. In consequence, every action within a system must support a particular goal. A goal is a statement that is always phrased as a declaration of system intent, as if it began with the words: "The system shall..."

The *initial system context* is the collection of anything that is given to the designer as a starting point for the creation of the system. It can be of many different forms from a formal requirements document to a collection of user stories, and come from a variety of sources. It is the conceptualization of the system from the user's point of view. The initial system context is the input to this phase, and to the MaSE methodology. Pieces of the initial context may alternately be referred to in a more descriptive manner such as "the system requirements" or "the user specification".

Use cases are descriptions of a sequence of events that is a desired occurrence in the system. They are examples of how the user (or the requirements document editor) thinks the system should behave in a given case. Some system specifications consist largely of use cases.

A standard "if then" statement is a use case. If event A happens, the system must do action B. In fact, a system could be completely defined by "if then" statements, and hence by use cases (though it would be an exhausting process to do so).

The product of this phase of the methodology is a structured hierarchy of goals called a *Goal Hierarchy Diagram* (Figure 18), which was introduced in Section 2.2.1.

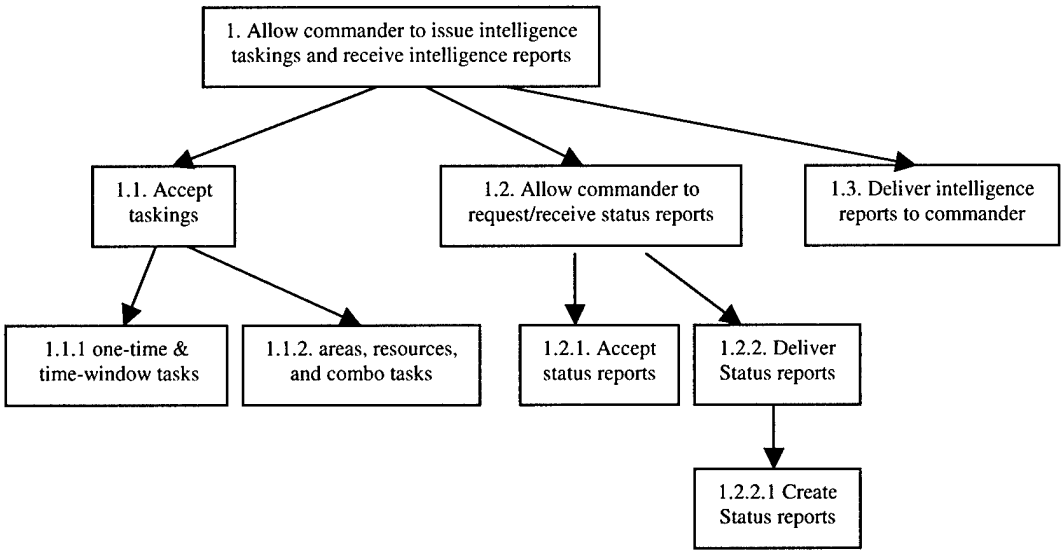

Figure 18: Goal Hierarchy Diagram

The Goal Hierarchy Diagram is modular to allow for modifications, additions, and deletions. The arrows denote sub-goals of a higher-level goal. Each level in the diagram is intended to contain goal "peers" that are at approximately the same level of detail. Though it may appear to be one, the Goal Hierarchy Diagram is not an actual tree structure. This is because identical goals are combined to avoid redundancy in the final step of the Capturing Goals phase, which would "cross" branches of a tree. A Goal Hierarchy Diagram can also be depicted in a list format. Figure 19 is an alternate method of representing the goals in Figure 18.

1. Allow Commander to issue intelligence taskings and receive intelligence reports
    1.1 Accept taskings
        1.1.1 One-time and time-window tasks
        1.1.2 Areas, resources, and combo tasks
    1.2 Allow Commander to request / receive status reports
        1.2.1 Accept status reports
        1.2.2 Deliver status reports
            1.2.2.1 Create status reports
    1.3 Deliver intelligence reports to commander

Figure 19: Goals in list form

4.1.2 Capturing Goals - Rationale

The first phase of the MaSE methodology is based on goals because they are a highly stable structure upon which to build a design. The general procedure of the Capturing Goals phase is to capture what is important from the requirements in a structured and modular form. MaSE uses goals as the requirements encapsulation because goals embody *what* the system is trying to achieve and will generally remain constant throughout the rest of the analysis and design process. This is in contrast to other possible analysis objects that are organized around requirement details that specify *how* something is done, such as use cases. In those cases the details can be overwhelming and often change (Kendall & Zhao 1998).

4.1.3 Capturing Goals - Substeps

There are several sub-steps to the Capturing Goals phase of MaSE. These are listed in the following subsections.

4.1.3.1 Capturing Goals - Identify Goals

The first step in capturing goals is to distill the essence of a set of requirements. These requirements may include detailed technical documents, user stories, or formalized government specifications. This begins by extracting scenarios from the initial specification and describing the goal of that scenario. The EGADS system requirements (see Appendix A) list an overall goal and several general requirements and prohibitions. Determining the purpose for each scenario listed in the requirements identifies goals. The EGADS system designer has extracted the goals shown in Figure 20.

Once these goals have been captured and explicitly stated, they are less likely to change than the detailed steps and activities involved in accomplishing them. They provide a foundation for the analysis model. Note that it is acceptable to remove detailed information when specifying goals. A placeholder in the goal should reference details, for example: "Meet the timing constraints as described in technical report #1."

- Allow commander to issue intelligence taskings and receive intelligence reports
- Link to non-homogeneous intelligence assets
- Accept tasking for areas, resources, and combinations
- Accept one-time and time-window taskings
- Allow commander to request and receive status reports
- Allow commander to set presentation preferences
- Ensure that intelligence reports go to intelligence section before commander
- Allow addition and deletion of data sources

Figure 20: EGADS Goals

4.1.3.2 Capturing Goals - Create Use Cases

The next step in the Capturing Goals phase is to distill use cases from existing requirements or other available resources. Since use cases are used later in the methodology to create ordered event sequences called Sequence Diagrams (Section 4.3), they must be identified early in MaSE. The Capturing Goal phase is the logical spot to create use cases since their creation may help to gather more information or clarify existing information about system goals. Use cases may already exist as part of the initial system context, or they may have to be extracted by the designer. They can be extracted from the requirements specification, user stories, or another available source. If the user is available this is the time to ask, "What should the system do if *this* happens?"

It is important not to let the analysis of scenarios and creation of use cases get out of hand and never end. A MaSE designer need only create enough use cases to cover potential paths of communication, not all possible combinations of messages and data. This is due to how the use cases will be used in the third phase of MaSE (Applying Use Cases). A good guideline is that every sequence of system events that differs significantly from others should be formed into a use case. A significant difference could be a different participant involved in a message sequence, a data stream that goes in a reverse direction, or any objects that are involved in a different order than in a previous message sequence.

EGADS includes an example that will translate nicely into a use case. Since the example suggests a sequence of events, but doesn't specify them, the EGADS designer asks a user what the intended sequence should be. Here is the complete use case the user creates:

A commander desires to know what sort of air defenses will exist in a target area of an air strike that must be executed within 72 hours. At stake is when will the strike be launched, and what SEAD (Suppression of Enemy Air Defense) assets must be included in the sortie package.

The commander prepares and executes an intelligence tasking specifying air defense units, the target zone, and a 72-hour time window with a due date of 6 hours. The tasking is distributed to mission controllers who have the ability to detect such units. In this case the tasking is sent to a satellite controller and a JWICS data controller.

Both controllers have data on the position available. The JWICS data is immediately available, and the Sat info takes a few hours to receive. A status request by the commander during this time frame would indicate this. The data indicates a medium AAA presence and a light infantry-based SAM presence.

The data, once obtained, is then collated and sent to the intelligence processing section. An intelligence analyst works the request and knows from past experience that historical data may be useful in such situations. She expands the search to backtrack several days and widens the search area to cover more of the opposition's forces, but focuses the search to only check the faster-retrieving JWICS data in order to make the 6-hour due date. The new data indicates that two units of mobile SAMs are heading toward the target area from rear positions, and are expected to be operational in 48 hours.

The entire report is sent to the commander, who realizes an attack within the next 36 hours would not require usage of extensive Wild-Weasel (anti-SAM) assets. He orders the attack.

Even with an incomplete grasp of military-specific language and acronyms that the user employed, this use case provides many bits of information. First, it illustrates a path of action throughout the system. Second, it introduces some new concepts to the goals including *mission controllers*, and a human *intelligence analyst* who has the ability to resubmit search parameters. The designer decides that the mission controllers are distributed controllers for different types of data-gathering assets (the satellite and JWICS controllers being two examples) since they accept taskings from the system and distribute them to different types of data-gatherers. There is not a system-level goal that can be gleaned from this information, so the designer files it away until a later phase in the methodology. The human intelligence analysts do have a system-level goal associated with them, since they are system-external entities that have a specific need: re-submitting searches. Therefore, the designer adds one further goal to the list of system goals:

- Allow intelligence analysts to refine and resubmit searches

In addition to the use case above, the user describes one other sequence of events to the designer of the EGADS system. It does not elicit any additional goals, but will be used later when applying the use cases.

> A new system comes on line that interfaces in real-time with JSTARS data. It is input into the system, advertising its capabilities. A new mission controller is setup to handle the interface to JSTARS.

This is a sequence that is not initiated by the commander, so it is worthy of note to the designer as a use case that applies to some other area of the system. The problem is that it is not obvious how this use case is started. The EGADS designer decides that a system administrator is needed for this since it describes an addition to an existing system. Since there is an existing goal that requires the EGADS system to "Allow addition and deletion of data sources", the administrator could also perform the complementary role of removing old data sources. This use case will also be stored for use in a later phase of MaSE.

The EGADS designer is aware of the entire MaSE methodology. In both cases where the designer stored information for "later" he was intending to create a *role* in the system for that particular object (mission controller and system administrator interface). The notion of roles will be discussed more in the following section. What is noteworthy is that MaSE does not preclude the designer from momentarily skipping forward to the creation of roles and then returning to capturing goals. It is intended to work in this manner. Specifically, at the point in the example where the EGADS designer stored the information for later, a MaSE designer should begin a list of system roles as described in the second phase of MaSE. This characteristic of MaSE is intended to be taken full advantage of in an automated tool such as agentTool (described in Section 5.1) where the diagrams and objects of all design phases are available simultaneously.

Use cases are most valuable in MaSE for helping lay down paths of communication that will later become conversations between agents. With this in mind, the designer should attempt to gather enough use cases to cover as many possible sequences of events as possible. On the other hand, a particular sequence need not be repeated many times with different data or types of messages since these sequences will be used only to determine the minimum required communication paths.

The most common form of a use case is the *positive use case* illustrated so far. That is, something that is desired to happen in the system. Other types of use cases may provide needed information though, depending on the system being developed. A second form of use case is a *failure use case*, which is a sequence that is still desired to occur within the system, but is illustrative of a breakdown or error. Failure use cases are similar to exception handling in some programming languages. At the end of a block of program code the exception statement says, "If something goes wrong in that block of code do this." In our example, a failure use case is an intelligence request not being handled by its due date. If this happens, a message will be sent to the commander and the intelligence section describing the failure.

The third type of use case, called a *negative use case*, is a sequence that is not desired to occur within a system. In EGADS, an example of a negative use case occurs when the commander receives an intelligence report that has not been processed by the intelligence section. Neither failure use cases nor negative use cases are currently utilized in MaSE (see Section 6.2 – future research areas).

4.1.3.3 Capturing Goals - Structure Goals

The final step in this phase is to structure the goals by analyzing them for importance, and construct a Goal Hierarchy Diagram. It is important to remember that, while it is good to

keep related goals together, modularity is also beneficial. So far scenarios and goals have been captured, but they are of various importance, size, and level of detail. Some scenarios always occur while some conditions seldom occur. The goal hierarchy diagram preserves such relationships, and divides goals into levels of detail and importance that are easier to manage and understand.

The most important goals must be recognized by the designer and placed at the top of the Goal Hierarchy Diagram. In EGADS, the most important goal is the first one, "Allow commander to issue intelligence taskings and receive intelligence reports". The EGADS designer determines this by finding the primary purpose of the system at the top of the system requirements: "EGADS links a Commander to intelligence gathering assets in the field".

The goals should be structured so that the main sequences of interaction and subordinate details can be distinguished from one another. All of the sub-goals must pertain to their parent goal in the hierarchy. The important characteristic of the hierarchy to maintain is that all sub-goals relate functionally to their parent. In other words, they relate to the same functions or mode of operation as their parent, but at a lower level. In some cases, sub-goals may logically (completely) sub-divide their parent as well, in which case the parent goal need not be mapped to a role in the next phase of MaSE (Section 4.2). At all levels in the hierarchy, goals supporting the primary mode of system operation should be separate from those that indicate an alternative mode.

The EGADS designer notices that many of the other system goals are in support of the primary goal listed above. For example, the ability of a commander to request and receive status reports about intelligence taskings is subordinate to the goal of initiating intelligence taskings and receiving intelligence reports. Furthermore, there are many goals in EGADS that are easily

broken up into sub-goals. The "request and receive status reports" goal identified above breaks readily into the "request" and "receive" pieces. This goal partitioning is done because goals that are very closely related to each other, such as the two sub-goals just mentioned, are not necessarily best handled by the same agent role. As will be discussed in Section 4.2, roles are drawn directly from the Goal Hierarchy Diagram.

Finally, some goals are clearly not in direct support of the primary system goal, but are an important part of the system. In that case, another "branch" of the Goal Hierarchy Diagram is created for these goals. In EGADS, the ability to connect with a changing set of many different data sources is an important system goal, but not an appropriate sub-goal of the primary goal of issuing intelligence taskings and receiving intelligence reports. Accordingly, the EGADS designer assigns it to a peer goal of the primary goal. In the EGADS example, the process of structuring goals described above continues and the EGADS system designer comes up with the structured goals shown in Figure 21.

The goals in Figure 21 can be easily re-written as a tree diagram since they are based on a numerical hierarchy. The primary goal, goal 1, has sub-goals 1.1, 1.2, etc. The designer need only remove duplicate goals from the tree to form a Goal Hierarchy Diagram. The Goal Hierarchy Diagram for EGADS is shown in Figure 22.

There are potential problems at this point that MaSE does not attempt to consider. Kendall and Zhao (Kendall & Zhao 1998) describe a few remaining problems including having a large number of goals that should be differentiated, and having difficulty determining a system's goals. A large number of goals can be overwhelming to an observer, which eliminates the Goal Hierarchy Diagram advantage of being a simplified representation of the relationships between goals. It may be useful in this case to differentiate between commonly occurring types of goals such as strategic goals, summary goals, user goals, and sub-functions; perhaps by having a different representative shape on the Goal Hierarchy Diagram. The potential difficulty in determining a system's goals, and possibly reaching a consensus regarding them, is a significant problem in larger and more complex systems.

1. Accept intelligence taskings from commander and issue intelligence reports
    1.1. Accept taskings
        1.1.1.Accept one-time and time-window taskings
        1.1.2.Accept tasking for areas, resources, and combinations
    1.2. Allow commander to request and receive status reports
        1.2.1.Accept status reports requests
        1.2.2.Deliver status reports
            1.2.2.1. Create status reports
    1.3. Deliver intelligence reports to commander
        1.3.1.Create intelligence reports
        1.3.2.Allow intelligence analysts to refine and resubmit searches
            1.3.2.1. Notify analysts of intelligence tasking
            1.3.2.2. Ensure that intelligence reports go to intel section before commander
        1.3.3.Ensure that intelligence reports go to intel section before commander
        1.3.4.Allow commander to set presentation preferences
2. Link to non-homogeneous intelligence assets
    2.1. Allow addition and deletion of data sources
        2.1.1.Allow addition of data sources
        2.1.2.Allow deletion of data sources

Figure 21: Structured EGADS Goals

At the conclusion of the Capturing Goals phase, the system goals have been analyzed, captured, and structured in a Goal Hierarchy Diagram. Additionally, the designer has a collection of use cases to apply later in the methodology. Now, MaSE proceeds to the second phase of design where the foundation of the actual design process is introduced: agent roles.

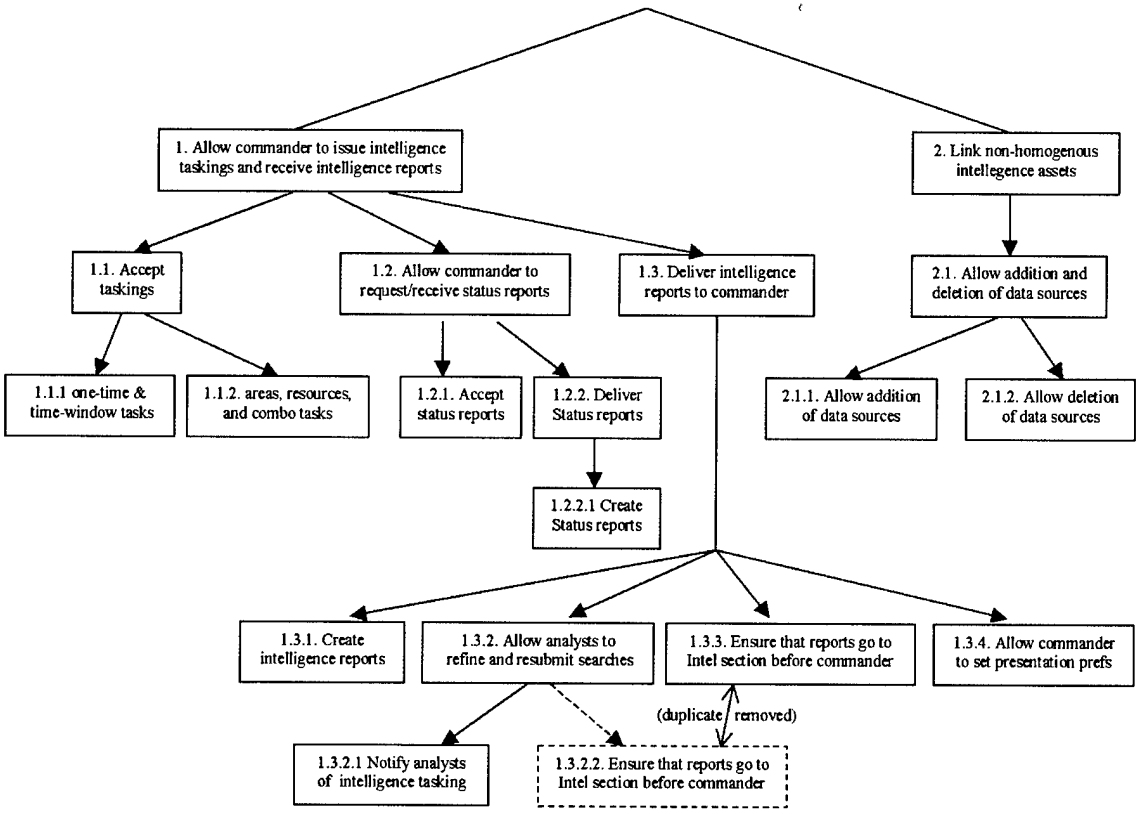

Figure 22: EGADS Goal Hierarchy Diagram

4.2 Transforming Goals to Roles

The second step of MaSE is to transform the structured goals into a form more useful for constructing multiagent systems: roles. Roles are the building blocks of agents in MaSE. As described in Section 3.2.2, roles represent system goals during the design phase. By using roles in this manner, the system goals are carried forward into the system design. The system goals

are satisfied because every goal is associated with a role, and every role is played by an agent class.

This section, and those dealing with subsequent phases, follows the same pattern as Section 4.1. First, new concepts appropriate to the given phase will be defined in subsection 1. Second, rationale related to the phase will be presented in subsection 2. Finally, details of the phase actions and considerations will be described in the remaining subsections.

4.2.1 Transforming Goals to Roles - Definitions

A role has been discussed several times previously in this thesis. The role definition used in MaSE was first presented in Section 2.1. It is an abstract description of an entity's expected function containing system goals that it has the responsibility of fulfilling. Roles are created to *do* something. They are more or less identical to the notion of an actor in a play or an office within an organization.

In addition, a role may contain a collection of *tasks* for which it has responsibilities. MaSE tasks were described in Section 3.3.6 as a detailed depiction of how a role fulfils a goal, depicted as a state diagram.

A collection of roles, called a *role model* as described in Section 2.3.1 (Kendall 1998), is very useful for reusing roles from previous systems designs, or from a role model catalog. A role model is an abstraction for modeling and designing agent systems. The basic idea is that patterns of agent roles are constructed, labeled, and archived. When a new system is designed, the patterns are recognized and a role model can be re-applied from an archive, resulting in a collection of agent roles that satisfy a subset of the system goals. As shown in Figure 23, the arrows on role models are paths of communication connecting roles, and the dots indicate multiplicity.

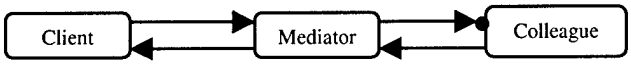

Figure 23: A Role Model

4.2.2 Transforming Goals to Roles – Rationale

Roles are the foundation of MaSE design, as discussed in Section 3.2.2. Specifically, they are used to build agent classes. Furthermore, roles contain the set of system goals defined in the analysis phase. In other words, roles form a bridge from what the system is trying to achieve (the analysis phase and goals) to how it goes about achieving it (design phase agent classes). The organization and allocation of roles among agent classes in a MAS can be easily changed by the designer, since roles can be manipulated modularly. This allows consideration of load issues. For example, a high communication volume between two roles could imply that those roles should be part of the same agent class. Also, two roles with high processing power requirements should perhaps be played by two different agent classes so they can be distributed to separate CPUs.

4.2.3 Transforming Goals to Roles - Details

The general case transformation of goals to role is one-to-one; each goal maps to a role. However, there are many exceptional situations where it is useful to combine goals. Similar or related goals may be combined into single roles for the sake of convenience or efficiency.

If possible, goals are mapped to pre-defined role models. This is done when a goal suggests a particular role model pattern. For example, the goal to "Employ centralized distribution of tasks" brings to mind the mediator pattern role model shown above in Figure 23. The mediator pattern shows that three roles would be required: the central mediator, a client that introduces the tasks, and a colleague that is played by all the agents that can be tasked. In support

of this, a library of role models can be created from patterns constructed during design. These patterns can then be applied to future designs. There is an existing catalog of role models described in related work (Kendall 1998); though, as described in Section 3.3.2, there is not yet a method of indexing the models that supports reuse. In other words, there is no easy way to explain how a designer "recognizes" a particular pattern. This problem is described in Section 6.2.3.

There are many considerations in transforming goals to roles. Some common goals imply particular roles, and may be reused. For example, many agent systems (though not the EGADS example) employ a type of agent registration system. A registrar role should be used in these cases.

Some system goals may be unstated in the requirements, but implied by other goals. They should be incorporated into roles as well. For example, interfacing with a user is likely a requirement. In the EGADS example it is implied that an intelligence analyst be able to access raw intelligence data for processing, and then enter results into the system. If an implied goal is discovered at this point in system analysis, it should be added to existing goals as if it were part of the stated system requirements. The subsequent methodology steps are then performed, such as adding the new goal to the Goal Hierarchy Diagram and incorporating it into a role.

Similar goals can be combined into a single role. For example, a set of goals that enforce constraints upon a schedule could be combined as follows. The goals of "No student or instructor may have a class during both the first and last meeting hours of the day" and "No course may have less than 3 students" and "All students must carry at least eight credit hours" could all be combined under the role of "Constraint Enforcer".

Some goals imply distributed roles. Any mention of separate machines or other distribution requires one role for each "side" of the distributed relationship. Interfacing with an

external source is the same. One role must interface with the source and another may be required to bridge the gap back to the system. This is also true for any database or file interface inside of the system. In addition, any data persistence implies some sort of information server for that data, and an appropriate server role as well.

A user interface implies a role by itself and should be separate from other roles as if it were a separate data source. The roles may be combined in the future into a single agent, but for now they are separate.

Broadcasting a single message to multiple recipients implies having a broadcast manager of some sort to register the members of the broadcast group. Again, this role could end up being that of the same agent as the agent registrar.

Any participants in the use cases created during the "Creating Goals" phase of MaSE should imply a role in support of the goal involved in that use case. A participant in a use case is any entity that sends or receives information as part of the sequence of events.

Optionally, annotating the Goal Hierarchy Diagram with roles also helps decide how to divide goals, since the diagram has related goals as neighbors. The designer draws boxes around goals that are to be combined, indicating the division of roles.

4.2.4 Transforming Goals to Roles - Example

Roles can be stored as a simple list. The EGADS designer takes the Goal Hierarchy Diagram and creates the roles shown in Figure 24 (parenthesis indicate goals associated with the roles). The Commander Interface and Analyst Interface are created because separate roles are needed for a user interface. The Mission Controller and Data Source Interface roles form two sides of a distributed relationship, since it is clear that the data sources must reside across a network. The Status Reporter and Registrar Roles support particular goals, and were also participants in use cases identified earlier. Finally, the Task Controller role is created as a

representative of a particular task. It is the "system" that the other roles talk to. Each task controller ensures that its intelligence task is formatted, handled, and routed correctly.

• Commander Interface	(1, 1.2, 1.3, 1.3.4)
• Analyst Interface	(1.3.1, 1.3.2, 1.3.2.1)
• Mission Controller	(2)
• Data Source Interface	(2)
• Status Reporter	(1.2.1, 1.2.2, 1.2.2.1)
• Registrar	(2.1, 2.1.1, 2.1.2)
• Task Controller	(1.1, 1.1.1, 1.1.2, 1.3.3)

Figure 24: EGADS Roles

4.2.5 Transforming Goals to Roles - Tasks

After roles are created, tasks may be associated with each role. Every goal associated with a role can have a task that details how the goal is accomplished. This must be done after role creation since tasks communicate with tasks in other roles. The creation of interacting state diagrams can be a very complicated procedure, which MaSE tackles to some degree when discussing construction of conversations in a later phase.

Figure 25 shows a sample task created for EGADS. It is the registration task for the Registrar role. The task begins in the start state and then receives a "register" message from role *r* on behalf of *r*'s agent. The task checks to see if the register is full – indicated by the boolean value "fullRegister". Depending on the value of fullRegister, the task splits into two paths. The "Delete Old" path checks to see if any old roles (rOld) and their corresponding agents can be deleted to make space in the register – if so, it sends them a delete message. Finally, a "failed" or "registered" message is sent back to the role *r*. Role *r* in this case would have a corresponding task that sends the initial message and waits for a response.

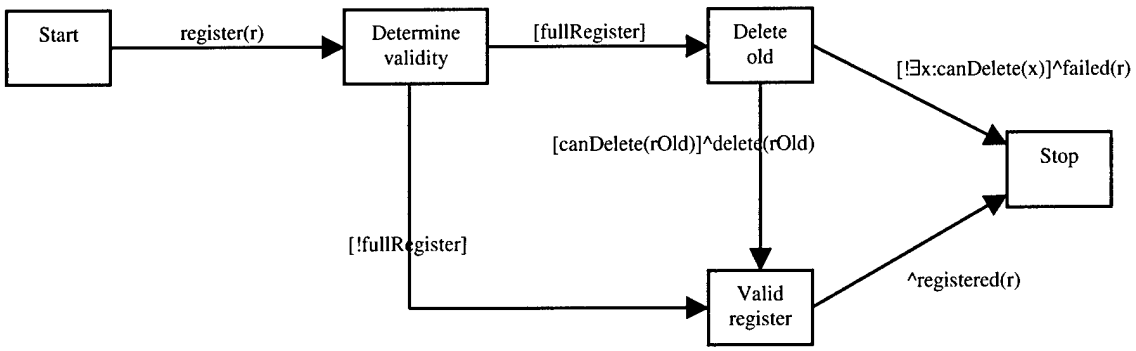

Figure 25: EGADS Registration Task

4.3 Applying Use Cases

After creating the roles that will be used to build the agent system, the next step in the MaSE design process is to construct agent classes from the roles. When the agent classes inherit communication paths between roles, they form conversations with other classes. It is these conversations that are the real backbone of a MAS, as they bridge the distribution that is the strength of agent technology. As described in the Constructing Conversations MaSE phase in Section 4.5, they can also be very difficult to construct. The third phase of MaSE looks down the road toward constructing these conversations and attempts to ease this difficulty.

4.3.1 Applying Use Cases - Definitions

Applying use cases requires taking the use cases identified in the capturing goals phase and restructuring them as a *Sequence Diagram* (Figure 26). A sequence diagram depicts a sequence of events between multiple processes. In MaSE, the different processes are different agent roles. The events being passed are messages between the roles. Sequence Diagrams were introduced in Section 3.3.5.

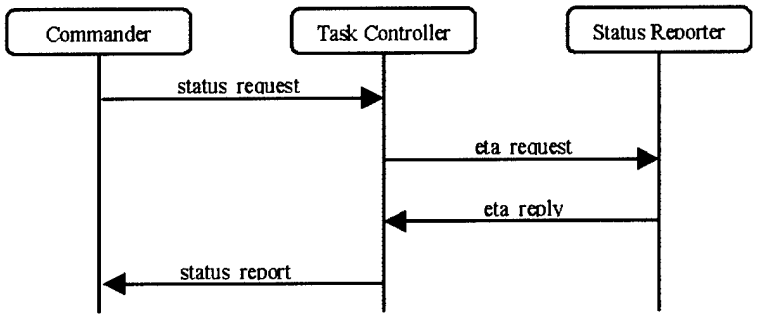

Figure 26: Sequence Diagram

4.3.2 Applying Use Cases – Rationale

The purpose behind the application of use cases in MaSE is to support the construction of conversations later in the methodology. The transformation of use cases into Sequence Diagrams preserves desirable event sequences, based on system goals. These event sequences are captured for eventual inclusion in conversations, ensuring that they will exist in the MAS.

A Sequence Diagram is used to determine the minimum set of messages that must be passed between roles. If a message is passed between two roles, then there must be a corresponding communication path between them. A high communication volume between two roles may suggest combining the roles into a single agent class, since agents that play these roles inherit the communications between them. Also, a communication path between roles played by separate agent classes means that a conversation must exist between the two agent classes to pass the message. The agent class playing the role that initiated the communication becomes the *initiator* of that conversation, and the receiving agent class becomes the *responder*.

4.3.3 Applying Use Cases – Details and Example

Transforming a use case into a Sequence Diagram is straightforward. First, the system roles that partake in the events are identified. Every participant in a MaSE Sequence Diagram is

a role. If there is a participant in the use case that is not a role in the system, a new role must be created.

In general, one Sequence Diagram is created for each use case. Typically it is only possible to create one sequence from a use case. However, if there are several possibilities, then make multiple Sequence Diagrams. An example of the need to create multiple Sequence Diagrams is when a use case has several alternate resolutions, such as "The diagnosis is sent from the doctor to the medical desk, and from the medical desk to the patient unless the patient is a minor, in which case it is sent to the patient's legal guardian from the medical desk." This results in two similar but distinct Sequence Diagrams that both need to be supported by conversations.

The Sequence Diagram created by the EGADS designer from the large use case in Section 4.1.3.2 is shown in Figure 27. The use case has clearly identified participants, with the exception of the Task Controller role. This role was created earlier by the designer for the express purpose of accepting tasks from the commander and parceling out requests to appropriate Mission Controllers though, so it is a logical choice for this Sequence Diagram.

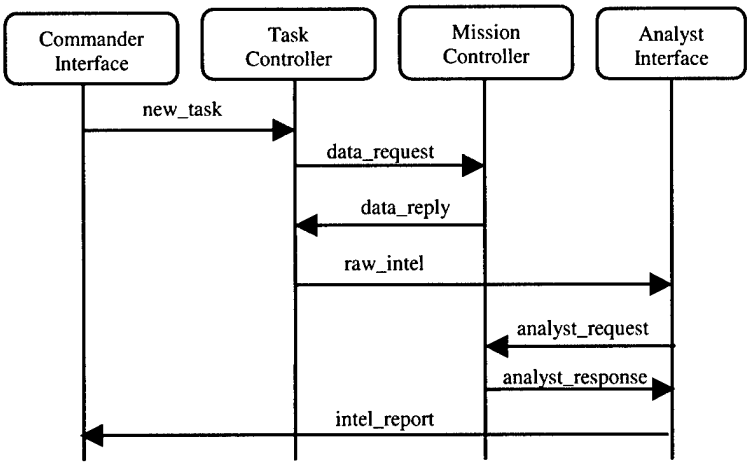

Figure 27: EGADS Sequence Diagram

After identifying the participating roles, creating the Sequence Diagram consists of reading through the use case and finding all instances of an event that occurs between two of the roles. Each event in the use case is drawn as an arrow on the Sequence Diagram in the order that they occur. The EGADS designer does this easily until the part of the use case that concerns the intelligence analyst. The analyst has an alternative of whether to resubmit a search for more data. However, the two-event resubmission sequence would not break the order of the other events, so it does not require a new Sequence Diagram to describe it under the "alternate resolution" rule described above. In other words, the addition or deletion of the "resubmission" events does not affect the sequence of the other events. Therefore the sequence that does not include a resubmission is covered by the one that does and the same conversations result.

The Sequence Diagram is also consistent with the system goal that the Commander can never receive raw intelligence data. An intelligence report can only be sent to the Commander by the intelligence section.

By applying the use cases to create Sequence Diagrams, all potential sequences of events will be accounted for in the conversations that will be designed from these use cases. Furthermore, since Sequence Diagrams operate between the system roles, creating them also creates communication paths between the roles. If desired for future reuse or clarity, role models can then be built from the roles, such as that created by the EGADS designer in Figure 28. A role model need not be created when the role relationships are clear (such as simple cases), or if the pattern already exists in an available catalog of role models.

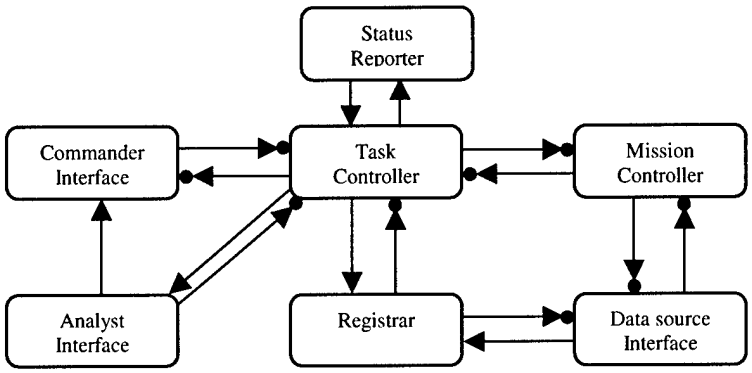

Figure 28: EGADS Role Model

4.4 Creating Agent Classes

In the Creating Agent Classes phase of the MaSE methodology, the agent classes are identified from component roles. The product of this phase is an Agent Class Diagram, which depicts agent classes and the conversations between them.

4.4.1 Creating Agent Classes – Definitions

An agent class is a template for a particular type of agent that will be in the system, just as an object class is a template for objects. During this phase of MaSE, agent classes have two component sets: roles and conversations. In a later phase, described in Section 4.6, internal details are added to agent classes. Each role is played by an agent class. Agent classes can play many roles, and can change roles dynamically. Furthermore, agents of the same class may play different roles at the same time. The conversations of an agent class are all those that it is a participant in, either as an initiator or responder.

The agent classes within a system are described using an *Agent Class Diagram*, which is similar to many object diagrams. The primary difference is the semantics of the relationships between agent classes. In Agent Class Diagrams, these relationships define conversations that are

held between agent classes. A sample *Agent Class Diagram* is shown in Figure 29. The boxes in the figure are the agent classes, containing the class name. Lines with arrows denote conversations and point from the *initiator* of the conversation to the *responder*, with the name of the conversation written either over or next to the arrow.

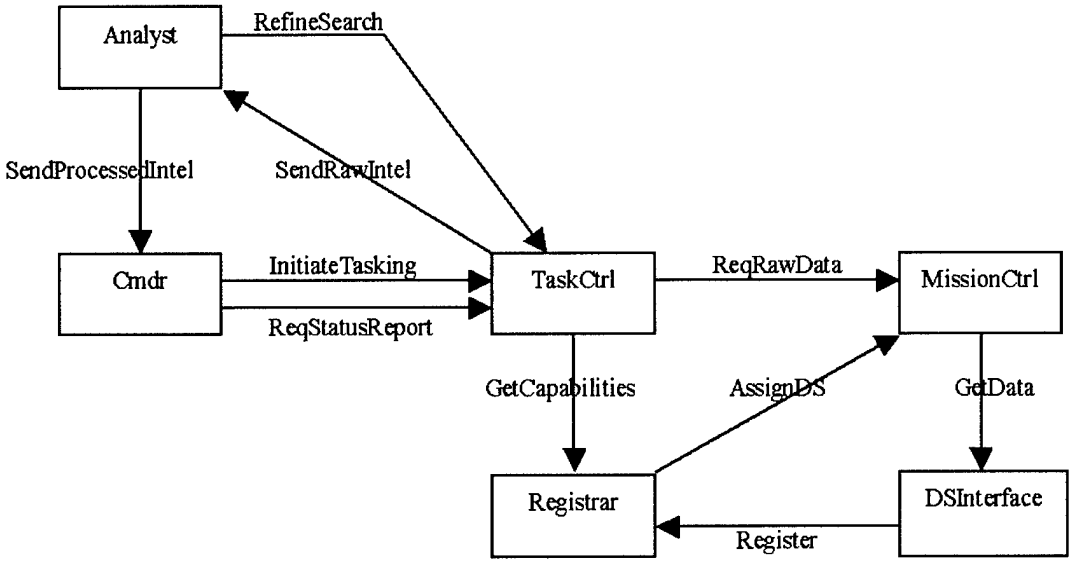

Figure 29: EGADS Agent Class Diagram

4.4.2 Creating Agent Classes – Rationale

The Agent Class Diagram is the first design object in MaSE that depicts the entire multiagent system. Having proceeded through the methodology to reach this point, the system represented by the diagram supports the goals and use cases identified in the first phase of MaSE. Of particular importance is the "shape" of the system – the way that the agent classes are connected with conversations. In fact, the primary purpose of this phase is to identify the agent classes that participate in each side of a conversation.

Earlier it was stated that roles are the "foundation" of MaSE. In that case, agent classes are the "bricks" that actually build the system. The reason that there are two different abstractions is that they provide the ability to manipulate two different system dimensions separately. The roles supply a modular way to arrange the goals of the system apart from higher-level considerations. On the other hand, the agent classes can be manipulated with consideration to communications and to system resources such as databases and external interfaces, without worrying about the system goals.

4.4.3 Creating Agent Classes – Details and Example

Just as before, when mapping goals to roles, there is generally a one-to-one mapping between roles and agent classes. However, the designer may combine multiple roles to make a single agent. Since agents inherit the communication paths between roles, any paths between two roles become a conversation between their respective classes. As such, it is desirable, where possible, to combine two roles that share a high volume of message traffic. When determining which roles to combine, size and frequency of communications are important, not just the number of communication paths. On the other hand, distributed resources must be handled by separate agents; so many roles must map to their own agent class. It is also important to identify which agent classes are required to interface with external resources such as humans, other software tools, and data stores.

If a role communicates only with one other role, it indicates that the two roles may be combined. This is what the EGADS designer decided to do with the "Status Reporter" role in Figure 28. It is combined with "Task Controller" to create the "TaskCtrl" agent class in the EGADS Agent Class Diagram shown in Figure 29. Other than this combination, the EGADS designer performs a one-to-one transformation from roles to agent classes. Next, the designer

maps the paths of communication between roles from Figure 28 into conversations. Some of the communications between roles in Figure 28 were two-way arrows that were turned into a single conversation, since information can flow both ways in a conversation. For example, the two communications between the "Task Controller" role and the "Registrar" role become the "GetCapabilities" conversation in Figure 29. Additionally, a single path of communication may become multiple conversations, such as the "InitiateTasking" and "ReqStatusReport" conversations. In that example, the path of communication between the component roles was drawn from multiple Sequence Diagrams: the "New Task" diagram shown in Figure 27, and the "Status Report" diagram shown in Figure 26.

4.5 Constructing Conversations

Constructing Conversations is the next phase of MaSE. It can happen before, after, or in parallel with the succeeding phase of Assembling Agents. The two phases are closely linked and, as will be discussed in Section 4.6.4, it is often beneficial to go back and forth between the phases. Up to this point, communications between agents have not been detailed beyond stating that they exist. At this point, the fact that a conversation must happen between two agents is known; now the particulars of the conversation are fleshed out.

4.5.1 Constructing Conversations - Definitions

A MaSE conversation defines a coordination protocol between two agents. Specifically, a conversation consists of two Communication Class Diagrams, one each for the initiator and responder. A Communication Class Diagram is a finite state automata that defines the conversation states of the two participant agent classes, as shown in Figure 30. The initiator always begins the conversation by sending the first message.

When an agent receives a message, it compares it to its active conversations. Upon a match, the agent transitions the appropriate conversation to a new state and performs any required

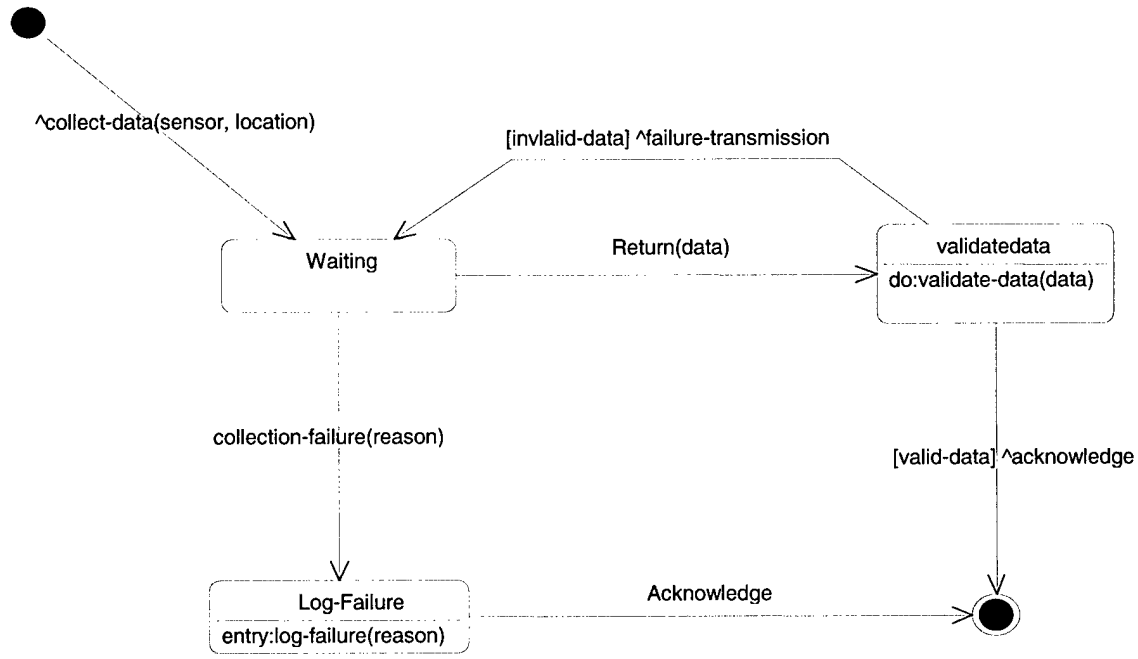

Figure 30: A Communication Class Diagram

actions from either the transition or the new state. Otherwise, the agent compares the message to all possible conversations that it may participate in with the agent that sent the message, and begins a new conversation if the message matches a transition from the start state. The syntax of a transition follows conventional UML notation as shown below and described by DeLoach (DeLoach 1999).

rec-mess(args1)[cond]/action^trans-mess(args2)

The above syntax means that if the message *rec-mess* is received with the arguments *args1* and the condition *cond* holds, then the method *action* is called and the message *trans-mess* is sent with arguments *args2*. Any missing element of the transition syntax shown above is replaced with a *true* identifier, meaning any combination of one or more of the elements consists

of a valid transition. For example, a transition with just a guard condition, *[cond]*, is allowed, or one with a received message and action, *rec-mess/action*. Figure 30 is recognizable as the initiator half of a conversation, since the transition from its start state is triggered by a sent message.

Any actions in a conversation must be mapped to methods in the agent classes. Actions can be attached to either transitions or states. On the other hand, some agent class methods may be already defined before a conversation is created that use them (see Section 4.6.4). Not all agent class methods need to be actions in a conversation, but having pre-defined methods gives the designer actions to choose from when constructing conversations. Work by Robinson (Robinson 2000) contains further details on agent class methods.

4.5.2 Constructing Conversations – Rationale

This phase of MaSE is where earlier work building Sequence Diagrams and tasks pays off for the designer. The main problem is in knowing what states and transitions to add to the Communication Class Diagrams. Sequence Diagrams and tasks were created to ensure that every message or event captured by them translates into conversation as a send transition and a receive transition on the two Communication Class Diagrams. The more Sequence Diagrams and tasks that the designer has available at this point, the more pre-defined parts of conversations are available.

4.5.3 Constructing Conversations – Utilizing Sequence Diagrams and Tasks

While the operation of a conversation is relatively simple, its design can be quite complicated. Conversations are defined at a high level. Specifically, the initiator and responder agent classes are specified for each conversation in the system. The problems encountered in this

phase deal with building the finite state automata (FSA) that define the operation and protocol of conversations.

The incorporation of Sequence Diagrams and tasks assists in the building of the conversation FSAs for the reasons listed above. Conversations must support and be consistent with all sequence diagrams derived earlier. They may also incorporate states from tasks. Some tasks, in fact, operate entirely over single conversations and can be designed directly. In general though, conversations are built by first adding all possible states and transitions that can be derived from the Sequence Diagrams and tasks. At this point much of the conversation often exists. For the rest of the conversation design, it is a matter of adding states and transitions as necessary to convey the required messages. This subject is addressed further in Section 6.2.4.

4.5.4 Constructing Conversations – Avoiding Deadlock

While constricting conversations, it is very helpful to verify them during design to avoid deadlocks. In general, a conversation is deadlocked when both sides are awaiting a message from the other side. There are also many other ways to improperly design a conversation. For example, every 'send' from one half of the conversation must have a corresponding 'receive' on the other half in order to avoid deadlock. Additionally, the conversation must be able to exit every state, meaning that every state must have a valid transition from it that eventually leads to the end state. The topic of deadlock and methods to avoid and detect it are covered in detail by Lacey (Lacey 2000).

4.5.5 Constructing Conversations – Balancing

Finally, the designer must balance between having many simple conversations or a few complex ones. If the system has a large number of simple communications, these should be passed by a series of smaller conversations. Larger and more complex conversations are only

appropriate if an elaborate protocol is required. In general, a conversation should support a single goal, and be as small as possible to support that goal.

4.5.6 Constructing Conversations –Example

One of the conversations created by the EGADS designer is shown in Figure 31 and Figure 32. It is the InitiateTasking conversation that occurs between the "Cmdr" and "TaskCtrl" agent classes, as shown in Figure 29. The conversation construction begins when the designer picks the first event out of the Sequence Diagram in Figure 27: "new_task". This event is passed between the "Commander Interface" and "Task Controller" roles, which are played by the agent classes mentioned above. Therefore, the "new_task" event must be a message in a conversation with the "Cmdr" agent class as the initiator and "TaskCtrl" as the responder. The EGADS designer decides that the commander would want some feedback that the tasking was accepted, and given an opportunity to resend a new tasking if it was not. This results in a fairly common conversation pattern, in which each half has two loops: one for a valid transmission (or data, action, trigger, etc...) and the other for an invalid one. The EGADS designer creates the two Communication Class Diagrams shown in Figure 31 and Figure 32.

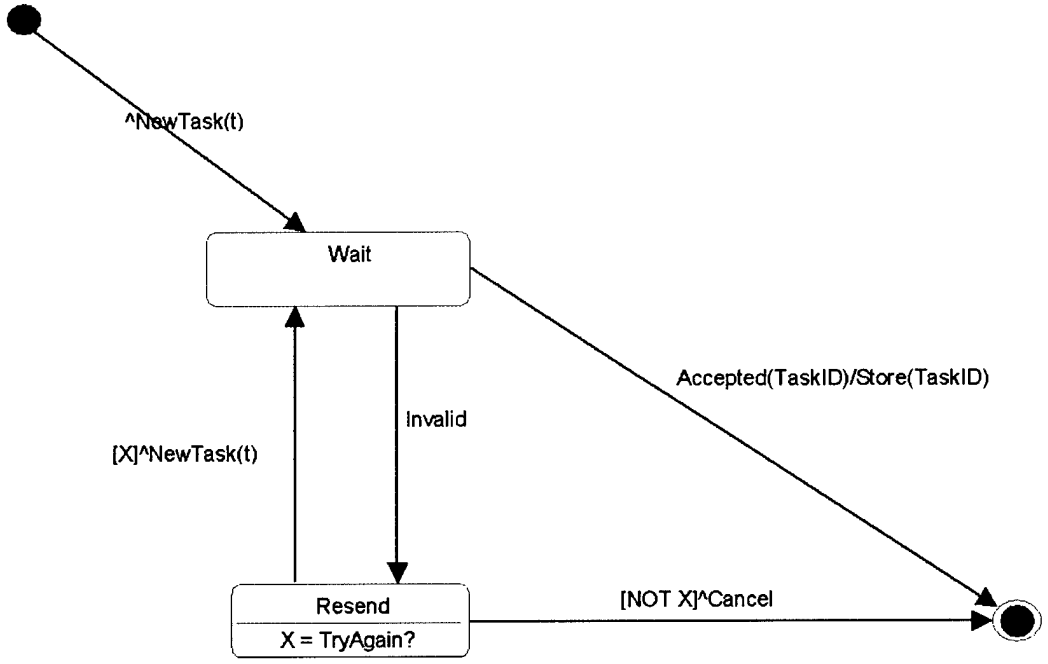

Figure 31: EGADS InitiateTasking Initiator

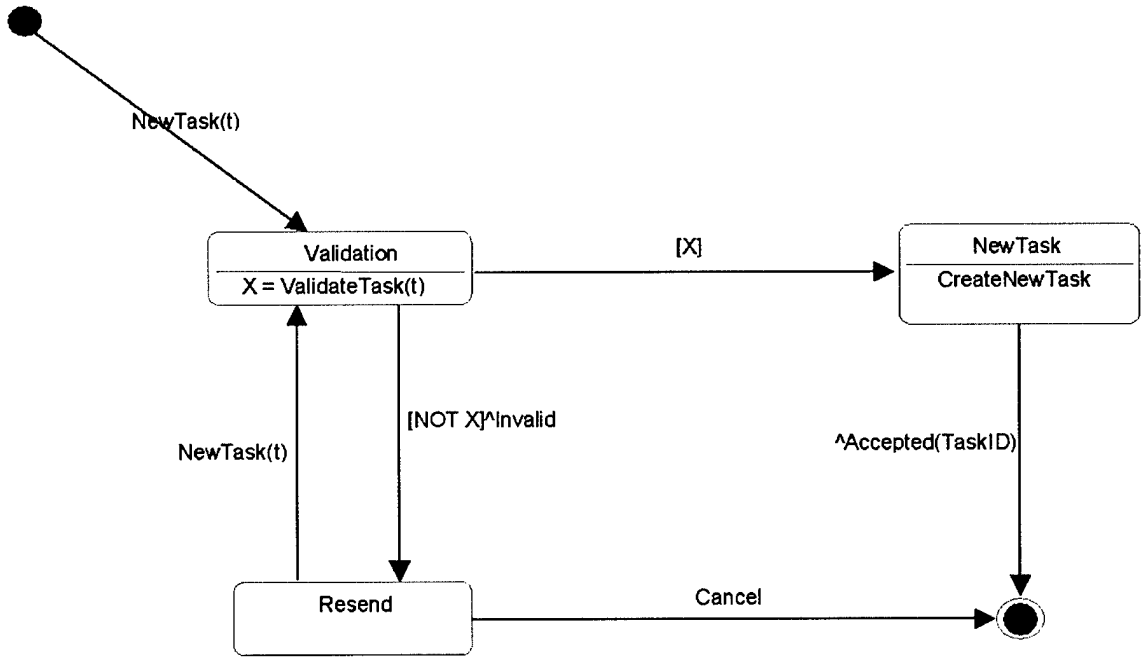

Figure 32: EGADS InitiateTasking Responder

4.6 Assembling Agents

In this phase of MaSE, the internals of agent classes are created. A parallel thesis by Robinson (Robinson 2000) describes the details of assembling agents from a component-based architecture. Therefore, this section only provides an overview of the process, and identifies links to the rest of the MaSE methodology. As such, there is no "Rationale" sub-section as there are in the other phases.

4.6.1 Assembling Agents – Definitions

The tools for describing individual agent classes are presented by Robinson (Robinson 2000). He provides five different architecture templates and describes the object oriented approach he used to create them. The architectures include Belief-Desire-Intention (BDI), reactive, planning, knowledge base, and user-defined agents. Each architecture template has a specific set of components. For example, a reactive architecture includes a Controller, MessageInterface, RuleContainer, and Effectors.

A designer can either define components from scratch or use pre-existing components. Furthermore, components may have sub-components that may in turn be architectures containing components. In theory the sub-components could continue indefinitely, though in practice they rarely are sub-defined past the top level of the architecture. Components are instantiated to produce actual code objects. Instantiation depends on the component, but requires either selection of pre-coded objects or the actual generation of code.

Components are joined with either inner- or outer-agent connectors. The inner-agent connectors connect with other components, and outer-agent connectors connect with external resources such as other agents, sensors and effectors, databases, and data stores. Furthermore,

components have associated state-diagrams that represent sequences of events passed from one component to another. An example is shown in Figure 33.

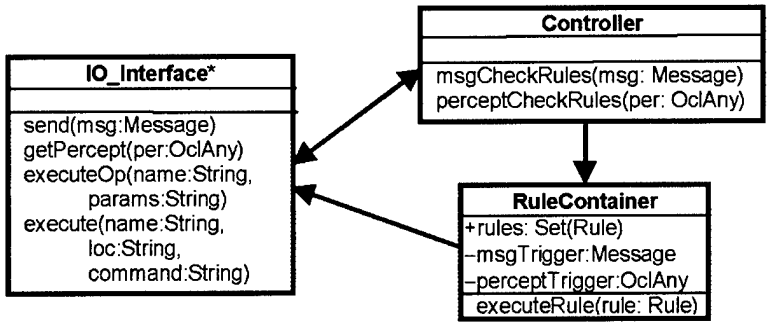

Figure 33: Generic Reactive Agent Class Architecture

4.6.2 Assembling Agents – Details

There are three methods of assembling components to define an agent class. The first is to retrieve one of the four pre-defined architectures (BDI, reactive, planning, knowledge base). The components are then either instantiated as is, or modified by addition, deletion, or modification of attributes and methods. Additionally, whole components and sub-components can be added or deleted. Finally, any new components must be connected to existing ones The second method is to retrieve pre-defined components, and assemble them into a user-defined architecture. This method also allows for modification of components, attributes, and methods as described above.

The third method is to define both the components and architecture from scratch. When defining architectures from scratch, it is difficult to decide what components to utilize or avoid. By this point the designer has much information available to assist in this determination, such as the goals, roles, tasks, and conversations associated with the agent class. The only absolute requirement is that all actions in a conversation be associated with methods in an agent. How, for

example, does a designer decide when to include a planner component? A planner is responsible for delivering a sequence of steps to achieve a goal, so the agent class must have a goal it is trying to achieve. Second, the agent class must have access to the state of the environment. Third, the agent class must have a means of affecting the environment through a set of operators. This set of operators already exists in most MaSE agent classes. Such operators include starting a conversation with another agent, running an agent method, passing control to another component within the agent architecture, or interfacing with an external component along an outer-agent connection. Therefore, any agent that is required to perform a sequence of steps in pursuit of a goal and has a means of detecting and affecting the environment in pursuit of that goal is a candidate for utilizing a planner component.

4.6.3 Assembling Agents – Example

One of the agent classes assembled by the EGADS designer is the Registrar class. Using the reasoning described by Robinson (Robinson 2000), the designer specializes the reactive architecture from Figure 33 to build the components of the Registrar Class shown in Figure 34. The components shown are a way to build the Registrar agent class so that it can partake in MaSE conversations with other classes. This Registrar class does not need any further components because all of its actions can be modeled by the reactive architecture. In other words, all of the actions taken by the Registrar are in reaction to events sent to it by other agent classes.

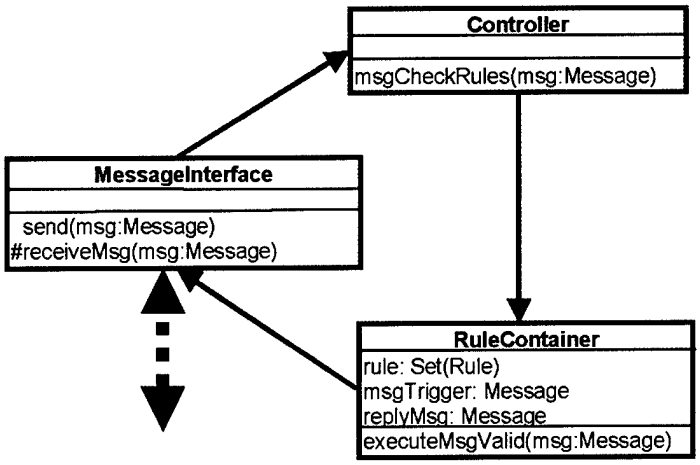

Figure 34: EGADS Registrar Class Components

4.6.4 Constructing Conversations versus Agent Assembly

This is not a phase of MaSE, but a comparison between two phases. As depicted back in Figure 17 and alluded to in the appropriate sections, constructing conversations and agent assembly are closely related activities. In practice, it is useful to go back and forth between these while staying within one functional area of the design. For example, the EGADS designer chooses to fully detail the "registration" conversation, and assemble the registrar agent before moving on to other conversations.

The question of which to do first is answered best by determining the style of conversations that the system uses. In particular, is the system communication-heavy? Are the communications relatively complex? The designer should build conversations first if the system consists of many simple conversations, or if the initial context of the system includes many use cases. Agents are better built first if there are a small number of complex conversations, or if many agents are already identified or reused.

4.7 System Deployment

The final phase of the MaSE methodology takes the agent classes and instantiates them as actual agents. It uses a deployment diagram to show the numbers, types, and locations of agents within a system. System deployment is actually the simplest phase of MaSE, as most of the work was done in previous steps. The idea of instantiating agents from agent classes is the same as instantiating objects from object classes in object-oriented programming.

4.7.1 System Deployment – Definitions

Deployment Diagrams are used to define a system based on agent classes defined in the previous phases of MaSE. Deployment Diagrams define system parameters such as the actual number, types, and locations of the agents within the system. Figure 35 shows an example Deployment Diagram for the EGADS system. The three dimensional boxes are agents and the connecting lines represent conversations between agents. The agents are named either after their agent class or in the form of *designator:class* if there are multiple instances of a class. Any conversation between agent classes appears between agents of those classes. Furthermore, a dashed-line box indicates that agents are housed on the same physical platform.

4.7.2 System Deployment – Rationale

A system must be arranged in a Deployment Diagram before it can be implemented in code. This is due to the differences between agents and agent classes. An agent requires such information as a hostname and address in order to participate in any communications external to the system that it resides on.

A Deployment Diagram also offers another opportunity for the designer to tune the system. Agents can be arranged among various machine configurations in order to better use available processing power or network bandwidth.

4.7.3 System Deployment - Details

In some cases the system requirements may have specified a particular number of components, or particular machines that they reside on. Otherwise the designer should consider message traffic when putting agents on particular machines. Obviously, inter-agent communication speed will depend on the network they communicate over. In many cases, agents can be deployed on the same machine. However, putting too many agents on a single machine destroys the advantages of distribution gained by using the agent paradigm.

Another consideration is the processing power available on particular machines and required by particular agents. If an agent has a high CPU requirement, it can be placed on a machine by itself. On the other hand, a machine that has low CPU availability due to other processes or older technology should not be required to handle a large number of agents. A strength of MaSE is that these modifications can be made after designing and generating a variety of system configurations, along with the gathering of performance data.

A final element to consider is automatic code generation. The MaSE methodology, and agentTool system are primarily concerned with actually engineering agent systems. As such, all of the steps of the methodology work toward that end. Code generation will be a largely automatic process from Deployment Diagrams. Code generation is not a piece of MaSE at this time, but is assumed to happen just after this phase.

4.7.4 System Deployment - Example

Completing the EGADS example, the designer creates the deployment diagram in Figure 35. It shows a system containing two different types of intelligence sources, unmanned aerial vehicles (UAVs) and satellites (SATs), with multiple sources for each type. Each type of source has a Mission Controller from the agent class "MissionCtrl". All of the agent classes of the data sources, mission controllers, and the task controller participate in conversations with the registrar agent, hence the numerous conversation lines connecting them. Finally, the single instances of the Registrar and Task Controller agents are placed on a single machine, as indicated by the dashed-line box, in order to conserve network bandwidth. Alternately, the EGADS designer could have also put the mission controllers on that same machine as well, but the network between the command center and the remote sites that handle the data sources is very low-bandwidth, so the expected high volume of conversations between mission controllers and data sources would be over that network. In the configuration shown, the messages passed over that lower-quality network are the low-volume requests from the task controller and mission controllers, plus the one-time registration conversations as new data sources are added.

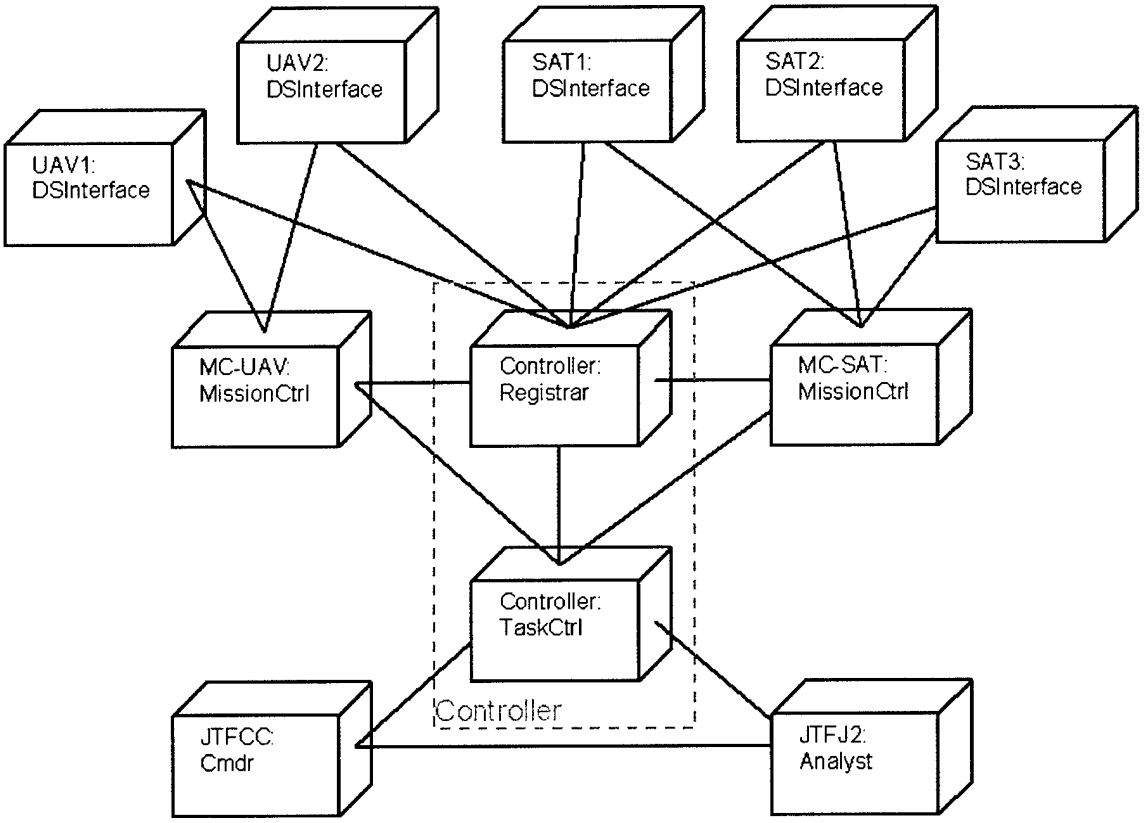

Figure 35: EGADS Deployment Diagram

4.8 Summary

The Multiagent Systems Engineering methodology is a seven-phase process that is designed around transformations from one abstraction to the next in a series, as shown in Figure 36. It begins by capturing the essence of an initial system context in a structured set of goals and use cases. Next, the goals are combined to form roles, which include tasks that describe how their associated goals are satisfied. The use cases are then transformed into Sequence Diagrams so desired event sequences will be designed into the system. After that, the roles are integrated into agent classes connected with conversations. The details of the conversations and agent classes are then worked out, which can happen in either order depending on the system. Finally,

the agent classes are deployed as agents in a Deployment Diagram from which the system can be directly implemented.

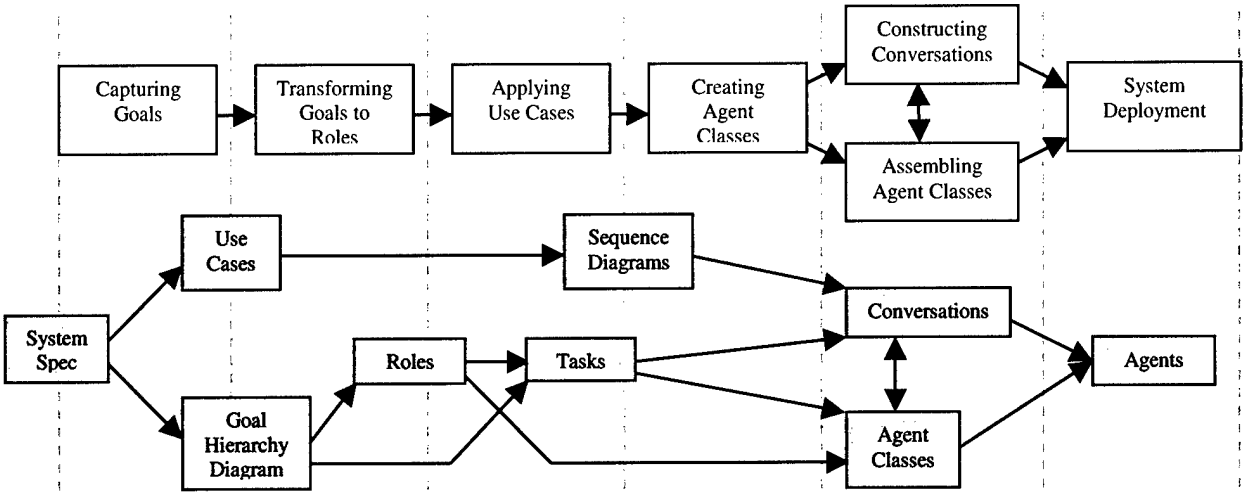

Figure 36: MaSE Methodology

V. Results

This chapter describes the creation of the agentTool system based on the MaSE methodology. It shows how the elements of the MaSE methodology are implemented in the application.

The implementation of agentTool was done in JAVA 1.2. It was a joint project between all current students and research assistants in AFIT's Agent Research Group: Tim Lacey, Marc Raphael, Mark Wood, David Robinson, and Jennifer Mifflin. Each was responsible for the portion of the implementation dealing with their thesis work.

The agentTool system is based on the MaSE methodology presented in Chapter 4. Currently agentTool implements three of the seven phases of MaSE, as indicated by the outlined region in Figure 37. The planned future of agentTool includes expanding it to cover more of the MaSE methodology. The goal of agentTool is to allow multiagent system designers to formally specify the required structure and behavior of a multiagent system and semi-automatically synthesize multiagent systems that meet those requirements.

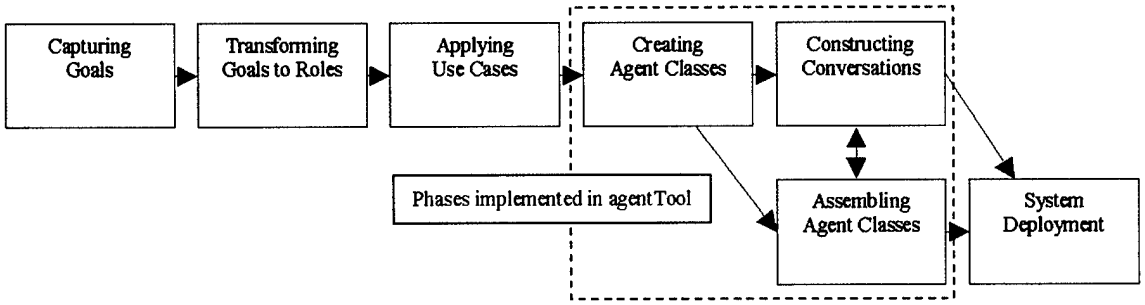

Figure 37: MaSE in agentTool

5.1 Objectives of agentTool

The long-term objective of agentTool research is to be a platform for the investigation and development of automated multiagent technology. Among other areas, this research will show that by using automated software synthesis techniques, systems of multiple intelligent agents can be developed that implement security and communication protocols in a provably correct manner. This research will also provide a mechanism to abstract the precise security and communication protocols so that the agent developer will not have to worry about them when specifying agent behavior. Also, assuming multiple security and communication protocols exist, an agent whose behavior is specified in such a system would be able to be generated using various combinations of security and communication requirements.

5.2 Operation of agentTool

The user interface to agentTool is shown in Figure 38. The menus across the top access several system store and retrieval functions, including access to a persistent knowledge-base developed by Marc Raphael (Raphael 2000). The buttons down the left side are for adding components to the different design panels, and the text window below them displays system messages.

The main window of agentTool is where all of the design "documents" are created. They are accessed through a series of tabbed panels across the top of the main area. The different panels have individual sets of rules for the objects that can be placed on them and for the syntax of text that can be placed on the objects. Objects are added to the panel and can be moved around the viewing space for clarity. Selecting an object enables access to the appropriate sub-panels. Figure 39 shows an example where two agents have been added, with a conversation between them.

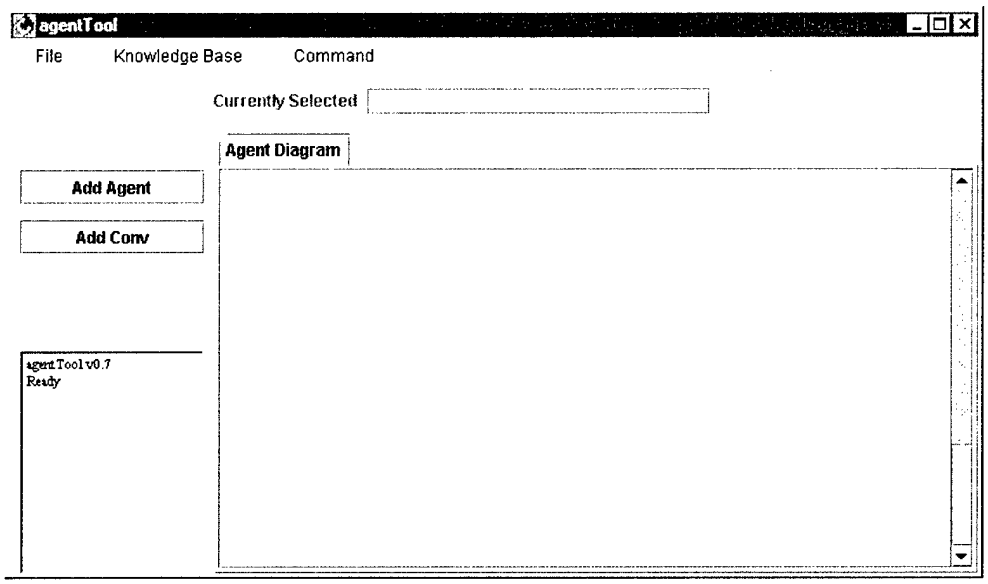

Figure 38: agentTool User Interface

The agentTool system enables a designer to depict high-level system behavior graphically using the MaSE methodology. The Agent Diagram Panel defines the types of agents in the system as well as the communications that take place between them. This system-level specification is then refined in sub-panels for each type of agent in the system. To refine an agent, the designer either selects or creates an agent architecture and then provides a detailed behavioral specification for each component in the agent architecture, as described in Section 4.6. The conversations between agents are similarly refined on sub-panels that depict their two state machine components, the initiator and responder, by enforcing MaSE syntax and using rules described in Section 4.5.

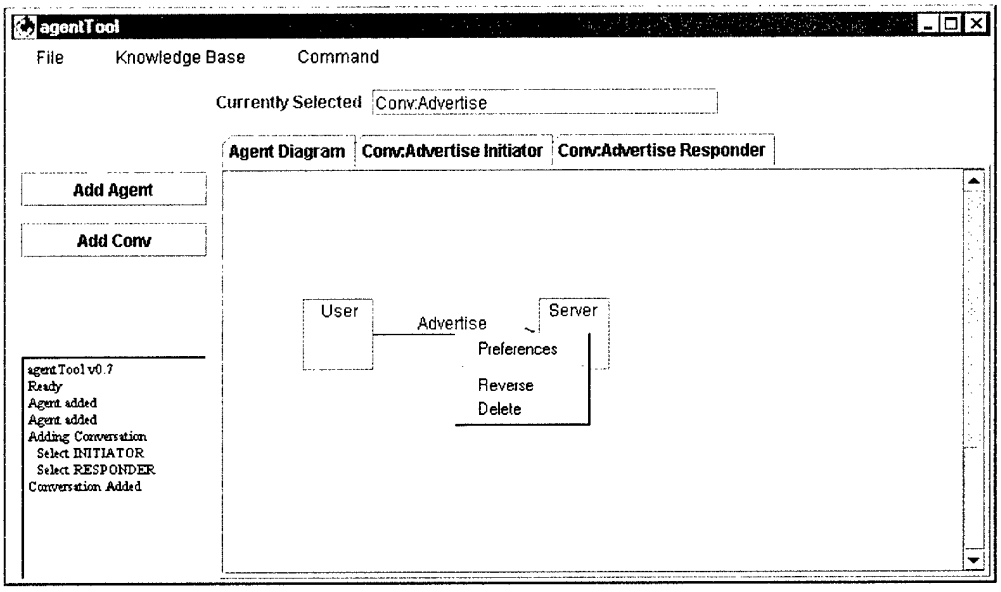

Figure 39: agentTool Popup Menu

The aspect of agentTool that is perhaps the most appealing is the ability to work in various pieces of the system and at various levels of abstraction interchangeably. This mirrors the ability MaSE provides to incrementally add detail during design. The "tabbed pane" operation of agentTool is what brings out this ability of MaSE, since it is always clear what level of the design hierarchy is currently being accessed, and what has to be done (clicked on) to access a different level.

It is easier to envision the potential of this ability by considering an implementation of the entire MaSE methodology into agentTool. During all phases of system development, all of the various analysis and design abstractions are available through the tabbed panes atop the main viewing area. The ordering of the tabs mimics the ordering of the methodology phases in MaSE, so selecting a tab to the left of the current pane would move "back" in the methodology and a tab to the right would move "forward". With an understanding of MaSE, clicking on a particular tab

will display a pane containing the expected diagram or model pertaining to that phase. Additionally, selecting a graphical object on any pane will (at the user's option) highlight "influenced" objects on all other panes. The influence includes any objects that can be reached through transformations in both directions through the methodology.

5.3 Building a Multiagent System using agentTool

Constructing a multiagent system using agentTool begins at the main system panel, called the Agent Diagram Panel, shown above in both Figure 38 and Figure 39. This panel is identical in form and function to the Agent Class Diagram of MaSE (see Section 4.4). The agent classes and conversations that are added to this screen are identical to the agent classes and conversations in MaSE.

It is assumed that before using agentTool, the system designer has proceeded through the first three phases of MaSE and created the agent roles, tasks, and Sequence Diagrams from which the Agent Class Diagram is constructed. At this point, work is transferred onto agentTool. This transference may also happen a piece at a time, as the designer decides what roles should combine to make what agents. In any case, agentTool work begins at the Agent Diagram Panel.

The following subsections illustrate the use of agentTool by tracking a system through the design process. The MaSE diagrams for this system can be found in Appendix B.

5.3.1 Adding Agent Classes and Conversations

In agentTool, a conversation cannot be added without agent classes to serve as its "anchors." A conversation can be between two agents of the same class, however. Therefore, it is possible to add all agent classes to the Agent Diagram Panel before any conversations are added. It is also possible to add "sections" of the system at a time, connecting appropriate agent

classes with conversations, then moving on to the next section. Either method is fully supported, and truly a matter of personal choice by the designer. Figure 40 shows the completed Agent Diagram Panel for a course-scheduling application that is used as an example system in this chapter. The other MaSE diagrams leading up to this point in the development lifecycle are included in Appendix B.

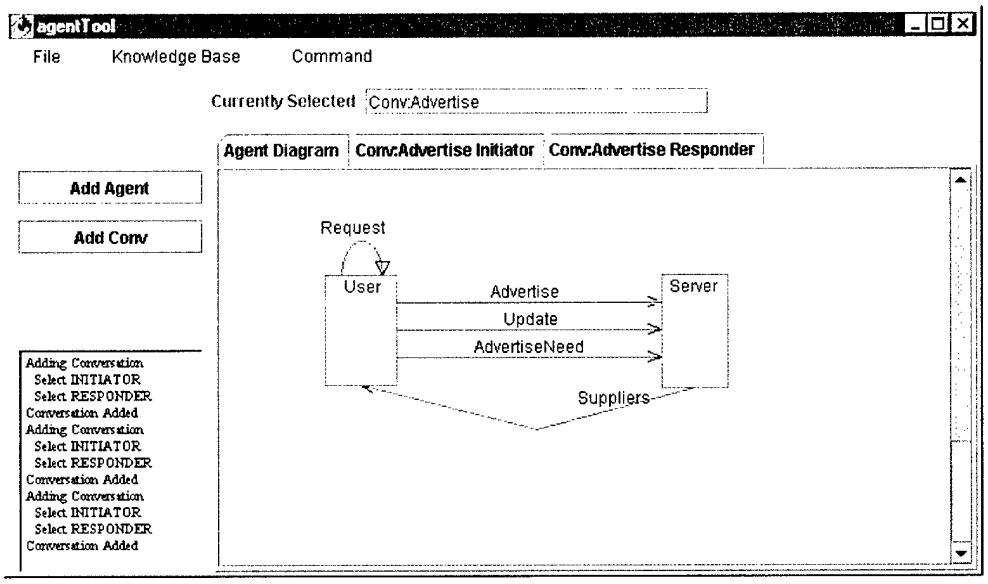

Figure 40: Agent Diagram Panel

5.3.2 Constructing Conversations in agentTool

The initiator and responder sub-panels of a conversation are identical to Communication Class Diagrams in MaSE (Section 4.5). States and transitions are added in the same manner that agent classes and conversations were in the Agent Diagram Panel. Selection of the "Add State" button adds a state to the panel, and the "Add Conversation" button adds a conversation between the two selected states. The attributes of states and transitions can be changed by right-clicking on the appropriate object, as shown in Figure 41, and selecting Properties. This displays a dialog box, such as the one in Figure 42.

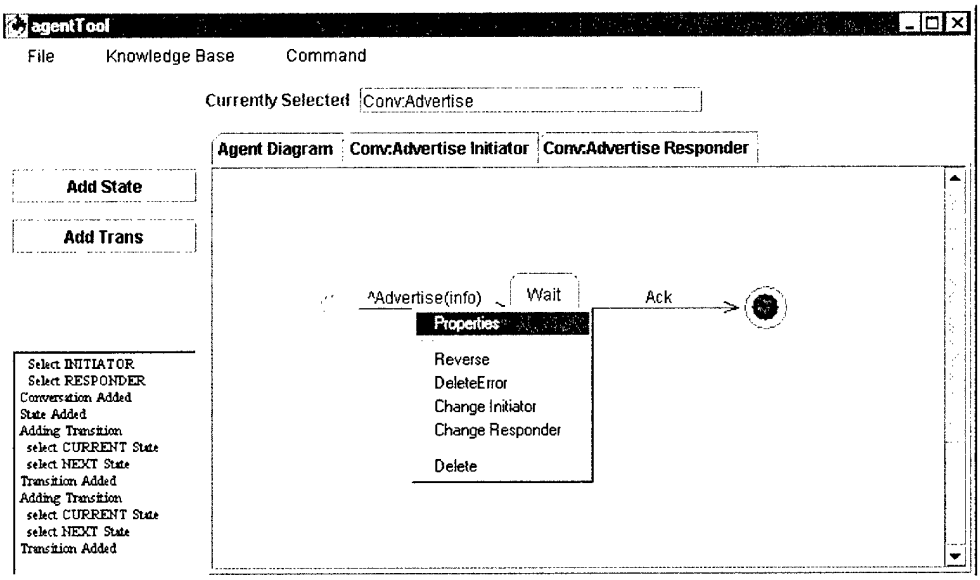

Figure 41: agentTool Conversation Panel

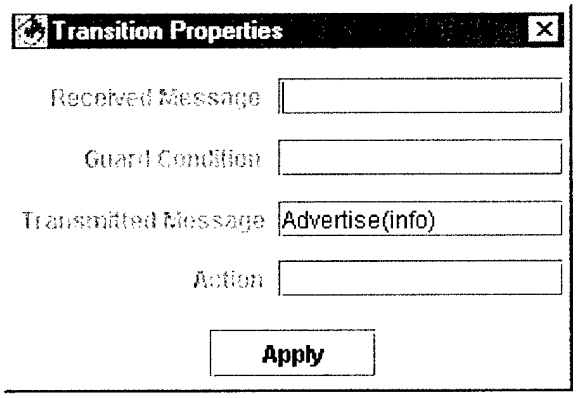

Figure 42: agentTool Conversation Properties Dialog

Any values assigned to states or transitions show up in text on the panel. These values can be changed at any time by again pulling up the Properties dialog box. The start and stop states cannot have any actions attached to them.

A conversation can be verified at any point in its creation by using the Verify Conversations command from the Command menu. This activates the verification application

created by Lacey (Lacey 2000). If any errors exist, the verification results in a highlighted piece or pieces of a conversation, as shown in Figure 43 on the "Ack" transition (highlights are yellow in the application). Each of the highlights is a piece of the conversation that is a potential source of an error that was detected by the verification routine.

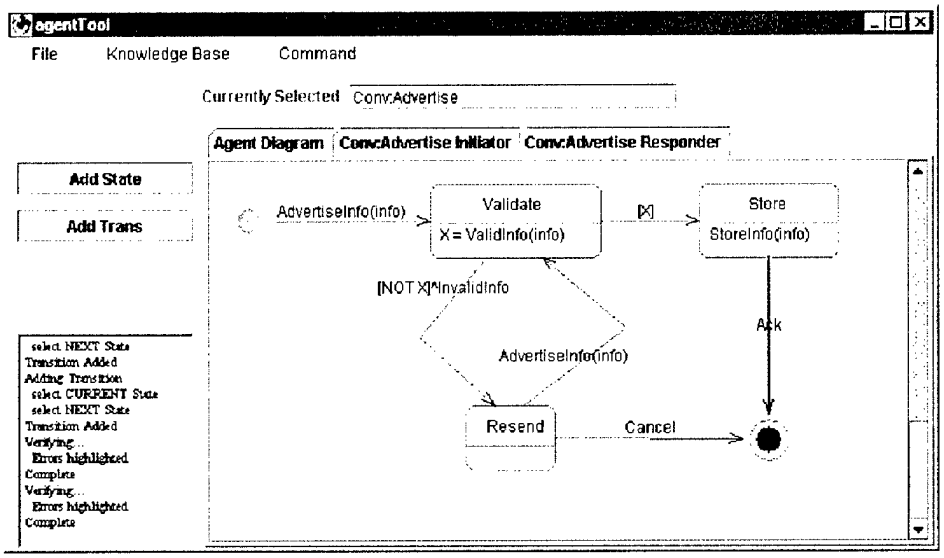

Figure 43: agentTool Conversation Error

5.3.3 Assembling Agent Class Components in agentTool

Robinson provides an excellent description of the ability of agentTool to define the internal components of an agent class (Robinson 2000). This subsection overviews the process.

Agent classes in agentTool have the same internal components described in Section 4.6. They can be added, removed, and manipulated in a manner similar to the other panels of agentTool. Agent classes do have an added layer of complexity, however, since all of their

components can have state diagrams associated with them and additional sub-components beneath them.

The agent class components shown in Figure 44 are the details of the "User" agent class from Figure 40. The pattern depicted is an instantiation of the reactive agent architecture shown earlier in Figure 33, specialized to participate in conversations.

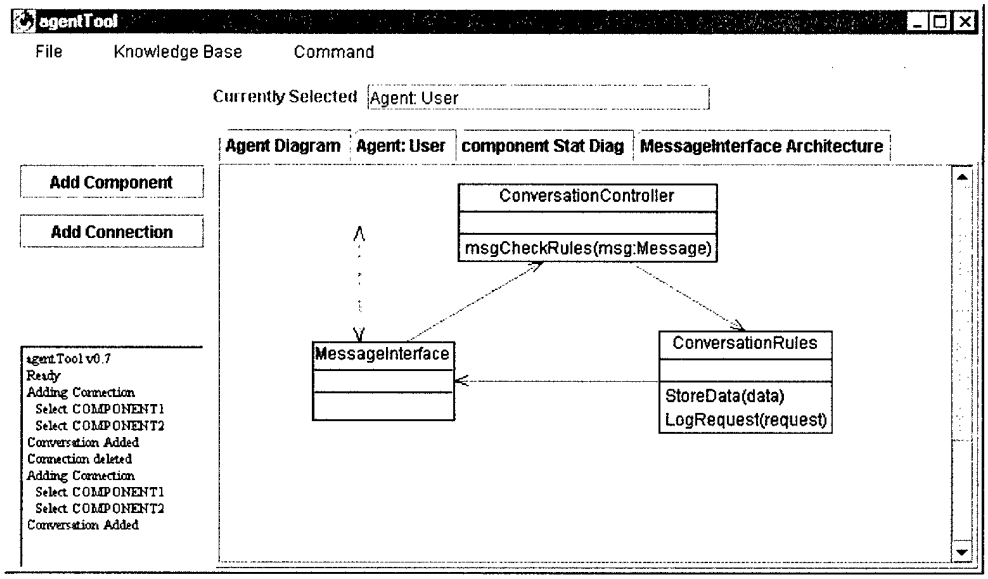

Figure 44: agentTool Agent Class Components

Details can also be added to lower levels of abstraction. In Figure 44 the "component Stat Diag" and "MessageInterface Architecture" tabs lead respectively to a component state diagram and sub-component panel of the MessageInterface component. The state diagram panel details how the component progresses through operational states, in much the same fashion as a conversation. The sub-component panel contains additional components and connections, exactly like the main agent class panel.

5.4 Underlying Formalisms of agentTool

The formal semantics of MaSE are reflected in the transformations from one abstraction to the next. For example, agents contain a set of roles that in turn contain a set of goals, and a conversation must have exactly two participants, though they can both be the same class. These semantics are both incorporated and supported by agentTool. For example, an agent class "object" in agentTool contains a set of conversations. In a future version of agentTool that incorporates the entire MaSE methodology, a role "object" could be mapped "backward" to the set of goals from which it was created, or "forward" to the agent class that plays it.

The agentTool system is based on an object hierarchy that mimics the objects in MaSE. The highest-level object is a "system", called an ATsystem in agentTool. Currently, an ATsystem contains sets (Vectors) of ATagent and ATconversation objects, just as a system in MaSE consists of agent classes and conversations. These objects contain further subcomponents reflecting the models in MaSE, such as ATagentComponents (sub-components of agent classes) and ATstatetables (half of a conversation).

5.5 Summary

The agentTool system demonstrates the ability of an automated tool to assist in multiagent system design. It also validates the MaSE methodology as a method suitable for automation, which is one of the goals of this thesis. Work on agentTool will continue in the future, including automated code synthesis and possible incorporation of the rest of MaSE into the application.

VI. Conclusions and Recommendations

6.1 Conclusions

This section describes conclusions reached from this thesis. Some things worked as planned and some did not. In general, the MaSE methodology turned out to have a larger scope and correspondingly less depth than anticipated.

6.1.1 Contributions

The main contribution of this thesis is the ability of MaSE to assist a multiagent system designer through the entire software development lifecycle, beginning from a textual system representation and proceeding in a structured manner toward working code. This full lifecycle coverage of MaSE turned out better than initially expected. At the commencement of research, there was no intention to address analysis issues such as capturing system requirements and use cases.

MaSE combines several preexisting models into a single structured methodology. Most of the models used within the methodology have therefore been already justified and validated within the realm of agents and multiagent systems. A sequence of guided transformations connects the elements of this strong foundation together into a clear high-level picture of how a designer should go about creating a multiagent system.

Since the connections between models are transformations, they can also be tracked in reverse order. In this manner, potential changes to the system design can be tested for effects on earlier models. For example, if it becomes clear during system deployment that a particular CPU cannot handle the load imposed by several agents that must reside upon it, the roles that make up

those agents could be identified for possible reassignment. In an automated tool, this ability would be even more powerful. A future vision of agentTool includes the ability to select an object at any level of the design, and receive visual feedback (highlighting, for example) indicating which objects it influences either "upstream" or "downstream" in the methodology.

Finally, the construction of the abstractions in MaSE from modular components (agent classes from roles in particular) supports reuse nicely. Different parts of a system, such as collection of roles used in registration, can be collected and stored for later use. This process also works in reverse. If a new system requires the ability to register new pieces such as data sources, the knowledge-base of existing components can be referenced and reused. Raphael is addressing this and related issues in his thesis on knowledge-bases to support multiagent system design (Raphael 2000).

6.1.2 Deficiencies

Several phases of MaSE describe transitions from one abstract construct to another. In each case, the transformation is accompanied by rules and guidelines for the designer to consider when performing the transformation. Nearly every transformation can be enhanced by the addition of a step-by-step procedure – a methodology – that details how the transformation must take place.

In particular, the construction of conversations was intended to be a process that produced the finite state machine for each half of the conversation. The addition of Sequence Diagrams and tasks to the methodology was specifically intended to facilitate that process, which was not fully produced in the end. This concern is repeated in Section 6.2.4 as an area of future research. Additionally, the transformations of goals to roles and roles to agent classes would benefit from a similar process that was focused more on a series of rules than on guidelines.

6.2 Future Research Areas

This section lists several topics that are partially addressed in this thesis. Each topic would clearly benefit from further investigation and, hopefully, the MaSE methodology will be stronger for it. Successive sub-sections describe future work on tasks, use cases, role models, conversations, and automatic code generation.

6.2.1 Tasks

The use of tasks in MaSE, as introduced in Section 3.3.6, is an effective aid to constructing conversations. Tasks incorporate information into the design process about actions taken in support of specific goals. The state-machine representation maps well into conversations since it shares a similar form. Tasks have not been significantly researched yet, however. For example, further investigation of the inter-role communications in tasks certainly seems called for. It seems reasonable that tasks could potentially communicate among several roles. Does each role involved in a task need a complementary version of that task, similar to the two halves of a conversation? What are the ramifications of having state variables that span a task? What information can safely be passed by tasks?

There are many possible questions relating to the syntax and semantics of the graphical language that describes tasks. How could it be made clearer? In a collection of related tasks, can inter-task communications be depicted in a similar manner to an Agent Class Diagram? These and other questions are certainly relevant, and will not negatively affect the MaSE methodology as long as the task characteristics described in preceding sections are maintained.

6.2.2 Use Cases

Failure and negative use cases are briefly described in Section 4.1.3.2. The concept of a use case utilized in this thesis is that of a *positive* use case. A failure use case is a description of an error in the system, and a negative use case is a sequence that is desired *not* to occur within the system. It is not clear how the inclusion of these concepts would affect MaSE, or even if they would fit in at all. Could they be translated into a Sequence Diagram? Would such a translation be useful?

6.2.3 Role Model Indexing

Role models are first introduced in Section 2.3.1. Kendall has done much research into role models and related ideas (Kendall 1998). From the perspective of MaSE, the most useful property of role models is that they are reusable patterns that can be stored in a catalog and retrieved for application on a particular system design. As discussed in Section 3.3.2, the reason they were not featured in MaSE is that it is clear neither how to index such a catalog nor how to "recognize" a particular pattern in a system design.

6.2.4 Conversations – State table construction

MaSE describes several ways to assist in the construction of conversation. Sequence Diagrams and tasks provide a set of messages that must be passed by a conversation, but it is far from a complete set. The most difficult part of conversation construction is not covered; that is actually constructing the state machines. A basic set of rules or guidelines dealing with assembling the pieces of a conversation, given the existing pieces of MaSE, would fit well into the methodology.

6.2.5 Automatic Code Generation

MaSE was created expressly for generating agent systems automatically in code. The methodology assists in creating all the necessary design objects and guides the system designer right up to the point where automatic code generation would occur. Code generation is beyond the scope of this thesis, but not outside the scope of this methodology. There are other theses at AFIT outside of the Agent Research Group that are addressing this issue (Ashby 2000). They are not dealing directly with agent systems, however.

6.2.6 Bridging Agent Classes and Conversations to Components

MaSE describes how to create the connections between agent classes and the roles that they play in a multiagent system. Also, a phase of the methodology is devoted to designing the internal components of an agent by using the agent component architecture created by Robinson (Robinson 2000). However, Robinson's work was not devised to work within MaSE. It is a general-case model that is intended to cover all types of agents. The specific problem is to map agent classes and conversations created using MaSE to the components of Robinson's model. For example, a MaSE conversation would require specialization of several inter-agent components. An IO_Interface could be instantiated as a Message_Interface, and the two finite state automata transformed into a transition table encoded inside a Controller component. In any case, more research is appropriate to bridge the gap between MaSE and the component-based architecture.

VII. BIBLIOGRAPHY

Ashby, Michael R. Tool-Based Integration and Code Generation of Object Models. MS thesis, AFIT/GE/ENG/00M-02, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2000

Brazier, F., Jonker C. and Treur, J. "Principles of Compositional Multi-Agent System Development" Proceedings of the IFIP'98 Conference IT&KNOWS'98, Chapman and Hall 1998

DeLoach, Scott A. "Multiagent Systems Engineering: a Methodology and Language for Designing Agent Systems" Proceedings of Agent Oriented Information Systems '99 (AOIS'99), pp. 45-57. Seattle WA, 1 May 1999.

Drogoul, A. and Collinot A. "Applying an Agent Oriented Methodology to the Design of Artificial Organizations: A Case Study in Robotic Soccer" Autonomous Agents and Multi-Agent Systems, 1(1), 113-129, 1998

Iglesias, C., Garijo, M. and Gonzalez, J. "A Survey of Agent-Oriented Methodologies" Intelligent Agents V – Proceedings of the 5th Intl. Workshop on Agent Theories, Architectures, and Languages (ATAT-98)

Jennings, N. R., Sycara, K. and Wooldridge, M. "A Roadmap of Agent Research and Development" Autonomous Agents and Multi-Agent Systems, 1(1), 7-38, 1998

Kelley, Jay W. Air Force 2025. 2025 Support Office, Air University, Air Education and Training Command. Air University Press, August 1996.

Kendall, Elizabeth A. "Agent Roles and Role Models: New Abstractions for Intelligent Agent System Analysis and Design", (1998)

Kendall, Elizabeth A. and Zhao, L. "Capturing and Structuring Goals: Analysis Patterns", 1998

Kinny, D., Georgeff, M. and Rao, A. "A Methodology and Modelling Technique for Systems of BDI Agents" Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW'96 (LNAI Volume 1038) p.56-71 (1996)

Lacey, Timothy H. A Formal Methodology and Technique for Verifying Conversations in a Closed Multi-agent System. MS thesis, AFIT/GCS/ENG/00M-12. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2000

Nwana, H. S. "Software Agents: An Overview", Knowledge Engineering Review. 11(3): 205-244. (1996)

Pressman, Roger S. Software Engineering: A Practitioners Approach, 3rd ed. McGraw-Hill Inc., New York, 1992

Raphael, Marc J. Knowledge Base Support For Design and Synthesis of Multi-agent Systems. MS thesis, AFIT/GCS/ENG/00M-21. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2000

Reticular Systems, Inc. 1999. AgentBuilder – An integrated Toolkit for Constructing Intelligent Software Agents. Version 1.3, Copyright 1999 Reticular Systems, Inc.

Robinson, David J. A Component-based Approach to Agent Specification. MS thesis, AFIT/GCS/ENG/00M-22. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2000

Rumbaugh, J. Object-Oriented Modeling and Design, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1991

Shalikashvili, John M. Joint Vision 2010. Joint Staff: Pentagon, 1999.

Sycara, K. P. "Multiagent Systems". Ai Magazine 19(2): 79-92 (1998)

Wooldridge, M., and Jennings, N. "Intelligent Agents: Theory and Practice". Knowledge Engineering Review, 10(2): 115-152 (1995)

Wooldridge, M., Jennings, N., and Kinny, D. "A Methodology for Agent-Oriented Analysis and Design" (1999)

Appendix A - ELINT Gathering and Decision System (EGADS)

EGADS links a Commander to their intelligence gathering assets in the field. A group of dissimilar data collectors must communicate classified intelligence data to the commander, through an intelligence processing section. The commander can issue specific tasking for desired data, request status reports, and receive processed intelligence reports from the processing section.

Particulars:

- The system must link to many different kinds of intelligence assets including UAVs, JSTARS, land and sea-based radar, satellites, and preprocessed GCCS and JWICS data.
- The commander's taskings may include requests about a particular area, resource (air, land, and sea units), or combination. They may be one-time requests, or define a time window within which all movements must be reported. If the window ends after the request is made, then updates are sent until the window closes.
- A commander may request a status report of any open intelligence tasking. The report will return completed portions of the tasking, and status of uncompleted portions including an estimated completion time (or unknown).
- A commander may set preferences to determine how all reports (status and intelligence) will be presented.
- Before presentation to the commander, all intelligence reports must be sent through the intelligence processing section.
- Data sources must be able to be added or removed from the system as they become available or obsolete.

Example:

A commander desires to know what sort of air defenses will exist in a target area of an air strike that must be executed within 72 hours. At stake is when will the strike be launched, and what SEAD (Suppression of Enemy Air Defense) assets must be included in the sortie package.

Appendix B – Example of MaSE System Construction

This Appendix contains information and figures that depict a course-scheduling system run completely through the MaSE methodology for verification and validation purposes. It is also used in Section 5.3 for demonstrating the agentTool system.

Agent-Based Collaboration – User Requirements

Basically, we want to demonstrate that a distributed agent-based system can be used to help users share information in a collaborative environment. For our demo, we will assume that

1. Each user has various types of information that it owns or generates.
2. Each user needs to use other types of information that they do not own or generate, but that are owned or generated by other collaborators.

To facilitate this collaboration, the underlying agent system should

1. Allow the users to advertise the types of information they own or generate.
2. Allow the users to advertise their need for other types of information.
3. Allow users to advertise that their information has been updated.
4. Match suppliers of an information type to users either in a one-time fashion or on a continuing basis.
5. Allow users to request the actual information directly from suppliers of the information.

A user can have more than one piece of information of a certain type. If this is the case, then all the information of that type should be sent when requested.

Therefore, the agent-based system should support the following scenarios:

1. If a user requires information a certain type, the user should be able to advertise their need and receive a list of possible suppliers of that information. The user can then choose one (or all) of the suppliers to ask for the information. When the user asks, the supplier returns all of its current information on that type.

2. When a user acquires or creates new information types, the user should be able to advertise its ability to provide the information. Once a user has advertised its capabilities, it must provide its current information to anyone who requests it until is un-advertises its capabilities.

3. If a user has a long-term requirement for a piece of information (either because no one currently can provide the information or it wants to be kept up to date), it can advertise its need for this information. Then, if anyone subsequently advertises the capability to provide such information, the name of the supplier should automatically be sent to the user until the user un-advertises its need for the information.

4. If a user already has advertised its ability to provide a certain type of information, and it generates a new piece of that information, it should be able to re-advertise its capability. This re-advertisement should be sent to anyone with a long-term requirement for the information and interpreted as an information update.

These users may be distributed but need not know who or where the other users/suppliers are.

Agent-Based Collaboration – Goals

1.0 Help users share info collaboratively
- 1.1 Allow users to advertise information
  - 1.1.1 Allow users to advertise
- 1.2 Allow users to advertise their need for information
  - 1.2.1 Allow users to advertise
- 1.3 Allow users to advertise that their info has been updated
  - 1.3.1 Allow users to advertise
  - 1.3.2 Allow users to update their info
- 1.4 Match suppliers of an information type to users either in a one-time fashion or on a continuing basis
  - 1.4.1 Match suppliers to users on a one-time basis
    - 1.4.1.1 Match suppliers to users
  - 1.4.2 Match suppliers to users on a continuing basis
    - 1.4.2.1 Match suppliers to users
- 1.5 Allow users to request the actual information directly from the suppliers of the information
  - 1.5.1 Allow users to request

Agent-Based Collaboration – Goal Hierarchy Diagram

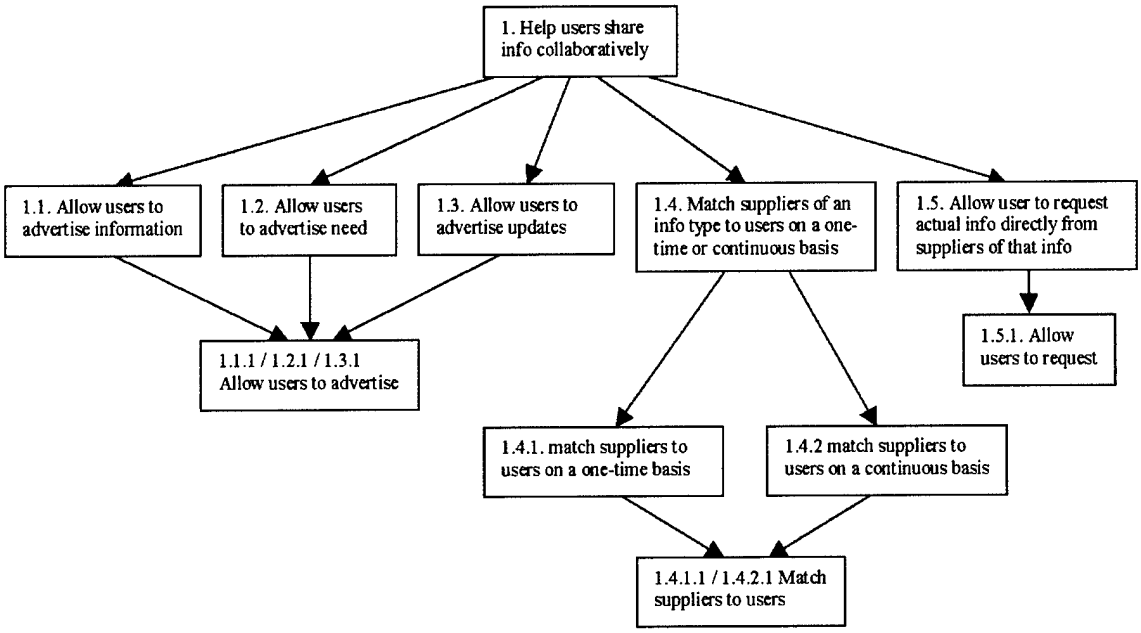

Figure 45: Agent-based collaboration - Goal Hierarchy Diagram

Agent-Based Collaboration – Roles

Roles	(goals)
Consumer	(1.2, 1.5)
Register	(1.x.1, 1.4.*)
Supplier	(1.1, 1.3, 1.5.1)

Figure 46: Agent-based collaboration - roles

Agent-Based Collaboration – Sequence Diagram(s)

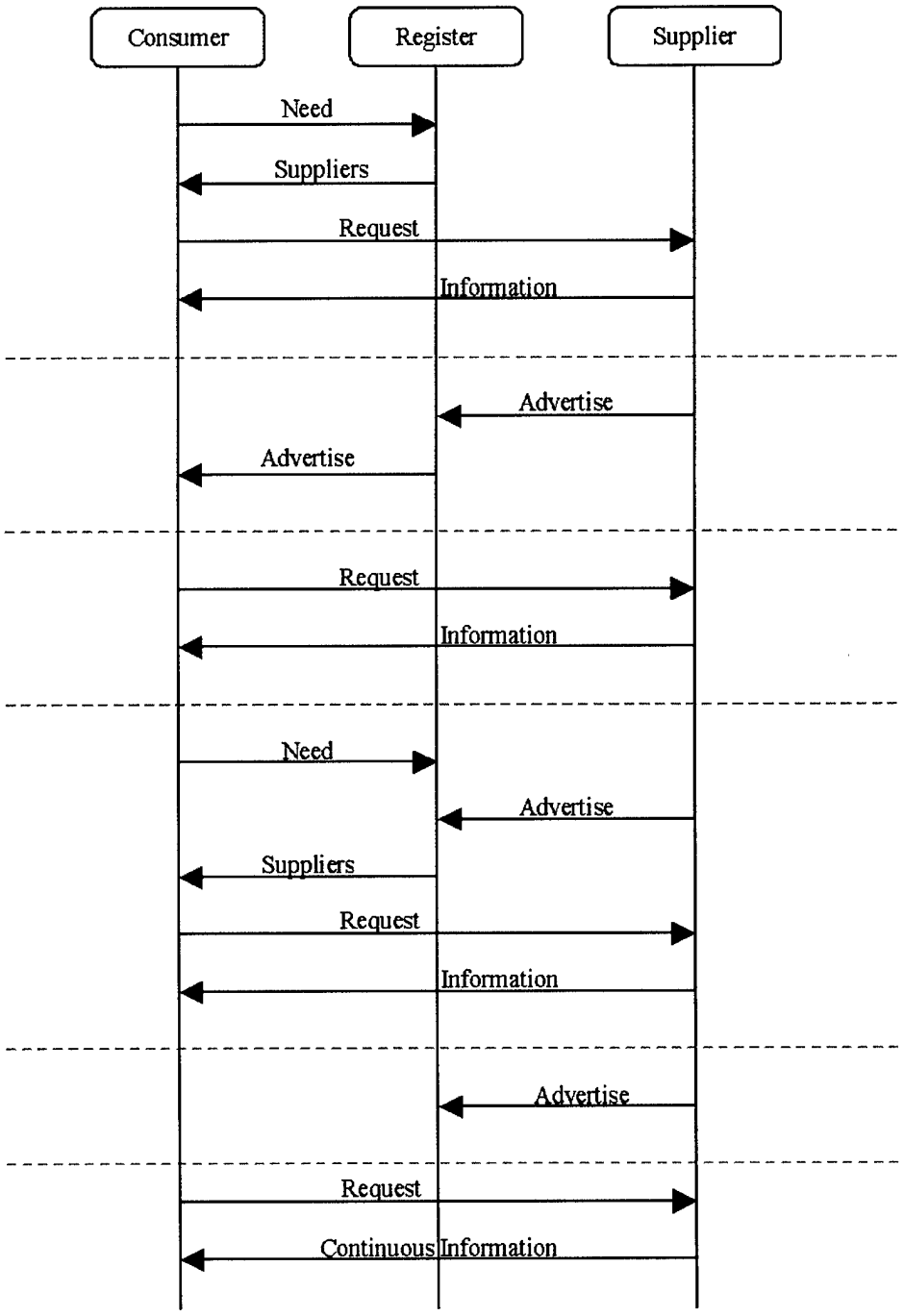

Figure 47: Agent-based collaboration - Sequence Diagrams

Agent-Based Collaboration – Roles and Agent Class Diagram

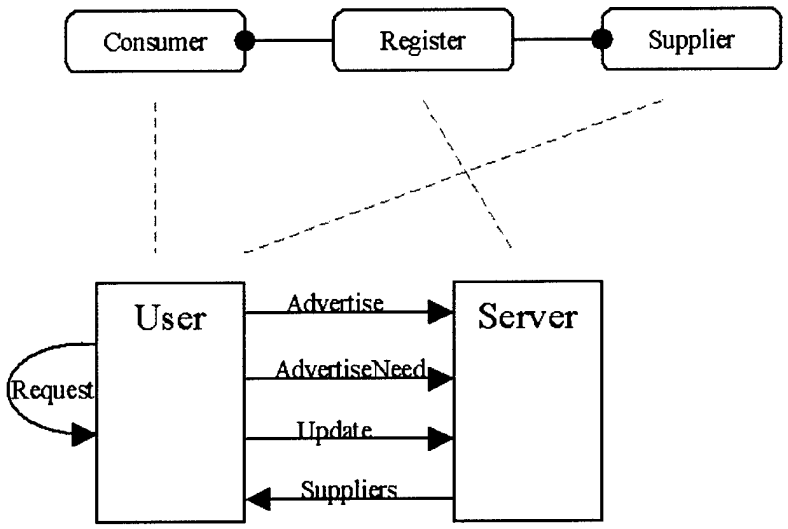

Figure 48: Agent-based collaboration - Roles and Agent Class Diagram

Agent-Based Collaboration – Conversation Creation Steps

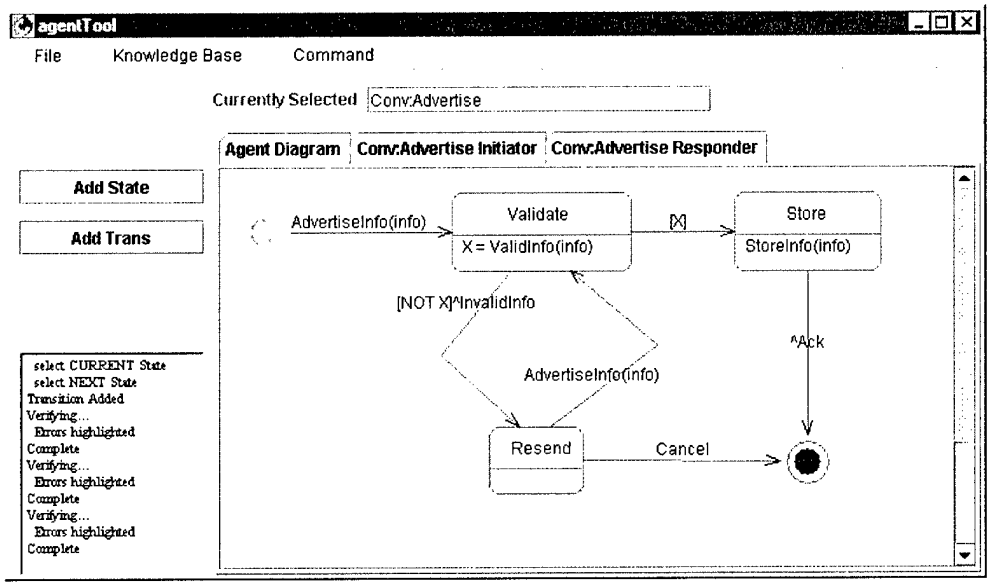

Figure 49: Conversation Initiator

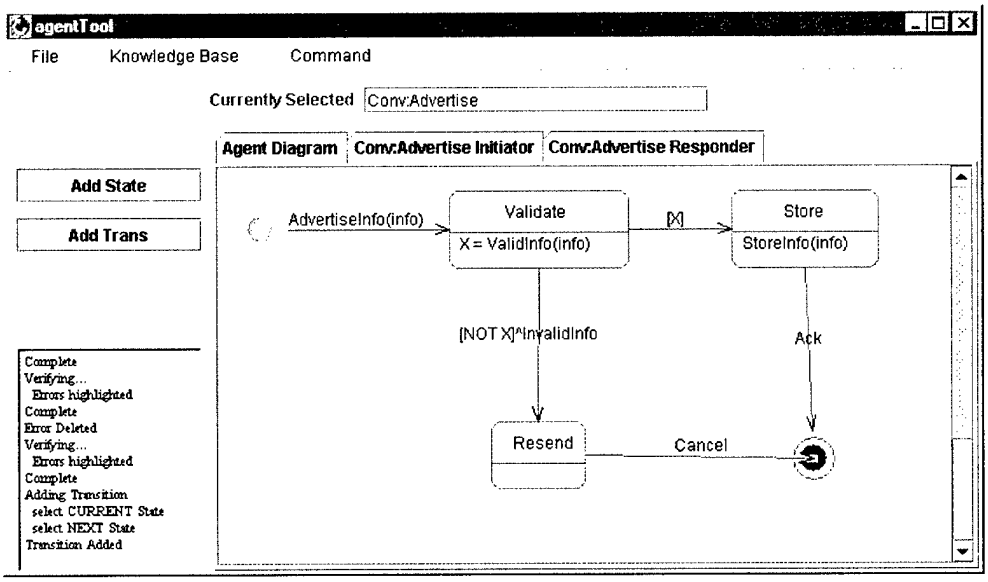

Figure 50: Invalid Conversation Responder - Step 1

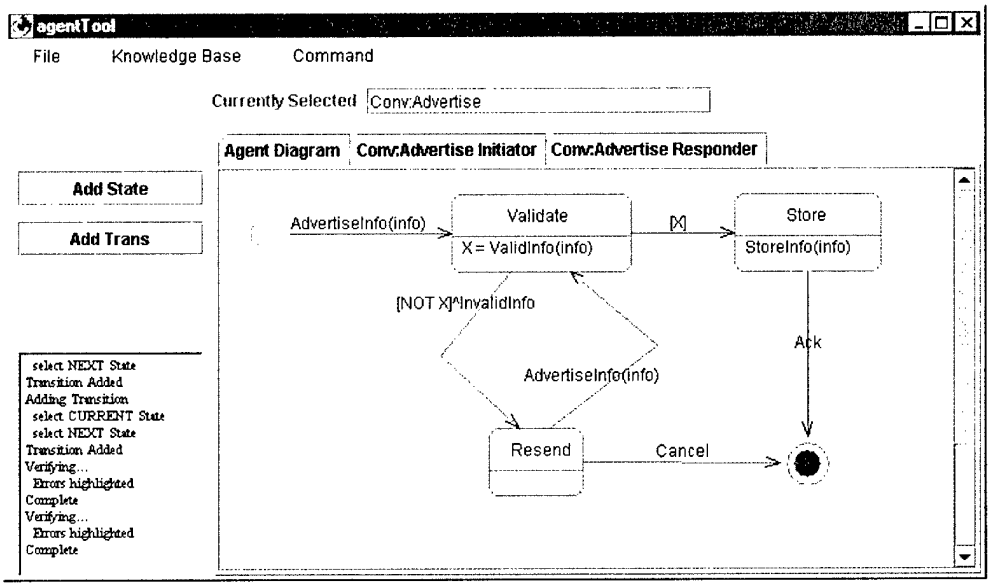

Figure 51: Invalid Conversation Responder - Step 2

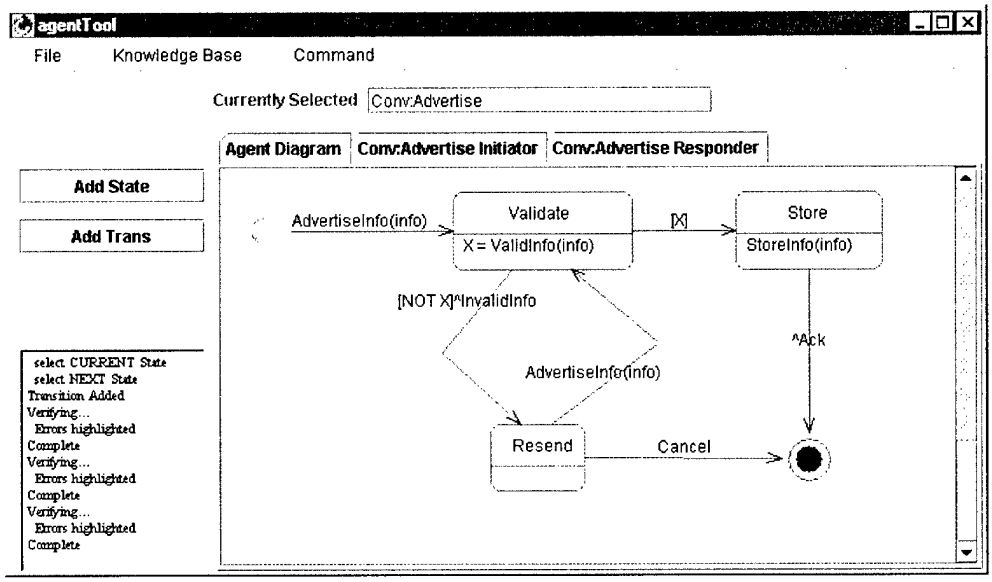

Figure 52: Valid Conversation Responder

Vita

Captain Mark F. Wood was born on 25 May 1971 in Seattle, Washington. He graduated from Bellevue High School in Bellevue, Washington in June 1989. He attended the New Mexico Military Institute in Roswell, New Mexico for one year, and then entered the United States Air Force Academy in June 1990. He was commissioned and graduated with a Bachelor of Science degree in Computer Science in June 1994.

His first assignment was at Keesler, AFB as a student in Basic Communications Officer Training in August 1994. In January 1995, he was assigned to the Defense Information Systems Agency at the Pentagon, where he served as the operations officer of the National Military Command Center Command and Control System. In August 1998, he entered the Graduate School of Engineering and Management, Air Force Institute of Technology. Upon graduation, he will be assigned to USSTRATCOM, Offut AFB.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	7 March 2000	Master's Thesis

4. TITLE AND SUBTITLE

Mulagent Systems Engineering: A Methodology for Analysis and Design of Multiagent Systems

5. FUNDING NUMBERS

6. AUTHOR(S)

Mark F. Wood, Captain, USAF

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 P Street, Building 640
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/00M-26

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFOSR/NM
Attn: Captain Freeman Alex Kilpatrick
801 North Randolph Street
Room 732 9-65
Arlington VA 22203-1977
(703) 696-6565

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Maj Scott A DeLoach, ENG, Phone: (937)255-3636, Ext. 4622

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 Words*)

This thesis defines a methodology for the creation of multiagent systems, the Multiagent Systems Engineering (MaSE) methodology. The methodology is a key issue in the development of any complex system and there is currently no standard or widely used methodology in the realm of multiagent systems. MaSE to covers the entire software lifecycle, starting from an initial prose specification, and creating a set of formal design documents in a graphical style based on a formal syntax. The final product of MaSE is a diagram describing the deployment of a system of intelligent agents that communicate through structured conversations. MaSE was created with the intention of being supported an automated design tool. The tool built to support MaSE, *agentTool*, is a multiagent system development tool for designing and synthesizing complex multiagent systems.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Agents, Multiagent, Methodology, Agent Design, Agent Analysis	126
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL