AFIT/GCS/ENG/02M-03

THESIS

Presented to the faculty of the Graduate School of Engineering & Management

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Science)

Jonathan M. DiLeo, B. S.

First Lieutenant, USAF

March 2002

.

AFIT/GCS/ENG/02M-03

ONTOLOGICAL ENGINEERING AND MAPPING IN MULTIAGENT SYSTEMS DEVELOPMENT

THESIS

Jonathan M. DiLeo, B. S.
First Lieutenant, USAF

Approved:

Lt Col Timothy M. Jacobs (Chairman)	date
Maj. Karl Mathias (Member)	date
Dr. Gregg Gunsch (Member)	date

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my faculty advisor, Lt Col Timothy Jacobs, for his guidance and support throughout the course of my research effort. I would like to thank Maj Karl Mathias for his advice and guidance regarding the ranking models used in this thesis and to Dr. Gregg Gunsch for serving on my thesis committee. I would also like to express my appreciation to my sponsor, Capt David Marsh, from the Air Force Research Labs for his support in this endeavor.

Finally, I would like to thank my wife for her love and understanding during my graduate studies. Without her support and tireless proofreading, the last eighteen months would have been impossible.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

ABSTRACT

Multiagent systems have received much attention in recent years due to their advantages in complex, distributed environments. Previous work at the Air Force Institute of Technology has developed a methodology for analyzing, designing, and developing multiagent systems, called Multiagent Systems Engineering (MaSE). MaSE currently does not address the information domain of the system, which is an integral part of designing proper system execution.

This research extends the MaSE methodology to include the use of ontologies for information domain specification. The extensions allow the designer to specify information flow by using objects from the ontology as parameters in agent conversations. The developer can then ensure system functionality by verifying that each agent has the information required to accomplish the system goals.

To fully describe the system design, the developer must describe the relationships between the system ontology and any agent component ontologies. This research also developed a ranking model to assist the user with creating such mappings, to show the relationships between the objects in the ontologies.

ONTOLOGICAL ENGINEERING AND MAPPING IN MULTIAGENT SYSTEMS DEVELOPMENT

I. Introduction

The Department of Defense is currently integrating its various distributed information systems to increase information superiority for the warfighter. *Joint Vision 2020* specifies the development of a global information grid as part of information superiority and emphasizes the importance of "information systems and equipment that enable a common relevant operation picture…that can be accessed by any appropriately cleared participant" for interoperability between the various services [Shelton 2000]. An appealing solution to integrating legacy systems into this global grid is the use of multiagent systems. The resulting system must be robust, reliable, and secure to meet our warfighting needs.

Constructing multiagent systems involves all the problems of traditional distributed systems along with the problems that arise from the behavior of the individual agents. Designers need an engineering approach for system development of multiagent systems to address and avoid these problems.

Integrating legacy systems further increases the difficulties of designing multiagent systems due to the varying semantics of different information systems. Different systems can have different terms that represent the same concept, introducing a complication in integrating the systems. Any complete methodology for building multiagent systems must address the data models used by the system and the individual components in the system.

1.1 Background

In recent years, interest in multiagent systems has increased as designers look for new methods of solving problems. Growth of the Internet has led to increased distribution of information and system resources among multiple machines. Agents act as abstractions for the individual components that coordinate to accomplish tasks in these distributed systems. Distributed environments lend themselves to a multiagent system design, so system designers have been looking for ways to use multiagent systems to solve their problems.

Multiagent Systems Engineering (MaSE) has been developed at AFIT to assist in the development of multiagent systems by leading the designer from the initial system specifications to a set of formal design documents [DeLoach and others 2001]. The transformations from each step in MaSE are formally defined and provide the engineering approach needed for multiagent system engineering. Despite its benefits in multiagent systems design, however, MaSE fails to address the design of the information domain. The designer constructs a set of design documents that do not address the semantics of the data passed between the agents.

This research corrects this deficiency by extending MaSE to include steps to specify the information domain and to use the objects defined in that domain, providing the designer with a complete set of design documents. This research uses ontologies to specify the classes, properties, object constants, and axioms that a system and its components use to represent the domain in which they operate.

1.1.1 Definition of an Ontology

The word *ontology* was taken from philosophy where it represents the study of the nature of being. There has been much debate on the exact definition of an ontology when used for knowledge engineering or artificial intelligence. Nicola Guarino compares the various definitions and the differences in their meaning in [Guarino 1996]. The most common definitions state that an ontology is a specification of a conceptualization [Gruber 1996] or that an ontology is the shared understanding of some domain of interest [Uschold 1996]. This research uses the latter definition, specifically that an ontology defines classes, functions, object constants, and axioms to constrain meaning of some type of world view of a given domain. The ontology describes the concepts and relationships used to interact in the domain.

1.1.2 Importance of Ontologies in Multiagent Systems

The specification of agent and system ontologies is important for the communication of the agents in the system and the future reuse of agent components. The benefits are further explained in the following paragraphs.

1.1.2.1 Communication

Ontologies are the key for exchanging knowledge between agents. Agent communication languages such as the Knowledge Query and Manipulation Language (KQML) specify the syntax used to communicate, but the semantics of the message actually represent the knowledge. Without the mutual understanding that an ontology provides, the knowledge being passed might be misinterpreted by one of the agents. This is one of the reasons presented in [Huhns 1997] for the use of ontologies in agent systems.

1.1.2.2 Component Reuse

Ontologies also allow for the reuse of agents. Braga, Mattoso, and Werner argue the importance of ontologies in the reuse of software components [Braga and others 2001]. Ontologies specify the domain that the component was designed to work in and influence the connectiveness of the components of a software system. Components that are designed for separate domains will not be able to directly communicate with each other and some type of translating component must be built. If the initial designer had not specified the ontology for the component, future designers would not know if a translator is needed for the component to properly operate once integrated into a new software project.

1.2 Problem

This research focuses on the need for an engineering-based methodology for building heterogeneous multiagent systems. Any complete methodology should assist the user in developing the structural, behavioral, and information models of the system from the initial system requirements. This research provides such a methodology by extending MaSE to include development and use of information domain specifications.

1.2.1 Goals

The main objective of this research is to develop a methodology for building heterogeneous systems using ontologies to specify the information domain. This involves the creation of a methodology for constructing and using ontologies at the system and component level of multiagent systems. As eluded to in the previous section, the components of a multiagent system can have a data model different from the system model. In this case, designers must specify the mapping of the classes in the ontologies. An example mapping is from an *Automobile* object to a *Car* object to illustrate that the two objects represent the same semantic content. As the number of classes in the data models increase, the process of mapping every class in the component ontology becomes more tedious and difficult. This research focuses on assisting the designer with this process by suggesting appropriate mappings.

This related goal involves the construction of a ranking model to evaluate the probability that two classes represent the same semantic content. This model can then rank every class in the system ontology based on the possibility of it matching an object from the component ontology. This ranking suggests the most appropriate mapping for the component class.

1.2.2 Approach

To accomplish these goals, this research first determines the requirements needed by multiagent systems for representing and using the information domain. MaSE is then extended to include the construction of ontologies for the system and components and to use the resulting ontologies throughout the system design. By modifying existing methodologies, this research created a methodology for building multiagent system ontologies. Extended MaSE is used to develop a sample multiagent system to ensure the resulting design documents fully describe the information, behavioral, and structural domains of the system.

This research develops a ranking model to compare objects using their underlying characteristics and attributes. These comparisons yield the probability that two objects represent the same semantic content. This research also develops three implementations of the model, with each comparing the attributes in a different manner, and tests them against sample data models to determine which implementation operates with the most precision.

1.3 Thesis Scope

The scope of this research is limited to integrating the use of ontologies into MaSE and agentTool, the automated assistant for MaSE. The general concepts of developing and using ontologies as an information domain specification mechanism can be applied to other existing multiagent system development methodologies. This research addresses the changes necessary to include these concepts in MaSE and agentTool to demonstrate that ontologies can fulfill the requirements of an information model in multiagent systems development.

1.4 Thesis Overview

The remainder of this document is organized as follows. Chapter 2 provides background material on ontologies and the MaSE methodology. Chapter 3 describes the requirements and evaluation criteria for using domain representation in multiagent systems and how ontologies are used to fulfill these requirements. It also discusses the appropriate places to introduce domain representation into MaSE and the steps taken to develop an information retrieval ranking model to assist with mapping component ontologies. Chapter 4 describes the extended MaSE methodology, including the creation and use of ontologies in multiagent systems. Chapter 5 describes the exact characteristics used by the ranking model to evaluate the probability that two objects represent the same semantic concept. Chapter 6 describes an example system developed with the extended MaSE methodology and agentTool. The results and experiences from the process of building this example system provide an evaluation of the extended MaSE. Chapter 6 also discusses the results from the ranking model experiments to determine the accuracy of the three implementations. Chapter 7 presents the conclusions reached by this research and describes possible future research topics.

II. Background

Before considering how to integrate the fields of multiagent systems and ontological engineering, one should first look at each field separately. This chapter is divided into two main parts to discuss research in the field of ontological engineering and multiagent systems engineering. The first part discusses the terms used in an ontology, current methodologies for building domain ontologies, specification languages for ontologies, and existing graphical toolkits to assist with the construction of an ontology. The second part discusses existing methodologies for building multiagent systems, including Multiagent Systems Engineering (MaSE) and agentTool, a tool developed to assist in applying MaSE.

2.1 Ontologies

An ontology is an explicit description of objects (or classes) in a domain, properties describing the features and attributes (or slots) of the object, restrictions on the slots (also known as facets), and relations between the objects in a domain [Noy and McGuinness 2001]. Classes are the main building blocks in an ontology. A class can have subclasses that are objects, which are more specific than the superclass. For example, a *Car* class could represent all cars, while *CompactCar* and *SportsCar* are subclasses of the *Car* class.

Slots describe the properties of objects. A car has a color and a Vehicle Identification Number. These properties can be defined as slots for the *Car* class. Cardinality, type, and range are the facets of a slot. Cardinality refers to the number of values a slot can have, while type refers to the types of values that can fill a slot. String, Number, Boolean, and Instance are common value types [Noy and McGuinness 2001]. The range of a slot is used for Instance value types and specifies which objects in the ontology can fill that slot. For example, consider *owner* a slot of the *Car* class. The cardinality of the slot is one, because there is only one owner per car. The type would be an Instance type with a range of the *Person* class, where *Person* is an object used to represent people in the ontology.

Terms used in the Unified Modeling Language (UML) of object-oriented software engineering are similar to those used in ontological engineering. Classes are analogous in each discipline and attributes of the class are slots. Relationships in UML are specified as Instance slots with facets used to specify the cardinality of the relationship.

2.1.1 Existing Methodologies for Building Ontologies

Although ontological engineers have designed numerous ontologies, no robust methodology for constructing ontologies exists. Uschold and Gruninger first proposed a framework for a methodology to construct ontologies to encourage research into developing a more detailed and scientific approach to ontological construction [Uschold and Gruninger 1996]. Methontology and the IDEF5 Method have both been proposed as general-purpose methodologies for building domain ontologies.

Domain ontologies focus on defining all the concepts and relationships in a specific domain. Engineering projects such as TOVE [Gruninger and Fox 1995] and Enterprise [Fraser and others 1995] developed ontologies for the Enterprise domain and the designers then published papers describing their experiences and the process they followed to construct the ontology. The Enterprise domain consists of the business domain involving business transactions, inventory, sales, etc. The process used in these projects is abstracted to the framework presented by Ushcold and Gruninger, as discussed in Section 2.1.1.1. This section describes the three general purpose methodologies used in constructing domain ontologies.

2.1.1.1 Uschold and Gruninger

Although not designed as a complete methodology, Uschold and Gruninger present a skeleton methodology for building domain ontologies in [Uschold and Gruninger 1996]. This skeleton is a good starting point for developing a complete methodology. They propose four stages: identify purpose and scope, building the ontology, evaluation, and documentation.

2.1.1.1.1 Identify Purpose and Scope

The designer must describe why the ontology is being developed as well as the range of intended users of the ontology. This facilitates ontology reuse by allowing others to quickly see the reason the ontology was constructed and what information the ontology contains.

2.1.1.1.2 Building the Ontology

With the purpose and scope of the project defined, the user begins to construct the ontology. This stage captures the domain and then codes the ontology into a representative language.

Capturing the domain develops the ontology by identifying the key concepts and relationships in the domain, producing precise unambiguous text definitions for each of them, and identifying the terms to refer to them. A very important part of capture is that the concepts, definitions, and terms need to be agreed upon. For a closed multiagent system, the designer can force the agents to follow a specific ontology, but in an open system people must agree to use a specific ontology. Without agreement on the capture phase, people will not want to use the results.

Coding takes the concepts and relationships along with the definitions and terms for each from the developed ontology and represents them in some formal language. This involves committing to a *meta-ontology* which be used to specify the output from the capture phase. A representation language is chosen and then the ontology is written in that language.

2.1.1.1.3 Evaluation

This stage reviews the produced ontology to ensure that it is complete and consistent. Work done by [Gómez 1995][Gómez 1996][Gruninger 1995] can be used to assess the proposed ontology during this stage. Gómez and Gruninger recommend a global technical evaluation to ensure well-defined properties in each of the definitions of the ontology.

2.1.1.1.4 Documentation

Skuce [Skuce 1995] points out that one of the main problems in the effort of knowledge sharing is the lack of adequately documented ontologies. To properly document the ontology, the designer should note all assumptions about the ontology and the process used to describe it.

2.1.1.2 Methontology

Methontology is based on Fernández's experience constructing an ontology for the Chemistry domain and is a "structured" method to build ontologies [Fernández and others 1997]. In each of the stages of the methodology, the authors encourage evaluation and documentation of all the outputs from the stage.

The first stage of Methontology is specification. This stage produces a specification document, which includes the purpose of the ontology under development, the level of formality for coding the ontology, and the scope, characteristics, and granularity of the ontology. Having defined the project, the analyst begins the knowledge acquisition stage, obtaining information to help produce a set of terms and their meanings. Any form of knowledge acquisition is allowed. (such as brainstorming, interviews, reviewing other ontologies, etc.)

Once the designer constructs a set of terms from the information obtained, the conceptualization stage structures this domain knowledge in a conceptual model to describe the problem and the solution in the vocabulary developed in earlier stages. The next step is integration, where the ability to integrate any existing ontology is considered. The authors highly encourage reusing existing ontological definitions. If the designer finds an ontology to reuse, he/she should develop an integration document, as shown in Figure 1.

Meta-Ontology	The frame-ontology in Ontolingua	
Term in your Conceptualization	**Ontology to be reused**	**Name of the term in the ontology**
Kilometer	Standard-Units in Ontolingua	Kilometer
Centimeter	Standard-Units in Ontolingua	Undefined
Exponent	KIF-Numbers in Ontolingua	Expt

Figure 1 Example Integration Document [Fernández and others 1997]

In the final stage, the analyst implements the ontology, coding it into a formal language. See Section 2.1.2 for a discussion on formal languages typically used for specifying ontologies. An example chemical ontology is developed using this methodology in [Fernández and others 1999]. The Ontology Design Environment, discussed in Section 2.1.3 assists the designer in the development of ontologies using Methontology.

2.1.1.3 KBSI IDEF5

The IDEF5 method is based on an evolving prototype model designed to assist in the creation and management of domain ontology models [KBSI 1994]. IDEF5 uses the terms *kind* to refer to classes and *characteristics* to refer to attributes of the kinds. The methodology is divided into five activities: organizing and scoping, data collection, data analysis, initial ontology development, and ontology refinement and validation.

2.1.1.3.1 Organizing and Scoping

The designer must first establish the purpose, context and viewpoint for the ontology development project and assign roles to the team members. The viewpoint describes from what perspective the domain is being designed from, e.g. *from the project manager's viewpoint*.

2.1.1.3.2 Data Collection

With the project defined, the designer must now determine what information the ontology should contain. IDEF5 defines three modes of data collection: observation of activities, interviews and analysis with domain experts, and direct transcription of data from documents from the domain. Six different types of forms are used to catalog the source material and term pool. The term pool is similar to a list created by brainstorming; it represents meaningful terms relevant to the development of the ontology. These terms might become kinds, relations, or characteristics in the ontology.

2.1.1.3.3 Data Analysis

The goal of this step is to analyze the source material and term set from the previous step and construct an initial characterization of the ontology. Listing objects of interest in the domain is the first part of this characterization. The viewpoint and context of the project, as determined in the organizing and scoping activity, guides the level of detail for specifying objects. The team should then look for systems of objects that work together to accomplish common goals.

2.1.1.3.4 Initial Ontology Development

The Developing Initial Ontology activity develops proto-kinds, proto-properties, proto-attributes, and proto-relations. The *proto* simply refers to the fact that these are the first attempt at specifying these concepts in the ontology. IDEF5 provides a visual representation of the ontology so that the ontology can be developed graphically.

2.1.1.3.5 Ontology Refinement and Validation

The designer must now ensure that the developed ontology contains all domain information. This activity completes the design process by validating and refining the proto-concepts in the developed ontology. The analyst should make instances of the proto-kinds using examples from data in the domain. Any information from the domain that cannot be represented by a proto-kind should be analyzed to determine if it is needed in the ontology. If it is, then a new or expanded kind must be developed to incorporate the information. Finally, the kinds should be checked to ensure there are no duplicates. Relations are verified in a similar procedure to ensure that there are no missing, duplicate, or contradictory relations. The ontology can then be changed from the IDEF5 graphical schematic language to the IDEF5 elaboration language. The elaboration language is a structured textual language with the full power of first-order logic.

2.1.1.3.6 Comparison to Methontology

IDEF5 is a more mature methodology in that its steps are more detailed than those in Methontology. The record keeping in IDEF5, however, may become cumbersome to the developer. When collecting possible terms, the methodology requires the designer to assign each term a tracking number, along with information regarding how the term was produced. This level of tracking may be appropriate for critical system development, but some designers may find it a little too involved.

2.1.2 Specification Languages for Ontologies

Each of the methodologies described in the previous section describes the encoding of the ontology into a formal language. LOOM[MacGregor 1991], Epikit[Genesereth 1990], Algernon[Crawford and Kuipers 1989], CycL[Lenat and Guha 1990], and KEE[Fikes and Kehler 1985] are all languages that can be used to represent ontologies, but Ontolingua[Gruber 1992] is used most frequently and DAML+OIL[van Harmelen and others. 2001] is the most recently developed language. Ontolingua was built as a language used to translate between the other specification languages. It consists of forms that allow definition of classes, relations, objects, functions, and theories based on a standard notation and semantics called Knowledge Interchange Format (KIF) [Genesereth 1998]. The syntax of Ontolingua definitions consists of a name, argument list, and a documentation string, followed by a set of labeled KIF sentences. To define a class, a designer uses the form:

```
(define-class class-name (?instance-variable)
        "documentation string"
        :def or :iff-def KIF-sentence
        :constraints KIF-sentence
        :sufficient KIF-sentence
        :equivalent KIF-sentence
        :default-constraints KIF-sentence)
        :axioms KIF-sentence
```

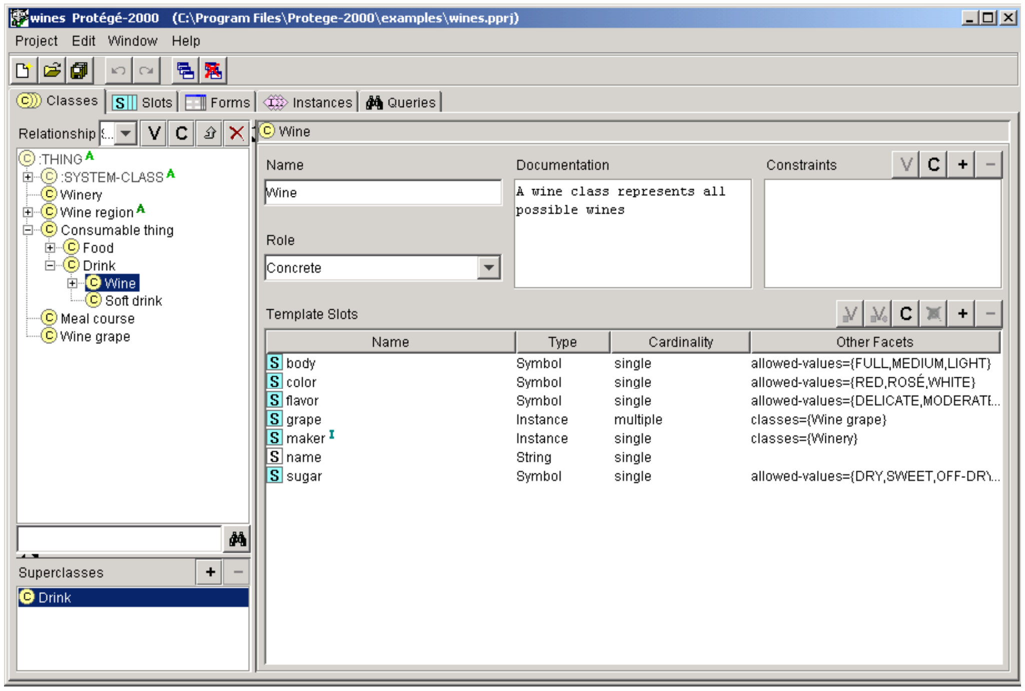
The benefits of Ontolingua are its expressive power and the large amount of already constructed ontologies [Ontolingua]. DAML+OIL is part of the DARPA Agent Markup Language Program designed to provide constructs for creating ontologies and marking information on the web so that it is machine readable and understandable. DAML+OIL is based on the Extensible Markup Language (XML) [W3C 1998] Uniform Resource Identifiers(URI) [Berners-Lee and others 1998], and the Resource Description Framework(RDF) [W3C 1999]. The benefit of DAML+OIL is the ability to use XML instead of predicate logic to specify the ontology. There is also a repository of existing DAML+OIL ontologies that can be downloaded [DAML+OIL]. To define a class, the designer would use the following form:

```
<daml:Class rdf:ID=”class-name”>
        <rdfs:label>class-name</rdfs:label>
        <rdfs:comment>
        A Comment
        </rdfs:comment>
</daml:Class>
```

2.1.3 Graphical Toolkits for Building Ontologies

Graphical toolkits allow the designer to graphically view the information being encoded into the ontology, thereby reducing information overload. The Ontology Design Environment (ODE), Protégé 2000, and OILEd exist as free software that can be used to develop ontologies. ODE is designed to assist in the development of ontologies using the Methontology methodology. OILEd is a program that allows users to develop ontologies and specify them in the DAML+OIL language.

Protégé 2000 is the most robust of the three programs because it allows for plugins to be downloaded and inserted into the program. These plugins can be used to import and export the ontology into different ontology specification languages and ontology visualization programs. A screen capture of Protégé 2000 is shown in Figure 2.



Name: Wine

Documentation: A wine class represents all possible wines

Role: Concrete

Name	Type	Cardinality	Other Facets
body	Symbol	single	allowed-values={FULL,MEDIUM,LIGHT}
color	Symbol	single	allowed-values={RED,ROSÉ,WHITE}
flavor	Symbol	single	allowed-values={DELICATE,MODERATE...
grape	Instance	multiple	classes={Wine grape}
maker	Instance	single	classes={Winery}
name	String	single	
sugar	Symbol	single	allowed-values={DRY,SWEET,OFF-DRY...

Figure 2 Protégé 2000 Main User Interface

The program has tabs which allow the designer to look at the objects and slots in the ontology. The objects are ordered by their taxonomy, that is by their subclass/superclass structure. When an object is clicked, the program displays the information from the object, including the slots for the object and the facets on the slots.

2.2 Multiagent Systems

An agent is anything that can perceive its environment through sensors and act upon the environment through effectors [Russel and Norvig 1995]. An intelligent agent is an agent that takes the best possible action in a situation in order to accomplish its goals. Determining what exactly characterizes the best possible action splits the field of artificial intelligence into those who feel intelligent agents should be rational and those that prefer agents to act like humans. The distinction between the way the agents operate is not important for this research.

A multiagent system consists of multiple intelligent agents interacting to accomplish their goals. In Multi-Agent Systems, Ferber defines a multiagent system as a system composed of the following elements:

- An environment, E.
- A set of objects, O. These objects can be perceived, created, modified, and destroyed by agents.
- A set of agents, A, representing the active entities of the system. ($A \subseteq O$)
- A set of relations, R, which link objects to each other.
- A set of operations, Op, which allow the agents in A to interact with the objects in O.

This research is concerned with goal-based multiagent systems. Each multiagent system has a set of goals that the agents in the system are designed to obtain. The exact behavior of the various agents is determined during the analysis and design of the multiagent system based on the goals required by the system specification.

Interest in multiagent systems has increased as resources become more and more distributed. With the advent of the Internet, many problems have become distributed so much that a centralized solution will no longer work. Some problems such as air traffic control naturally lend themselves to multiagent systems due to their distributed nature. Various methodologies have been developed to provide an engineered approach to the development of multiagent systems. Three such methodologies are discussed in the next sections.

2.2.1 Multiagent Systems Engineering Methodology

The Multiagent Systems Engineering (MaSE) methodology has been a topic of the Agent Research Group at AFIT for the last few years. Research has focused on developing a robust methodology for constructing multiagent systems. MaSE divides the development of multiagent systems into analysis, design, and implementation phases [DeLoach and others 2001]. MaSE consists of three steps in the analysis phase and four steps in the design phase. The phases and their steps are shown in Figure 3. The implementation phase uses the documents from the previous phases to program the system into code.

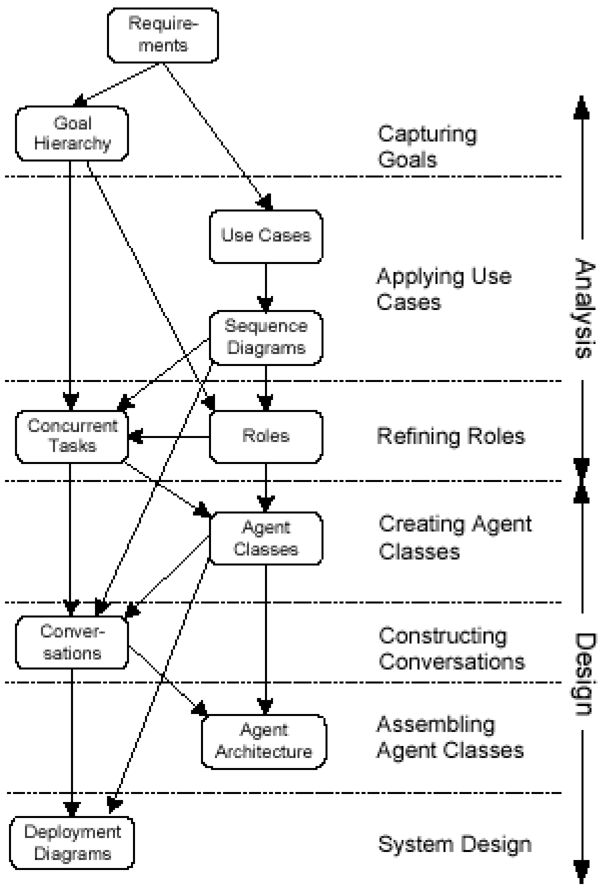


Figure 3 MaSE Phases, Steps and Models

2.2.1.1 Analysis Phase

The analysis phase is concerned with establishing a set of roles and assigning tasks to those roles to describe the system requirements. The capturing goals step consists of transforming the initial system specification into a goal hierarchy. Goals are used to reflect the purpose of an agent's actions. This step allows the designer to organize the goals that the system needs to accomplish. Figure 4 is an example of the output from this step.

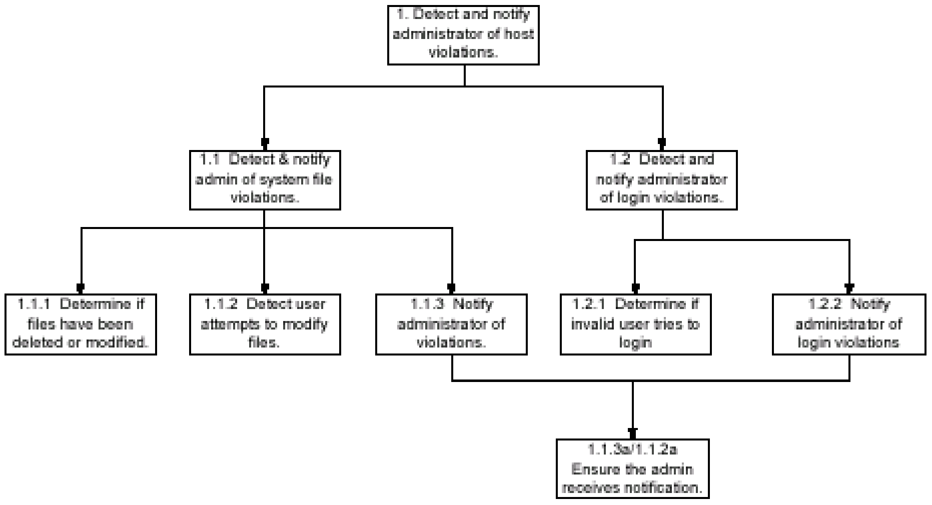

1. Detect and notify administrator of host violations.
1.1 Detect & notify admin of system file violations.
1.2 Detect and notify administrator of login violations.
1.1.1 Determine if files have been deleted or modified.
1.1.2 Detect user attempts to modify files.
1.1.3 Notify administrator of violations.
1.2.1 Determine if invalid user tries to login
1.2.2 Notify administrator of login violations
1.1.3a/1.1.2a Ensure the admin receives notification.

Figure 4 Example Goal Hierarchy Diagram [DeLoach 2001]

The basic scenarios that the system should perform are captured in the applying use cases step. Use cases are created and then transformed into sequence diagrams. The final step of analysis uses the outputs from the first two steps to create roles and assign the tasks to be performed by those roles in the system. Tasks are associated with each role to describe the behavior that the role must have to accomplish its assigned goals. Concurrent tasks are shown graphically using a finite state automaton.

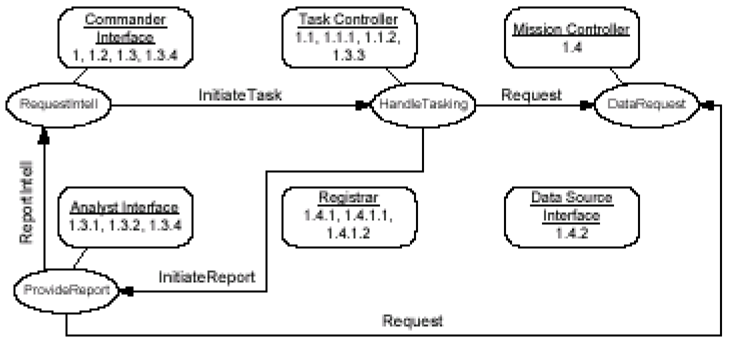


Figure 5 Example MaSE Role Model [DeLoach and Wood 2000b]

Figure 5 is an example of a Role Model in MaSE and Figure 6 is an example Concurrent Task Diagram. Each role is associated with at least one goal from the goal hierarchy. Transitions in the concurrent task diagrams follow the syntax *trigger [guard] ^ transmission(s)*. This syntax represents that a transition is enabled only when the event *trigger* occurs and the condition *guard* evaluates to true. When the transition is executed, the specified *transmission(s)* will be executed by the role. Once in a state, the task remains in that state until all actions defined in that state have been completed and a transition out of the state has been enabled.

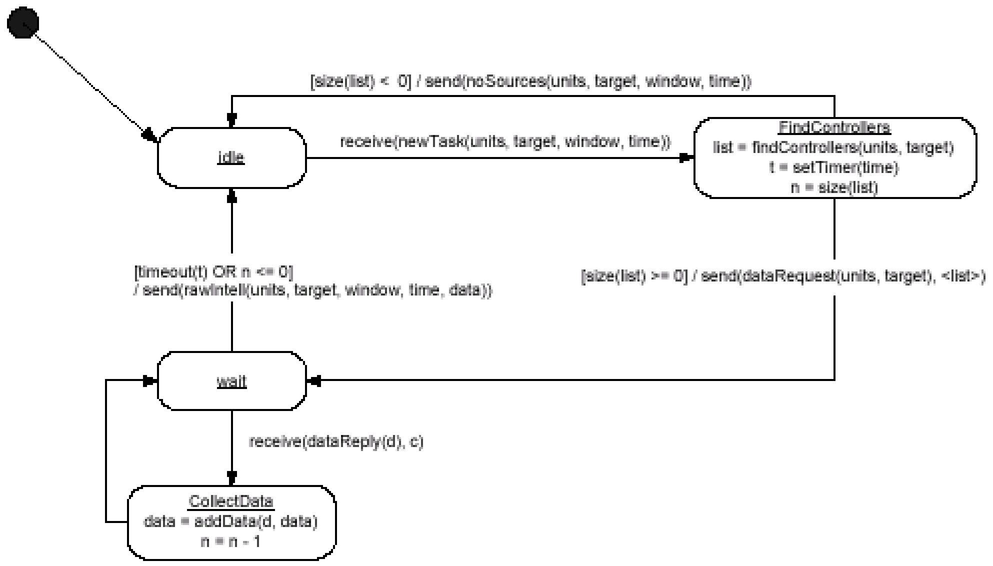

[size(list) < 0] / send(noSources(units, target, window, time))
idle
receive(newTask(units, target, window, time))
FindControllers
list = findControllers(units, target)
t = setTimer(time)
n = size(list)
[timeout(t) OR n <= 0]
/ send(rawIntell(units, target, window, time, data))
[size(list) >= 0] / send(dataRequest(units, target), <list>)
wait
receive(dataReply(d), c)
CollectData
data = addData(d, data)
n = n - 1

Figure 6 Example Concurrent Task Diagram [DeLoach and Wood 2000b]

2.2.1.2 Design Phase

Once the requirements for the system have been specified, the design of the multiagent system begins. The first step in the design phase is creating agent classes, where the roles are assigned to different agent classes. This step creates an Agent Class Diagram that shows the classes in the system and the conversations between classes. These conversations are defined in the constructing conversations step, where finite state automata are used to show the states in a conversation.

Figure 7 is an example Agent Class Diagram with Figure 8 describing one of the conversations between agents in the class. Each conversation has two diagrams: one for the initiator and one for the responder of the conversation.

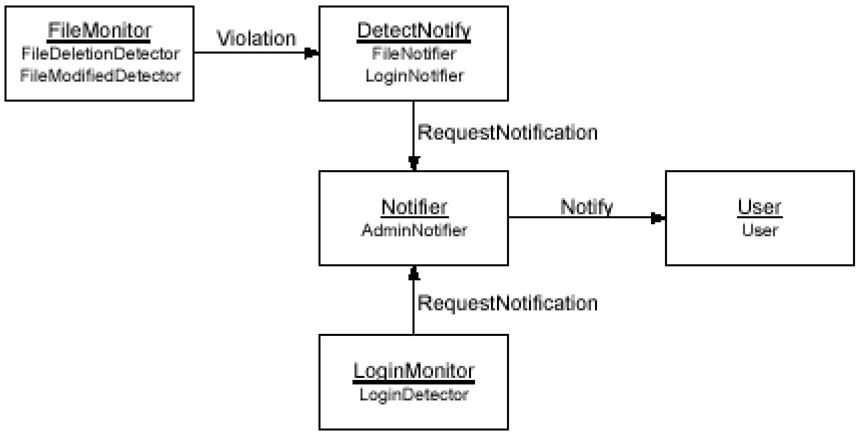


Figure 7 Example Agent Class Diagram {DeLoach 2001]

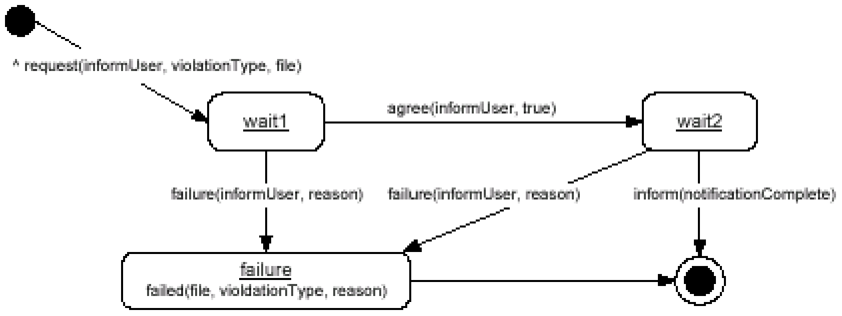

^ request(informUser, violationType, file)
wait1
agree(informUser, true)
wait2
failure(informUser, reason)
failure(informUser, reason)
inform(notificationComplete)
failure
failed(file, violdationType, reason)

Figure 8 Example Conversation Diagram {DeLoach 2001]

The third step in design, assembling agents, defines the components of the architecture. The final step of system design creates a Deployment Diagram to show the amount and location of each type of agent in the system. The outputs from the design steps describe the actions and conversations used in the multiagent systems. The semantics of the parameters passed in those conversations are not currently defined in the MaSE process.

2.2.1.3 agentTool

agentTool is an automated assistant for the MaSE methodology [DeLoach and others 2001][DeLoach and Wood 2000a]. The program has tabs for each of the outputs from the phases in MaSE, and currently allows for the automatic generation of design documents based on the role models and task diagrams. This automatic generation uses transformations from research by Clint Sparkman [Sparkman

2001]. Once the designer specifies the system in agentTool, the program can output Java code for the system. Because ontologies are not incorporated in the MaSE methodology, the code produced from agentTool classifies all parameters as Java Objects. The designer must then go through and change the declarations to the appropriate data structure used to represent the semantic concepts of the parameters. Including the construction of the system ontology removes the necessity of this step, as the parameters can automatically be specified as the correct type.

2.2.2 Additional Multiagent Systems Engineering Methodologies

This section discusses two other methodologies for agent-oriented software engineering. Gaia is one of the earliest developed agent-based software engineering methodologies and MESSAGE is one of the newest methodologies. As MESSAGE was developed to improve on Gaia, this section first discusses the Gaia methodology.

2.2.2.1 Gaia

Gaia is specifically tailored for the analysis and design of agent systems. [Wooldrige and others 2000] The authors wanted to develop a way to describe an agents autonomous behavior and agent interactions, which could not be specified using traditional software development techniques, such as object-oriented development. Figure 9 shows the various models of the methodology and the relationships between them.

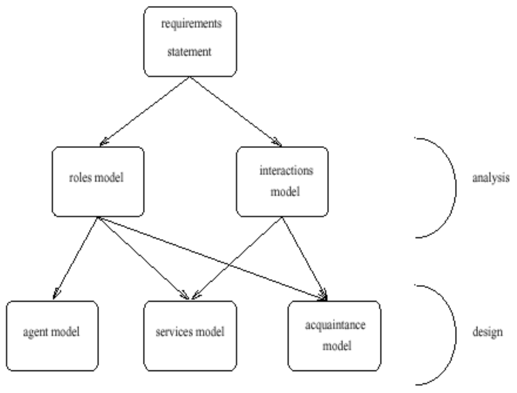
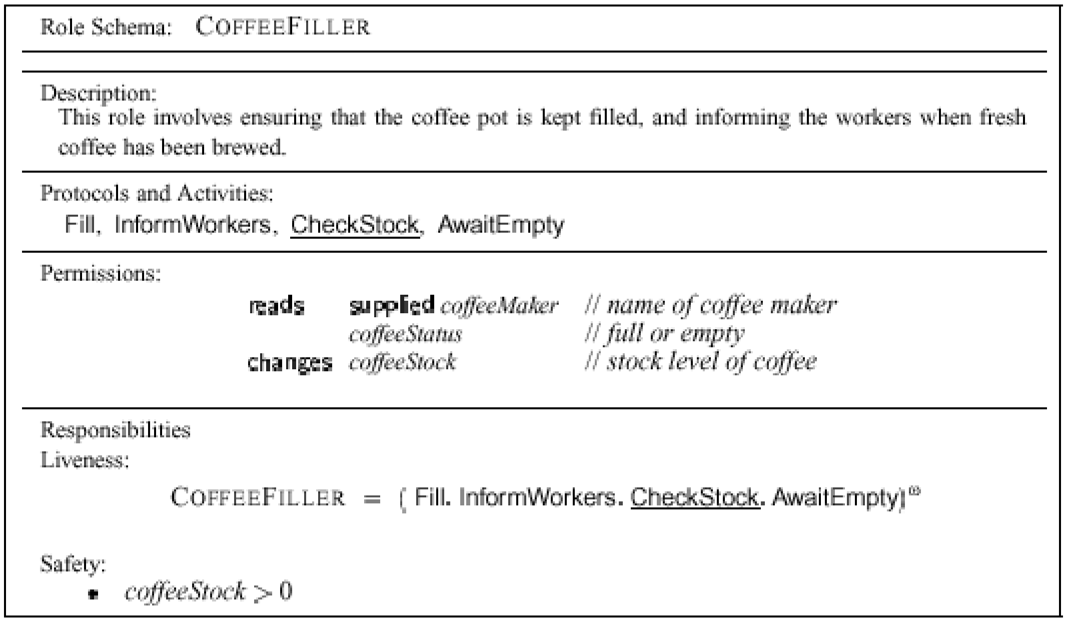


Figure 9 Relationships Between Gaia Models

2.2.2.1.1 Analysis

The analysis model defines the roles that exist in the system and the interaction between these roles. The Roles Model describes each of the roles in terms of their responsibilities, permissions, activities, and protocols, as shown in Figure 10.



Role Schema: COFFEEFILLER

Description:
This role involves ensuring that the coffee pot is kept filled, and informing the workers when fresh coffee has been brewed.

Protocols and Activities:
Fill, InformWorkers, CheckStock, AwaitEmpty

Permissions:

reads	**supplied** *coffeeMaker*	*// name of coffee maker*
	coffeeStatus	*// full or empty*
changes	*coffeeStock*	*// stock level of coffee*

Responsibilities
Liveness:

$$\text{COFFEEFILLER} = (\text{Fill. InformWorkers. } \underline{\text{CheckStock}}\text{. AwaitEmpty})^{\omega}$$

Safety:
- *coffeeStock* > 0

Figure 10 Example Gaia Role Schema [Wooldridge and others 2000]

Permissions describe the rights of the role to access variables in the system. For example, the *CoffeeFiller* may read the variable *coffeeStatus* and change the variable *coffeeStock*. The permissions are key to the role accomplishing its responsibilities. In the above example, the *CoffeeFiller* is responsible for performing the activity *CheckStock* and performing the protocols *Fill*, *InformWorkers*, and *AwaitEmpty*, while ensuring the *coffeeStock* is always greater than zero. Activities are underlined and represent private actions while the protocols represent interactions with other roles.

The Interaction Model describes each of these protocols in further detail. Figure 11 graphically represents the *Fill* protocol. The diagram indicates that the *Fill* protocol is initiated by the *CoffeeFiller* role and involves the *CoffeeMachine* role. The protocol involves filling the supplied coffeemaker and informing the *CoffeeMachine* about the value of *coffeeStock*. Each protocol has a purpose, initiator, responder, inputs, outputs, and processing functions. Figure 11 graphically illustrates all of these attributes, in a very non-intuitive manner.

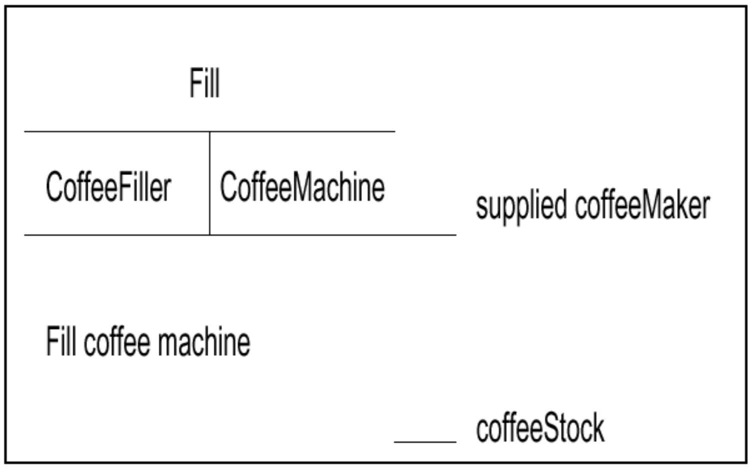


Figure 11 Example Gaia Interaction Model

2.2.2.1.2 Design

Once the roles and the interactions between them are specified, the designer then transforms the models into a level of abstraction such that traditional design techniques can be applied to implement the agents. This is the goal of the design phase, which consists of three models: Agent, Services, and Acquaintance.

The Agent Model describes the structural model of the system, defining the agent types in the system and the roles performed by each type. The model also includes the number of actual instances of each agent type in the implemented system.

Once the roles are assigned to agent types, the Services Model is built to identify the functions of the agents. The functions are derived from the activities performed by the roles assigned to an agent. Each identified service is described in terms of its inputs, outputs, pre-conditions, and post-conditions. The high level behavior of each service is defined, leaving the implementation details for later specification by the developers.

The authors describe the Acquaintance Model as the simplest model in the methodology [Wooldridge and others 2000]. This research considers it the least useful model, defining the communication links that exist between the agent types. The problem is that the model does not define the conversations or messages sent between the agent types, only the fact that some type of communication exists between the two. The model is represented using a directed graph to show the communication flow, to allow the designer to identify any potential bottlenecks in the system.

The system is now described in enough detail to implement the system using traditional design techniques. Gaia represents a good stepping-stone for a complete methodology for building multiagent systems, as it defines the early design stages but finishes without the system fully defined.

2.2.2.2 Methodology for Engineering Systems of Software Agents (MESSAGE)

MESSAGE is a two-year long project to build upon Gaia and other early multiagent engineering methodologies and incorporate existing UML development approaches [Evans and others. 2001]. In this manner, MESSAGE guides the user further along in the design, using UML instead of leaving the analyst to finish using some other technique, as the Gaia methodology does. MESSAGE thus produces a lower-level design, and is more useful for development than Gaia.

MESSAGE defines five views to describe the data, structural, and behavioral models of the system: organization, goal/task, agent/role, interaction, and domain view. The *organization* view shows the agents in the system and the relationships between them. The *goal/task* view uses UML Activity Diagram notation to describe the states that an agent goes through to perform tasks to accomplish goals. These goals are assigned to roles in the *agent/role* view. This view describes what roles an agent performs and what the goals of each role are. To accomplish the goals, the agents have to interact, which is behavior described in the *interaction* view. For each interaction, this view describes the initiator, collaborators, relevant information supplied, and the event that triggers the interaction. The final view, *domain*, defines the information domain of the system so that the appropriate objects can be passed during interactions.

2.2.2.2.1 Analysis

Because the models are interconnected, the methodology recommends starting analysis using three parallel streams of development that are then incorporated. One stream creates the entities and identifies the interactions between the entities in the *organization* view. The second stream identifies the goals, tasks and the relationships among them in the *goal/task* view. The last stream identifies the information domain entities in the *domain* view. These three streams provide an initial set of concepts that are then linked together in the different views and described in detail.

2.2.2.2.2 Design

Once the analyst has fully described the views of the system, he/she begins the design phase to produce computational entities to implement the multiagent system described in the analysis phase. MESSAGE describes two methods of designing the system, without recommending either one. As such, the methodology leaves the user to choose a design strategy appropriate for the situation. One interesting item is that one of the design methods discussed in [Evans and others 2001] involves creating use cases and sequence diagrams, a step which is an analysis step in most software development methods.

MESSAGE provides a set of analysis and design documents that describe the system in greater detail than the Gaia documents. MESSAGE also uses familiar UML modeling techniques to improve the information visualization of the analysis and design documents. However, one difficulty with using the methodology is the parallel development process. Designers must define five different views in a simultaneous process. The iterative process of MaSE allows the designer to perform the same tasks as in MESSAGE, but places the analysis and design process in a step-by-step method than can be iterated through in a manner that may be easier for users to follow.

III. Problem Approach

This research hypothesizes that ontologies developed to represent the information domain of a multiagent system, when coupled with methods formalizing the behavior of multiagent systems, provide design documentation that describes the actions of the agents and the view of the domain necessary to generate a properly functioning system. To evaluate this hypothesis, this research first determines the requirements needed to represent and use the information domain of a multiagent system. These requirements are discussed in this chapter along with the type of ontology used to specify the information domain in MaSE. Finally, this chapter presents a method for constructing ontologies for multiagent systems based on existing methodologies for building domain ontologies. This ontology building method is integrated into MaSE, along with additions to use objects in the data model, forming a complete analysis and design methodology. This chapter presents a discussion of each of these steps.

3.1 Requirements for Domain Representation in Multiagent Systems

3.1.1 Requirements for Information Domain Models in Multiagent Systems

To adequately specify the domain of a multiagent system, a representation must meet certain requirements. These requirements are based on effective software engineering models and how domains are used in multiagent systems. This section describes the requirements that ensure the domain representation is adequately specified in the design of a multiagent system.

The first requirement is that the representation must specify the objects in the domain. The names of the objects in the domain should be unique, and a description of each object should be included. The description is used to help future designers understand what the original developers of the representation meant for the object to semantically represent.

Once the objects are listed, the properties of those objects must also be specified. The properties further describe the semantic content of the object. Properties, such as *has_Color*, are often represented as attributes of an object, but that is an implementation decision.

After the object's characteristics are specified, the representation must then specify the relationships between the objects. The relationships show how objects are related and how they interact in the domain. Relationships are frequently represented as attributes of the object. An example relationship is *Works_for* that exists between a *Laborer* and *Manager* Object. *Works_for* illustrates that a *Laborer* must work for a *Manager*, thus restricting the possible interpretations of the objects.

Axioms define further constraints on the domain objects that cannot be described as properties or relationships. For example, a disjunct axiom between a *Beast* and *Human* object would be used to represent that, although both are a *Thing* object, something cannot be a *Human* and a *Beast*. Other axioms can be specified using first-order logic. The axioms must describe all restrictions on the domain objects that the system uses during system execution. If no type of inference or knowledge-based system is used in the multiagent system, the axioms in a domain representation can be omitted. Axioms specify the preconditions required for proper system execution.

The domain representation must also contain metadata about the domain itself. This metadata aids future developers in understanding the objects, properties, relationships, and axioms. The designers develop the representation to meet a certain goal, and to fit the needs of the system being developed. As such, it is important to describe the system under development. The metadata also includes information for software maintenance such as the name of the representation, names of the original developers, version number, date of creation, etc.

Each specification regarding the domain, objects, relationships, axioms, and metadata must be clear, consistent, and concise. This is necessary so others can easily comprehend the documented aspects of the software system. *Clear* requires the use of unambiguous terms to describe the domain. If a term can have multiple definitions, the surrounding description should ensure that future readers can easily understand which definition is desired. *Concise* balances out clear, by describing the domain with the least amount of terms necessary. The description of each object should be long enough to aid understanding without being so perfuse as to hinder the future readers of the domain representation. The system should be concise in the number of terms used to describe the objects, as well as the number of objects, properties, relations, and axioms specified for the domain. *Consistent* requires designers to specify objects in the same level of detail. Axioms are consistent if they are not contradictory. If the objects and axioms are not consistent, the system may malfunction and future developers may not understand the domain representation.

The final requirement is that the representation should be built to only include information that is used by the system. This aids reuse of the software system, because the domain representation acts as a type of precondition for the proper execution of the system. In order to reuse a multiagent system, the larger system must either have the same view of the domain, or the designer must provide a mapping between the representations of its components. If a multiagent system over-specifies its domain, future reuse could require additional work mapping to objects in the representation that the component system never uses.

3.1.2 Requirements for Information Domain Use in Multiagent Systems Development

Along with the requirements on the domain specification, there exist certain criteria that a methodology for developing multiagent system should meet when describing the information domain. This section discusses each of the criteria and their importance.

A methodology for designing multiagent systems should include a step to allow for the specification of the information domain of the system. Just as it is important to specify the data model in a traditional software development process, the data model for a multiagent system must also be specified. The agents in the system interact by passing messages and these messages frequently involve passing parameters. These parameters are objects of some sort, and without an information domain specification, the methodology cannot address the information contained in these parameters.

The development of this system data model should occur at a logically appropriate time in the multiagent system design methodology. The development should occur prior to describing any information passing by the agents, since the data models are used in specifying the types of objects passed. The construction of the data model should also occur after the designer evaluates the problem domain enough so that he/she knows what information must be included in the data model.

The methodology used to create the information domain specification should be an iterative process to match the iterative nature of software development. If the designer discovers missing or inappropriate information later in the system development, he/she should have the ability to modify the information domain specification appropriately.

Once the system data model is constructed, the multiagent system design methodology should allow the analyst to specify objects from the data model as parameters in the conversations between the agents. To ensure the proper functionality of the multiagent system, the designer must be able to verify that the agents have the necessary information required for system execution. Since the information is represented in the classes of the data model, the design of the methodology must show the classes passed between agents to allow the designer to verify the proper flow of information in the system.

Along with building a system data model, the multiagent system design methodology should allow agents to have their own individual data models. By addressing this capability, the methodology allows for the development of heterogeneous systems. The requirement for a multiagent system to integrate with existing systems often creates such heterogeneous systems. With the various data models comes the requirement to show how the information models relate.

The methodology should provide the ability for the designer to show the relations between the data models in some manner. Showing the relationships indicates to the code developers what information from one model is required to create objects in the other model. Without describing these relations, the developers may not be able to code the conversations between two agents with separate data models, as each agent uses different classes to describe the information.

3.2 Methodology for Including the Information Domain in MaSE

This section discusses the approach of this research to fulfilling the criteria for building and using information domain specifications in multiagent systems. Ontologies will be used to specify the information domain, requiring extensions to MaSE to address the development and use of the developed information model. The rest of this section discusses the type of ontologies used, the methods for building them, and the necessary additions and modifications to MaSE.

3.2.1 Structure of an Ontology

Chapters I and II outlined the various definitions of an ontology. One consequence of varying definitions is that the structure of an ontology can vary from application to application. To devise an overall structure for ontologies, this research devised a comprehensive ontology structure by performing a union on the capabilities of the various structures discussed in Chapter II. The resulting structure, represented by the UML model in Figure 12, contains the expressive power to represent any of the ontologies presented in ontological research. Each part of the structure is discussed in this section.

The *Ontology* object contains the metadata about the ontology. This data contains the name of the ontology, the designers, version number, the language the ontology was developed in, an identifier, and the description of the ontology. The description field is designed to explain the purpose for which the ontology was originally designed and any other information the designers feel that people should know about the ontology. The ontology is composed of a collection of classes (or objects) and axioms.

Axioms allow the ontology to describe characteristics of objects using first order logic. The *Axiom* object is an abstract class from which the various types of axioms inherit. The three types of axioms identified are: equivalence, disjoint, and covered. Equivalence is self-explanatory: it allows for the ontology to specify that two classes represent the same semantic idea in the ontology. This type of axiom occurs when two ontologies are imported to form a larger one. The two imported ontologies can use different objects to represent the same semantic ideas, so designers use equivalence axioms to illustrate the similar objects from the imported ontologies. Two classes are disjoint if a domain object cannot simultaneously belong to both classes. As an example, consider the *herbivore* and *carnivore* classified objects with a disjoint axiom between them. This indicates that an animal can be a herbivore or carnivore, but can not be both. The covered axiom allows for restrictions and other characteristics to be specified for an object using first order logic or a natural language. It consists of a list of statements (coverers) and the class that the statements restrict or describe. For example, a covered axiom for an *adult_elephant* class would be that the age attribute has to be between 5 and 8 years of age. Any elephant younger would be considered a *youth_elephant* and any elephant older would be a *senior_elephant*, and appropriate covered axioms would be defined for those objects. Designers use these axioms to specify additional information about the classes.

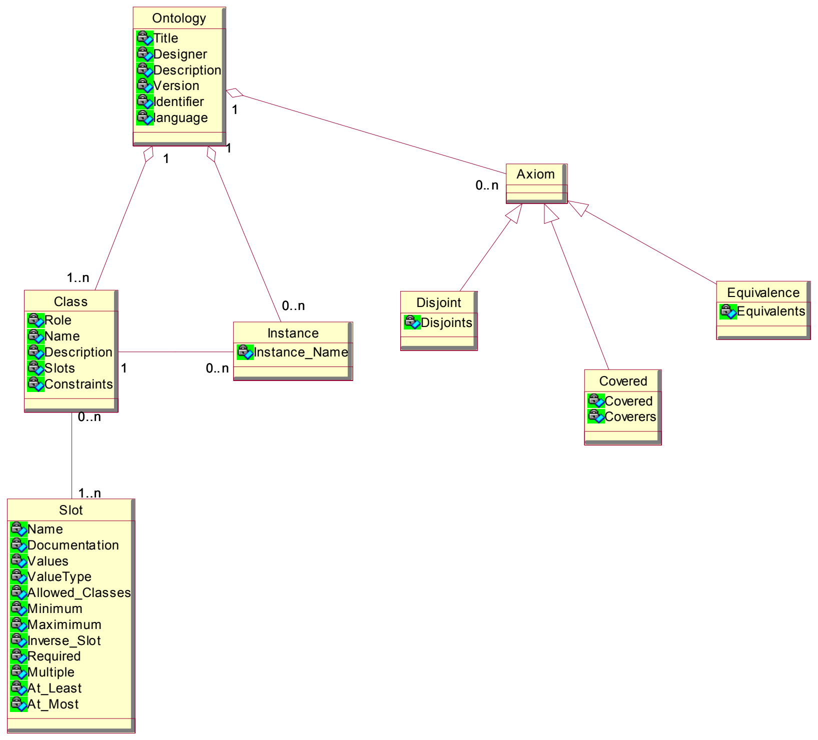


Figure 12 Structure of an Ontology

Classes are the main building blocks of an ontology. The structure shown in Figure 13 describes the structure of a *Class* object. A class can be abstract or concrete, which is the role of the class. An abstract class cannot be instantiated and is used to group related classes under one class that defines the similarity between the classes.

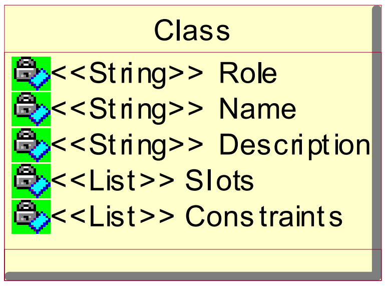


Figure 13 Structure of a Class in an Ontology

The name and description fields are used to uniquely describe the class and what it represents in the domain. The class has a collection of slots (or attributes) and a collection of constraints. Constraints define the axioms that apply to the class. Slots describe the various attributes of the class in the domain. These attributes represent properties, characteristics, and states of the object or relationships between classes in the domain.

The structure of a *Slot* is illustrated in Figure 14. Each slot has a name and description, similar to a class object in the ontology. The *ValueType* field defines the type of slot. The most common types of slots are: Number, Boolean, String, Instance and Enumerated. Enumerated slots have a list of allowed values. An example would be *Color* as a slot with the enumerated values red, blue, or green. These enumerations are saved as a list in the *Values* field for the slot. An instance slot shows relationships between classes in the ontology. An instance slot can have one or more specified classes from the ontology as values. The allowed values of an instance slot are saved in the *Allowed_Classes* field. An example of an instance slot is *has_Parent*. This slot belongs to a *Person* class and the *Allowed_Classes* field shows that the slot can point to any instance of a *Person*. The instance slot *has_Parent* illustrates that there is a relationship that can exist between two *Person* objects, where one class is the parent of the other. Other information such as the minimum and maximum number of classes that the slot can contain or the minimum and maximum value for an integer or float slot can be specified. The designers can also set the *Required* field to specify whether the slot is required for a class.

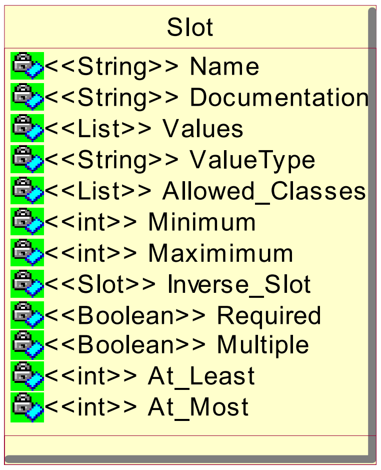


Figure 14 Structure of a Slot

The *Inverse_Slot* shows the inverse relationships in the ontology. For example, the slot *has_Parent* could be defined to specify the parent of a *Person* object. The inverse for this slot would be *has_Child* slot.

The classes, their axioms, and their instantiations all combine to form an ontology. A specification of an ontology contains these components and must be concise, complete, clear, and consistent.

3.2.1.1 Characteristics of an Ontology

The methodologies for building domain ontologies described in Chapter II contain an evaluation section that examines the ontology to ensure it is *concise*, *clear*, *consistent*, and *complete*. These characteristics ensure that the ontology is well defined for future reuse and comprehension. The terms *concise*, *clear*, and *consistent* represent the same concept in ontologies as in the requirements for domain representation discussed earlier.

To be *concise* requires that every piece of information in the ontology, such as names and definitions of classes, contain the least amount of information necessary for describing the domain. This characteristic is balanced by the necessity for the ontology to be complete.

All terms used in describing the ontology must be *clear*. The terms should be unambiguous and easily understood by others. This characteristic is key to reuse as future developers will not reuse ontologies that contain aspects that cannot be easily understood.

Another key to reuse is the *consistency* of the ontology. Consistent ontologies provide no contradictory or overlapping terms. This is particularly important for ontologies that are frequently used in knowledge-based systems that contain forms of machine reasoning. If reasoning over the ontology yields contradictory results, the system will not perform well.

To be *complete* requires that the ontology describe the domain to the level of granularity specified by the designers in the metadata description of the ontology. This definition does not mean that everything in the domain should be covered. For example, when constructing an ontology based on the domain of Air Traffic Control, the ontology should not describe an *Airplane* object as a collection of *Nuts*, *Bolts*, and *Aircraft_Parts*. Although those classes are the building blocks of an *Airplane*, they are not necessary to describe the domain in terms of scheduling and directing the airplanes in flight. An ontology is complete when it covers all objects in the detail necessary for the designer's purpose.

3.2.2 Methodology for Building System Ontologies

So designers can use ontologies to specify domain representations in multiagent systems, an appropriate methodology for developing ontologies must be specified. The existing methodologies for designing domain ontologies are built to describe *everything* about a specific domain. This is not appropriate for multiagent systems because one of the requirements is that designers only specify information required for proper system execution. The existing methodologies work for multiagent systems after slight modifications.

Reinventing the wheel, by developing a whole new methodology, does not make sense, because many years of research have gone into developing the domain ontology methodologies mentioned in Chapter II. Instead, this research extracts the main parts common to the methodologies. Thus, the important parts of the methodologies are included without the administrative overhead, such as tracking all possible classes or documenting every little step, as is the case in the IDEF5 method [KBSI 1994].

The resulting methodology is an iterative process that matches the iterative nature of multiagent system design. Methodologies such as MaSE allow designers to iterate through various steps, thus providing more flexibility than a sequential step-by-step methodology. Changes can be made to previous steps without having to start the whole process again.

Four main steps can be extracted from the existing methodologies for building domain ontologies. These steps are used as the foundation to create a methodology to construct ontologies for use in multiagent systems. The four steps are:

- Define Purpose and Scope of Project
- Collect and Analyze Data
- Construct Initial Ontology
- Refine and Validate Ontology

The remainder of this section is devoted to briefly discussing these steps and the general activities that fall into them. Chapter IV presents detailed descriptions of each step, along with directions, principles and hints for each phase of the ontology development process.

3.2.2.1 Define Purpose and Scope of Ontology

As discussed earlier, the metadata for an ontology should describe the purpose and scope of the project. This data aids future designers in understanding the ontology based on the goals for which it was developed. This phase describes the purpose of the ontology based on the system requirements and software engineering documents that are used to develop multiagent systems. For example, when building a multiagent system to play poker in a distributed network, the purpose of the constructed ontology is to *describe the objects in the poker domain*. The scope of the system is to *describe all objects necessary to permit the development of a distributed poker game*.

3.2.2.2 Collect and Analyze Data

This step involves analyzing data in the problem domain along with the system requirements to discover terms that may later become part of the ontology. There are many methods with which the designer can extract candidate terms for the ontology. Sample methods include: brainstorming, interviews with users, and reviewing project documents. The collected terms are analyzed to determine those that are needed for proper execution. Those not required are removed from the list. The remaining terms are used as possible classes, relations, and characteristics for the initial ontology. For the poker example, the list of possible terms includes: hand, cards, player, bet, fold, raise, call, money, and pot.

3.2.2.3 Construct Initial Ontology

The possible terms are organized into an ontology that describes the nature of the domain needed by the system to meet its requirements. The methodology provides hints to the designer to help determine proper classes, properties, and relationships based on the terms collected in the previous step. The construction step also addresses the use and integration of previously built ontologies. In the poker system, the initial ontology could define a player object that had money and a hand of cards along with the other necessary system classes.

3.2.2.4 Refine and Validate Ontology

This final step is an iterative process that occurs throughout the development of a multiagent system. The initial ontology can be validated through the creation of instances of the objects in the ontology to ensure the system can execute properly using the data. Throughout the multiagent system development process, refinements to the ontology may arise. Within this step, the designer can modify the ontology to meet added system requirements at any point in the multiagent system development. To validate the poker example, the designer would step through the use cases to ensure they can be completed using the data objects in the ontology. As the designer continues to build the system, changes to the ontology are made as necessary.

3.2.3 Methodology for Building Component Ontologies in Multiagent Systems

Along with developing the system ontologies, the designer must develop the component ontology. The component ontologies specify the data models of any agents that do not have the same data model as the system ontology. Developing the ontologies for the agent components is analogous to the development of the system ontology; the scope is just at a lower level. The only added step in creating component ontologies is to map the terms in the ontology to the corresponding terms in the system ontology. If the component ontology is the same as the system ontology, no mapping is required. If not, each component term should match to a term in the system ontology. This allows the developers to see how the designers want the systems to match up in terms of the system data.

3.2.4 Integrating the Construction of Ontologies into MaSE

To integrate ontological engineering into MaSE, this research introduces a new step in which the designer constructs the ontology for the system in the MaSE analysis phase. This research determined the step should occur after the creation of use cases. This placement allows the designer to use terms from the goal hierarchy, use cases, and sequence diagrams as possible concepts in the ontology and the resultant ontology can be used to create tasks in the refining roles step of MaSE. Tasks often indicate parameter passing, so the step is placed after the construction of the ontology to allow the designer to specify the type of the parameters as classes from the ontology. The extended MaSE diagram is shown in Figure 15.

This research considered placing the ontological construction before the use cases step to allow the actors in the sequence diagrams to pass objects from the ontology. This resulted in constructing the ontology before the use cases, as sequence diagrams and use cases are the same step in MaSE. It is preferable to build the ontology after the designer constructs the use cases, because possible terms for the ontology are extracted from the use cases. To place the ontology construction before the sequence diagrams but after the use cases would require the sequence diagrams to become their own step. The disruption of the original flow of MaSE, where use cases and sequence diagrams occur in the same step, outweighs the benefits of constructing the ontologies before the sequence diagrams.

The agent architecture step was modified to include the ability for designers to specify ontologies for the different agent components and to map them to the overall system ontology. The ranking model discussed in Section 3.3 assists the user with mapping the various ontologies. This research modified all steps that involve message passing to include specification of the types for the parameters passed in the messages. When combined with the additional step of constructing the system ontology, these extensions augment MaSE to include domain representation. The extended MaSE is presented in more detail in Chapter IV.

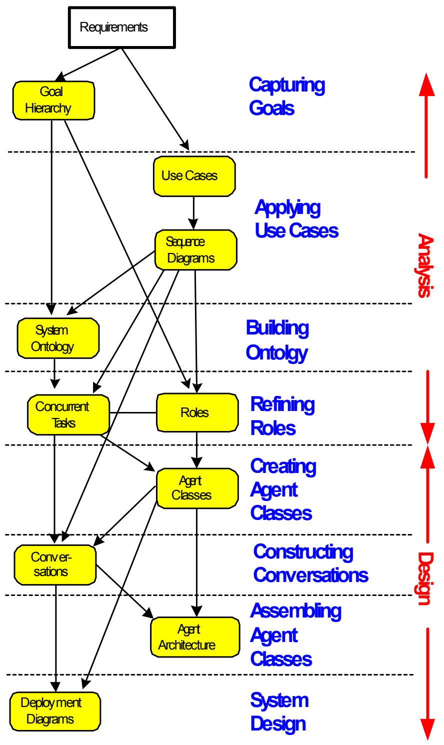


Figure 15 Extended MaSE Methodology

3.2.5 Alternatives to Ontologies

One alternative to using ontologies for domain representation in multiagent systems is the use of Unified Modeling Language (UML) data models. Although UML is a feasible alternative, it minimizes the benefit obtained by using the existing libraries of constructed ontologies and services for ontologies that currently exist in agent architectures, such as FIPA [FIPA TC B 2001]. In an open system environment, multiagent systems involve the interaction of agents that may not know anything about the domain representation of the other agents in the system. FIPA provides an ontology lookup service that allows agents to request the ontology used by other agents. The existing libraries and ontology lookup services allow for the agents to obtain knowledge about the domain representations and to allow them to communicate. The use of ontologies for domain specification in this research enables these benefits, while such services or libraries are not widespread for sharing UML models.

3.3 Geometric Score Reduction Model for Ranking Object Similarity

Mapping the component ontologies to the system ontology can be a monotonous and difficult task. As the number of objects in the ontologies increase, the user must map and distinguish between more objects. To assist the user with this endeavor, this research developed an Information Retrieval ranking model which ranks the objects in an component ontology based on their similarity to a specified object in the system ontology.

In Information Retrieval, a user needs a certain semantic content in a collection of documents. The ranking model of the information retrieval engine ranks the documents based on their similarity to the semantic content that the user is trying to find. When mapping an object from an ontology, the user looks for the object that represents the same semantic meaning in the target ontology. The Geometric Score Reduction Model developed in this research ranks the objects in the target ontology based on each object's probability of representing the same semantic concept as the selected object. The probability is set as the similarity score for the object and used to sort the objects from highest to lowest for presentation to the user.

3.3.1 Determining the Similarity Score of an Object

The ranking model uses the fact that the probability of two objects matching is the same as the probability that all characteristics of the objects match, represented by Equation 1. The term *match* will represent the fact that two items (objects, properties, or characteristics) represent the same semantic concept in a domain. This research considers the name, attributes and role of an object as appropriate characteristics to use when comparing objects. Thus, the equation for computing the probability of matching objects is transformed, as shown in Equation 2.

(1) $$P(Object_1 = Object_2) = P(Characteristic_{1,1} = Characteristic_{2,1} \cap \ldots \cap Characteristic_{1,i} = Characteristic_{2,i} \cap \ldots \cap Characteristic_{1,n} = Characteristic_{2,n})$$

(2) $$P(Object_1 = Object_2) = P(Name_1 = Name_2 \cap Attributes_1 = Attributes_2 \cap Role_1 = Role_2)$$

If matching characteristics are considered independent from one another, the probability of the objects matching is computed as the product of the probabilities that each characteristic matches. The characteristics are clearly not independent, because if the names of the objects match, there is more chance that the other characteristics will match, too; however, the correlation between the events will be constant among each object compared in the target ontology, so we can ignore the correlation altogether. Thus, the probability of two objects matching can be computed as shown in Equation 3, without having to compute the correlation of characteristics that is needed to compute Equation 1. Equation 4 shows the formula used by this research to compute the similarity of two objects based on their name, attributes and roles.

(3) $$P(Object_1 = Object_2) \approx P(Characteristic_{1,1} = Characteristic_{2,1}) * \ldots * P(Characteristic_{1,i} = Characteristic_{2,i}) * \ldots * P(Characteristic_{1,n} = Characteristic_{2,n})$$

(4) $$P(Object_1 = Object_2) \approx P(Name_1 = Name_2) * P(Attributes_1 = Attributes_2) * P(Role_1 = Role_2)$$

A number of factors must be considered when deciding the degree to which two characteristics match. Even if two objects match, that does not necessarily mean the characteristics represent the exact same semantic concept in the domain. For example, if the name *Smoke* was used by two objects, one object might represent the action of smoking while the other object might represent smoke from a fire. Similarly, if the characteristics do not match exactly, they can still represent the same concept. An example is an object *Class* and an object *Course* used by two ontologies to represent the same concept. When comparing the two names, the ranking model would not see an exact match, but the objects do match semantically. As a result, the similarity value of two characteristics that do not match is set to a value that represents how important that characteristic is in defining the actual semantic representation of the object. For example, because designers can develop ontologies in different languages, such as French or English, the name of an object does not have an extreme importance on the semantic content of the object. So, the similarity value for the name characteristic of two objects when the names are not exactly the same can be set to 40%, or some other number above zero.

Chapter V discusses the implementation of the ranking model, including a discussion on the characteristics of an object and why each one is or is not used in the ranking model. The ranking model considers the attribute structure of an object as the most defining aspect in distinguishing the semantic content of one object from another object. As such, the algorithms used to compare the attribute structure of the objects are discussed in Chapter V.

3.3.2 Evaluating Ranking Models

Traditional methods of evaluating information retrieval models use the metrics of precision and recall. A ranking model returns a set of documents, called the relevant set, from the collection of documents. Using this set, the analyst determines the recall as the number of documents, deemed relevant by an expert, that appear in the set divided by the total number of relevant documents. Recall, as shown in Equation 5, represents the percentage of documents accurately found by the ranking model. Precision represents how many non-relevant terms are included in the relevant set. The precision is the number of actual relevant documents in the set divided by the size of the relevant set, as shown in Equation 6.

$$(5) \qquad R = \frac{Numberof\,\mathrm{Re}levantDocuments\,\mathrm{Re}turned}{TotalNumberof\,\mathrm{Re}levantDoucments}$$

$$(6) \qquad P = \frac{Numberof\,\mathrm{Re}levantDocuments\,\mathrm{Re}turned}{TotalNumberofDocuments\,\mathrm{Re}turned}$$

When mapping between data models, however, these metrics are not appropriate. There is only one relevant object per query, so the recall is zero or one, based on whether or not the model returns the object. Since the ranking model returns all the objects from the system ontology, in order of their similarity, the recall will always be 100%. Precision would then be one divided by the number of objects in the system ontology.

This research uses the rank of the relevant object as the metric for evaluating the Geometric Score Reduction Model. This metric is meaningful in that it demonstrates the number of objects the designer must look at before finding the relevant object. For example, when searching for a match for a *Cat* object, the ranking model returns the sorted list: *Person*, *Car*, *Tuba*, *Feline*, and *Automobile*. The designer is looking for the relevant object of *Feline*, which is ranked fourth by the ranking model. By recording the rank as the metric, the R-Precision can be calculated as 25% while also showing that the user must look at four objects before finding the relevant one.

Time is another consideration when evaluating ranking models. For example, a ranking model might be 100% accurate, but take four days to execute. This is not acceptable for use when mapping between data models. As such, Chapter VI evaluates the Geometric Score Reduction Model based on the average rank of relevant objects and the time to rank the objects for three experiments.

IV. Extended Multiagent Systems Engineering

Extended MaSE is a complete methodology for building multiagent systems that guides a designer through software development by describing the structure, behavior, and data model of the system. This chapter describes the extended MaSE methodology in detail. The sections of MaSE not modified are summarized from [DeLoach and others 2001] to present the whole methodology.

agentTool automates the design of multiagent systems using the MaSE methodology. To support extended MaSE, this research augmented agentTool with an ontology builder and mapper. Also, the existing software now supports passing ontology objects between agents. The functionality was added while maintaining the original workflow model of agentTool. Following any modified MaSE step, this section discusses the changes made to agentTool to support the activities of that step.

4.1 Capturing Goals

The first step of MaSE captures the goals of the system, developing a structured set of goals from the initial system specification. This involves identifying the goals and then structuring them into a hierarchy. Goals are extracted from the system's functional requirements, which specify the services the system must provide and the actions the system should perform. MaSE uses goals as the basis of the analysis phase since goals are more stable than requirements that tend to change over time [Kendall and others 1998].

4.1.1 Identifying Goals

To capture the goals, the analyst must first extract them from the functional requirements. Goals represent what the system is trying to accomplish. The designer must modify the functional requirements from a *do this, do that* type statement to the overall essence of what the system is trying to accomplish. For example, in a distributed course scheduling system the designers determine the following goals:

- Produce Schedule

- Display Schedule
- Manage Existing Schedules

4.1.2 Structuring Goals

The goals are structured to show their sub-goal relationships with one another, based on the inter-relationships and importance of the goals. To aid user understanding, the Goal Hierarchy Diagram divides the goals into levels of detail and importance. Figure 16 is an example showing a partial Goal Hierarchy from the course-scheduling example.

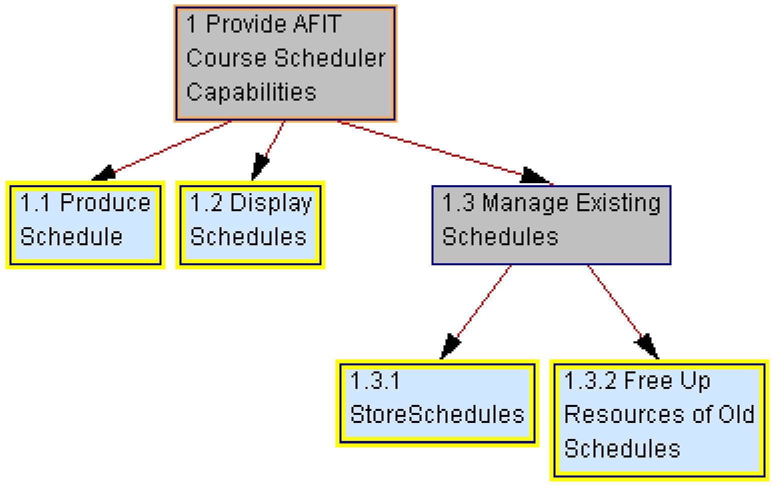

1 Provide AFIT Course Scheduler Capabilities
1.1 Produce Schedule
1.2 Display Schedules
1.3 Manage Existing Schedules
1.3.1 StoreSchedules
1.3.2 Free Up Resources of Old Schedules

Figure 16 Goal Hierarchy Diagram

4.2 Applying Use Cases

Once the goal hierarchy is complete, the analyst captures use cases from the system requirements and develops sequence diagrams to help identify an initial set of roles and communications paths in the system. The use cases define scenarios that the system must handle, and sequence diagrams represent the use cases as events between roles in the scenarios. The analyst will use these event sequences to define the tasks of the system roles.

4.2.1 Creating Use Cases

Use cases are examples of how the user thinks the system should work. The designer can develop use cases from the requirement specifications or by interviewing the system's users. This step identifies paths of communication, so the analyst should design use cases that cover varying event sequences, without repetition. The use cases should not describe every sequence of events the system must handle, but should show how the system accomplishes each goal in the hierarchy.

4.2.2 Creating Sequence Diagrams

A sequence diagram depicts the sequence of events from the scenarios described in the use cases. Roles are created, based on the use cases, and placed at the top of the diagram. Figure 17 shows an example sequence diagram, with the arrows between lines representing events passed between the roles. The order of events proceeds from the top to bottom, so to generate a schedule the *Scheduler* first passes a *retrieve_requirements* event to the *RequirementsManager*.

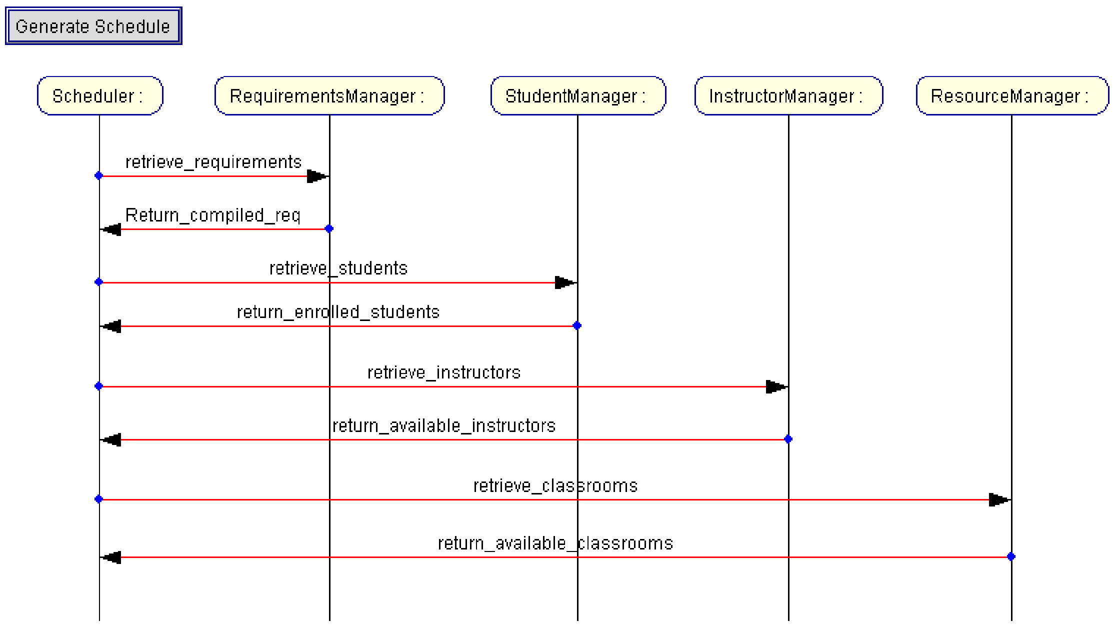


Figure 17 Example Sequence Diagram

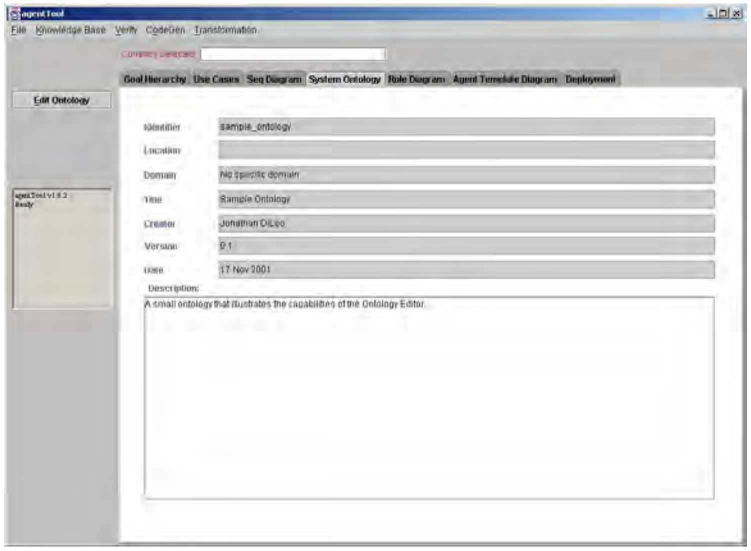
To transform the use cases into sequence diagrams, the analyst uses entities named in the use case as roles. Any communication or information passing in the use case becomes an event in the sequence diagram. This information passing also yields possible terms for the system ontology, developed next in the methodology.

4.3 Developing the System Ontology

This step represents the first addition made by this research to the original MaSE and uses concepts from the previous steps as a basis for constructing the ontology of the system. The designer first determines the purpose and scope of the ontology and then collects and analyzes data from the information domain for possible use in the ontology. Finally, the analyst constructs the initial ontology and refines, validates, and matures the model into a complete ontology.

To support the ontology creation steps of MaSE in agentTool, this research built an ontology builder program and integrated it into agentTool. The main window of agentTool contains tabs for each MaSE step. This research added a tab for the system ontology after the sequence diagram tab, so that the order would match the steps of the Extended MaSE methodology. The system ontology tab displays the metadata for the system data model and allows users to launch the Ontology Editor to modify or further view information about the ontology, as shown in Figure 18. With this approach, the user can view a quick summary of the data model and can receive more detailed information if interested.

The designer uses the ontology builder, shown in Figure 19, to view additional information or to edit the ontology. The Ontology Editor program design is based on Protégé 2000 to shorten the learning curve of users that are accustomed to Protégé 2000 [Noy and others 2001]. The main window consists of a tree view of the objects in the data model, organized by their inheritance, and three tabs: Classes, Axioms, and Metadata. Each tab contains the information regarding that section of the ontology. Designers create, modify and view classes, slots, and axioms in this program.



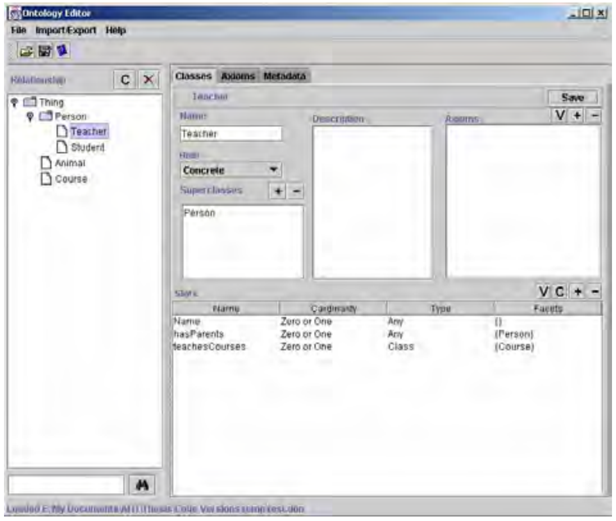
Identifier	sample_ontology
Location	
Domain	No specific domain
Title	Sample Ontology
Creator	Jonathan DiLeo
Version	0.1
Date	17 Nov 2001

Description:
A small ontology that illustrates the capabilities of the Ontology Editor.

Figure 18 Viewing System Ontology in agentTool

4.3.1 Define Purpose and Scope of Ontology

Designers specify the purpose, scope, and general information regarding the ontology in the metadata tab of the ontology builder. Figure 20 shows the metadata tab with an example ontology. Each ontology has a unique identifier and the location on disk or the URL where the ontology can be found. The metadata tab allows the user to describe the general purpose and description of the ontology and provide contact information.



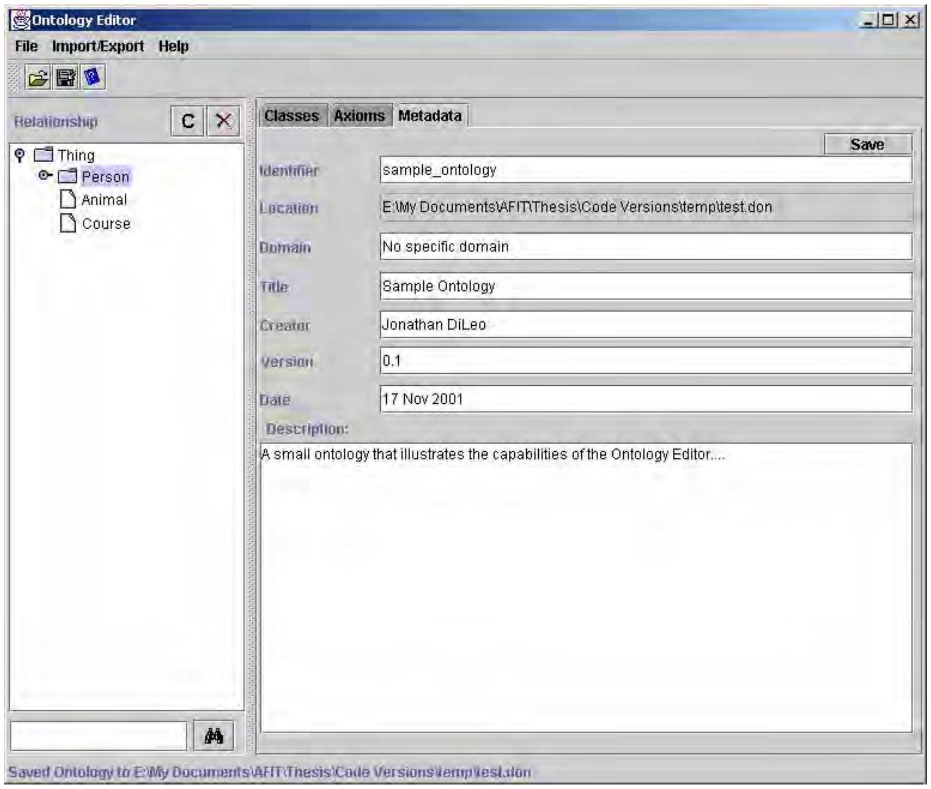
Name: Teacher
Role: Concrete
Superclasses: Person

Slots

Name	Cardinality	Type	Facets
Name	Zero or One	Any	()
hasParents	Zero or One	Any	(Person)
teachesCourses	Zero or One	Class	(Course)

Figure 19 Main Window of Ontology Editor

By describing the purpose of the ontology, the designer limits its scope. For example, when designing a multiagent system to perform course scheduling, the ontology must define classes regarding courses, quarters, instructors, classrooms, etc. The software requirements and the goal hierarchy help define the purpose of the ontology, as the purpose of the ontology is to fulfill the information needs of the multiagent system. The purpose describes why the ontology exists, such *as to list all classes in the education domain required when scheduling courses*. This description of the purpose determines the scope and the domain that the ontology will reside in. The scope defines the level of detail that the ontology describes the objects, such as *defining only the semantic ideas necessary to schedule courses in a distributed network environment*.



Identifier: sample_ontology
Location: E:\My Documents\AFIT\Thesis\Code Versions\temp\test.don
Domain: No specific domain
Title: Sample Ontology
Creator: Jonathan DiLeo
Version: 0.1
Date: 17 Nov 2001
Description:
A small ontology that illustrates the capabilities of the Ontology Editor....

Saved Ontology to E:\My Documents\AFIT\Thesis\Code Versions\temp\test.don

Figure 20 Metadata Tab in Ontology Editor

To further define the scope, the designer can utilize the previously identified use cases to determine the types of data that the system will use. For example, a use case may describe one agent passing another agent a specific course to schedule. The designer uses this situation to determine the level of detail necessary to describe a course so that the system can execute the events described in the use case appropriately.

4.3.2 Collect Data

Having defined the scope, the analyst knows the level of detail and domain the ontology represents and can start building the model. The designer first creates a list of possible terms that the ontology must contain. Designers form this list by examining the goal hierarchy, use cases, and sequence diagrams from the previous MaSE steps. In Figure 17, for example, the *Scheduler* role requests instructors, students, and resources in the system execution. From this diagram, the designer knows that the ontology must include concepts to represent these items and adds their names to the list as possible terms for the ontology.

4.3.3 Construct Initial Ontology

This step takes the list of concepts and organizes them into classes and attributes and produces an initial draft of the data model. When creating the ontology, the analyst must remember to only specify the concepts that the system needs to accomplish its goals. The ontology should not specify all attributes of a *Human*, such as *height*, *age* and *weight*, when the system only requires the name of a *Human* to function.

4.3.3.1 Reusing Existing Ontologies

The designer must first decide whether any existing ontologies will meet the system needs. The user reviews ontology libraries and existing company data models looking for objects that resemble the concepts listed in the term list built earlier. The benefit to using an existing ontology is that the system is interoperable, in terms of passing data, with any other system that uses the same data model. If no existing ontologies fully specify the information needed for the system, the designer must build a new ontology. If the designer finds an ontology that partially satisfies the system needs, that ontology can be used as a starting point for the new ontology. Users should post created models in some shared repository so that others can reuse the data model, increasing the interoperability of future systems.

4.3.3.2 Build Class Hierarchy

The first step in building an ontology from scratch creates classes from selected terms in the term list, created in the collect data step, and organizes the classes into a hierarchy. The hierarchy is based on subclasses and every class is a subclass of *Thing*.

Analysts can build the hierarchy using a top-down, a bottom-up, or a middle-out approach. The designers start by selecting terms from the list that are independent objects. In other words, they do not describe other objects. The user must ask if the term represents a characteristic of another term or if the term is described using other terms as its characteristics. For example, an *Animal* is an object while *Age* is an attribute of the object. In a middle-out approach, the designer takes the terms and selects those thought to occur in the middle of the hierarchy, allowing the designer to increase detail while creating subclasses and abstract details while creating parent classes. For example, to represent the similar attributes in an *Instructor* and *Student* class, the analyst can create a *Person* class as a parent class of both. The designer can also further specify the *Student* class by creating a *Part-time Student* and *Full-time Student* subclasses. The hierarchy is complete once the designer identifies and structures all the objects needed by the system.

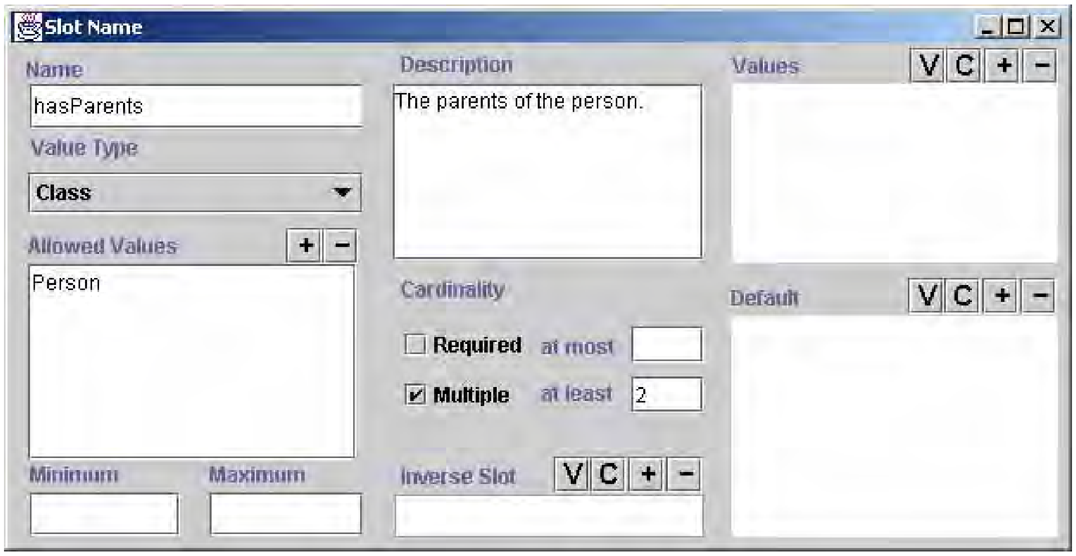
The main window of the Ontology Editor allows for the creation of classes. The user can then specify information about the class by determining the role of the class, adding or removing inherited classes, and providing a description of the object.

The *Teacher* class is displayed in Figure 19. This is a concrete object with three attributes: *Name*, *hasParents*, and *teachesCourses*. There are no axioms regarding this class in the ontology, since none are displayed in the axiom list.

4.3.3.3 Add Attributes to Classes

The analyst now defines the attributes (also known as slots) of the classes identified in the previous step. The attributes should describe the properties of the class at the level of detail required by the system to accomplish its goals. Each attribute is described in terms of its data value, name, description, and cardinality. For example, a *Person* class has the attribute *age*. The cardinality of the *age* slot is one and is required for every *Person* object.

Designers use the Modify Slot window, shown in Figure 21, to view and modify the characteristics of an attribute in agentTool. The user inputs the name and description of the attribute and chooses the data type from the drop-down box.


Slot Name

Name: hasParents

Value Type: Class

Allowed Values: Person

Description: The parents of the person.

Cardinality: [ ] Required at most ; [✓] Multiple at least 2

Figure 21 Add/Modify Slot Window

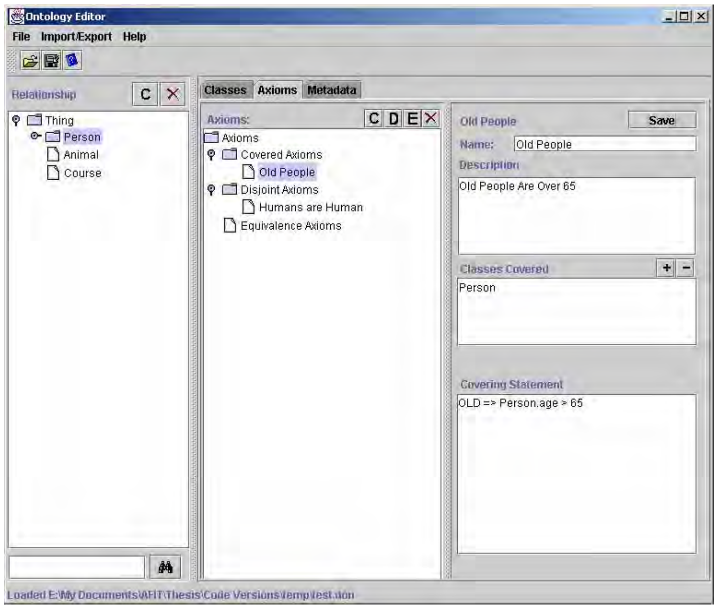
The allowed values list displays the possible classes that can fill this slot. If the data type is a string, integer, or float, this list is not displayed. The user also specifies whether the slot is required and if there can be more than one instance to fill this slot. In Figure 21, the slot *hasParents* is defined as a multiple slot that has at least two instances. Users can add the slot's inverse, such as *hasChildren*, if appropriate. The *Values* and *Default* lists are not implemented at this time. With the values list, users can specify instances of a class and with the default list the user can specify the default value of the slot. This implementation does not support instances, as designers rarely hard code specific instances into the data model. The lists are left in the window for future development, if necessary.

4.3.3.4 Define Relationships

The attributes of a class define the properties and relationships of the class. This step encodes all necessary relationships between classes as attributes of the class. For example, a *Person* owns a *Car*. The designer can represent this relationship with an attribute of the *Person hasCar*, which is of type *Car*. If the system needs to know the inverse of the relationship, the analyst uses an inverse slot. The *Car* class would contain the slot *hasOwner*, with type *Person*, as the inverse slot of *hasCar.*

4.3.3.5 Define Axioms

Once classes and their attributes have been defined, the designer specifies the domain axioms. As discussed in Chapter III, axioms can specify restrictions on the classes and attributes. If the system requires any restrictions on the data that cannot be represented by the attribute characteristics, the designer must develop the appropriate axioms. Using a banking system as an example, an *Account* class will have an attribute *balance*. The value of this field can never be less than zero. To represent this in the ontology, the designer constructs an axiom *BalancesMustBeGreaterThanZero*, which states that *balance* of the *Account* must be greater than or equal to zero at all times.



Name: Old People

Description: Old People Are Over 65

Classes Covered: Person

Covering Statement: OLD => Person.age > 65

Loaded E:\My Documents\AFIT\Thesis\Code Versions\temp\test.ion

Figure 22 Axiom Tab in Ontology Editor

The axioms tab, shown in Figure 22, allows for the creation and deletion of axioms in the Ontology Editor. The program organizes the axioms by their type: Equivalence, Covered, or Disjoint. Users can view selected axioms in the right panel of the window. Figure 22 shows the axiom that defines when a person is considered old. The axiom concerns the *Person* class and contains a short description and the statement that a person is old if the value of the age slot is greater than 65.

4.3.4 Refine and Validate Ontology

Once the designer defines the classes, attributes, and axioms of the ontology, he/she must validate that the ontology meets the system requirements. Any missing information is added to the ontology, and any extraneous information is removed from the ontology. If any information is incorrectly specified, the designer makes the necessary corrections to the ontology. To validate the model, the analyst examines the situations described in the use cases and sequence diagrams to ensure that the ontology describes all the information needed in those scenarios.

This step is repeated throughout the development of the system. If at any time the designer realizes that some piece of information is missing from the ontology, the ontology is modified to include this information. The ontological construction is complete once the analyst is satisfied that the ontology represents all the necessary information from the sequence diagrams and use cases.

4.4 Refining Roles

The last step in the analysis phase of MaSE, refining roles, transforms the goals and sequence diagrams into roles and tasks. Roles form the foundation for agent classes in the Design phase, and each role is assigned at least one goal from the Goal Hierarchy Diagram. The analyst can combine related goals into a single role for efficiency, if desired.

4.4.1 Creating Roles

The analyst must take the initial set of roles developed from the sequence diagrams and assign goals to them from the goal hierarchy. Then, the designer adds roles for each goal that does not have a role assigned to it. In this manner, the analyst ensures proper system execution by assigning every system goal to a role. Once every goal is assigned to a role, the analyst must look at how that goal is accomplished in the system. If a role exists that does not have a goal, it is either superfluous and can be removed or a goal is missing from the hierarchy. The analyst must decide which is the correct action to take.

4.4.2 Concurrent Tasks

Once roles have been created, the analyst designs tasks to describe the behavior necessary for the role to accomplish its goals. An example of roles and their tasks is shown in Figure 23. The roles are represented in the boxes along with the numbers of the goals assigned to it. Tasks are shown in ovals attached to the role they belong to. The arrows show the flow of communication between the tasks. Each task is a single thread of control and is represented graphically using a Concurrent Task Diagram, an example of which is shown in Figure 24.

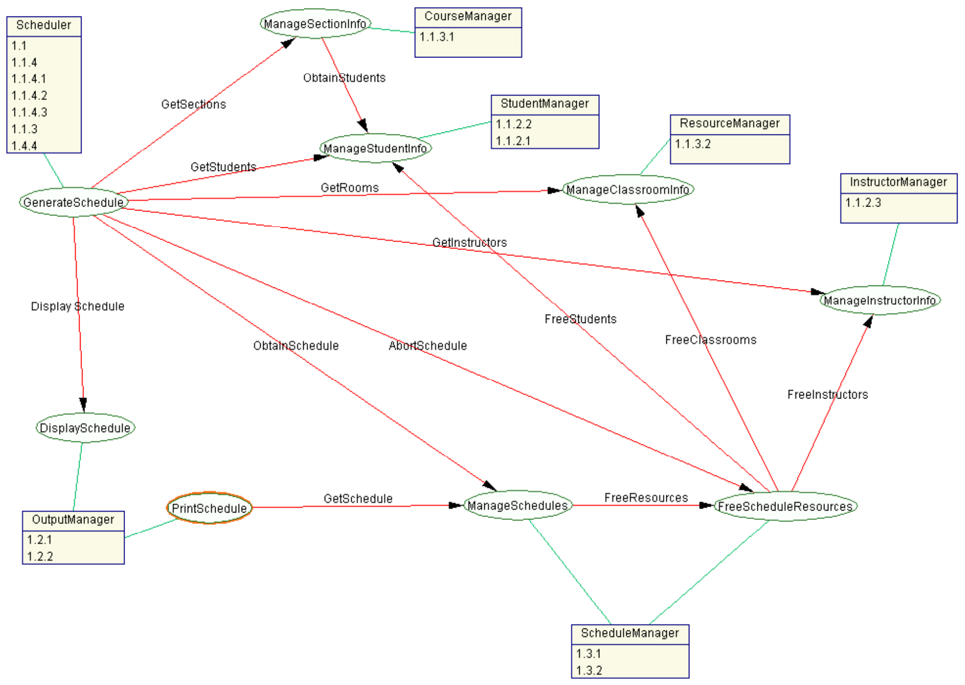


Figure 23 Example Role Diagram

The Concurrent Task Diagram is a finite state automaton representing the role's behavior throughout the task. The transition between states uses the syntax *trigger [guard] ^ transmission(s)*. The *trigger* portion specifies what event can allow the transition to occur. If a *guard* is specified, not only must the *trigger* event occur, but the *guard* statement must also evaluate to true before a state transition occurs. Upon the transition, the event *transmission(s)* will occur. This event frequently involves sending messages to other tasks. In Figure 24, the reception of an abort message from another role enables the first transition. The abort message contains a single parameter named *schedule*. Previously, MaSE required only the name of the parameter. Now with the inclusion of the information domain, each parameter lists the name and data type. The analyst uses the system ontology to specify the types of the parameters. Upon receiving the abort message, the role will send a message to *resourceManager*. The message will have the message *unscheduleClassrooms* and it includes the schedule that the system is erasing.

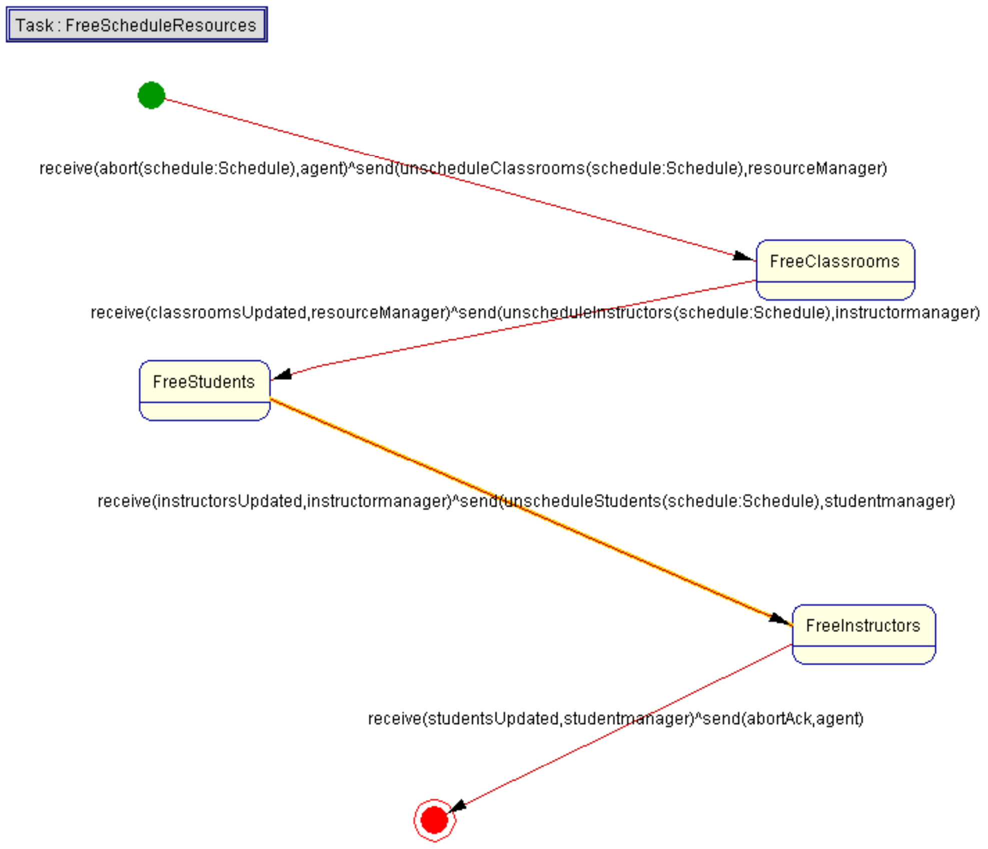

Task : FreeScheduleResources
receive(abort(schedule:Schedule),agent)^send(unscheduleClassrooms(schedule:Schedule),resourceManager)
FreeClassrooms
receive(classroomsUpdated,resourceManager)^send(unscheduleInstructors(schedule:Schedule),instructormanager)
FreeStudents
receive(instructorsUpdated,instructormanager)^send(unscheduleStudents(schedule:Schedule),studentmanager)
FreeInstructors
receive(studentsUpdated,studentmanager)^send(abortAck,agent)

Figure 24 Example Concurrent Task Diagram

The analyst can use sequence diagrams as an aid in constructing concurrent task diagrams. Sequence diagrams describe the minimal set of messages necessary to complete the various scenarios. The analyst can take the messages received by the role in the sequence diagrams and ensure they are in a concurrent task diagram for that role to accomplish the scenarios. The Analysis phase is complete once concurrent task diagrams have been provided to describe how each role will meet its goal.

4.5 Creating Agent Classes

The first step in the Design phase creates agent classes from the roles in the Analysis phase. This step defines agent classes in terms of the roles they will play and their conversations. To ensure the system functions appropriately, each role must be assigned to at least one agent class. Once roles are assigned to agent classes, the analyst can determine the conversations between agent classes by consulting the Role Model diagram.

The Role Model diagram, as shown in Figure 23, defines the tasks of each role and the communication between tasks. These communications must be assigned to the corresponding agents. If two agents contain tasks that communicate in the Role Model diagram, there must be a conversation between the two agents. Figure 25 is an example Agent Class Diagram. Each box contains the name of the agent and the roles of the agent. In Figure 25, the *UserAgent* fulfills the roles *Scheduler* and *OutputManager*. From the Role diagram in Figure 23, the communication between the *GenerateSchedule* task and the *FreeScheduleResources* task is represented as the conversation *abortSchedule* in the Agent Class Diagram between the agents that play the roles *Scheduler* and *ScheduleManager*.

Following through this transformation from the task communication in the role diagrams, the analyst can identify the necessary communication between agents. Once the Agent Class Diagram is complete, the designer has identified the agents and the conversations in the system.

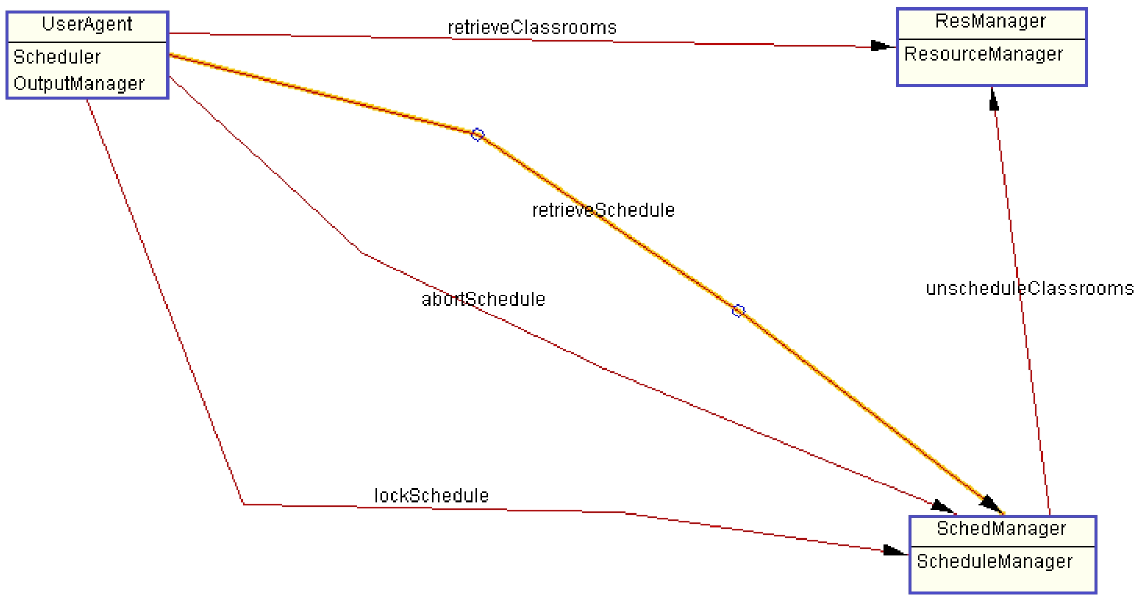


Figure 25 Example Agent Class Diagram

4.6 Constructing Conversations

Now that the conversations are identified, the analyst must specify the behavior of each conversation. This step describes the details of all system conversations in terms of a finite state automaton. Each conversation consists of two Communication Class Diagrams, one for the initiator and one for the responder of the conversation. The diagrams describe the communication states of the two agent classes participating in the conversation. The syntax for a transition in the Communication Class Diagram is:

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

A transition is enabled when the agent receives the message *rec-mess* with the parameters *args1* and the guard condition *cond* holds. Similar to the transitions in the Task Diagrams, this research modifies MaSE so that the parameters for messages have the syntax *name*:*Type*, where *name* is the name of the variable and *Type* is the class type from the system ontology. When the transition is executed, the agent will perform the specified *action* and transmit the message *trans-mess* with the arguments *args2*. Every field in a transition is optional.

Figure 26 is an example of the Communication Class Diagram for the initiator of the *retrieveSchedule* conversation. The agent sends the message *retrieveSchedule* to request the schedule for a specific course type and then waits for the schedule to be returned by the other agent. This response comes in the form of the message *returnSchedule*, with the requested schedule as the parameter *sched*.

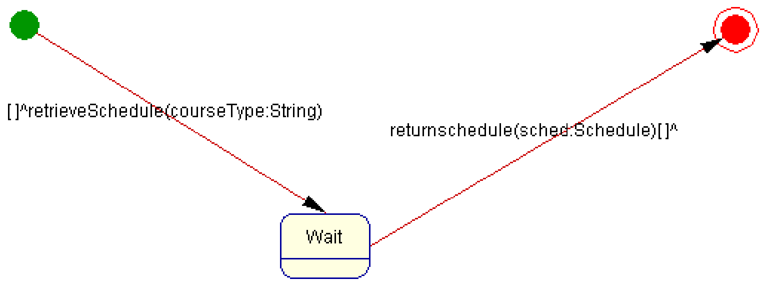

[ ]^retrieveSchedule(courseType:String)
returnschedule(sched:Schedule)[ ]^
Wait

Figure 26 retrieveSchedule Conversation Initiator

Figure 27 shows the responder Communications Class Diagram for the same conversation. The conversation starts when the agent receives a request to retrieve the schedule. The agent then returns the schedule to the requesting agent.

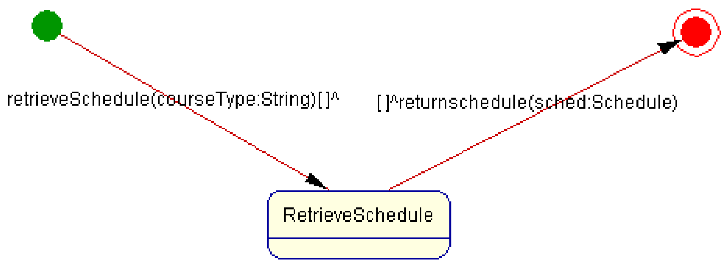

retrieveSchedule(courseType:String)[ ]^
[ ]^returnschedule(sched:Schedule)
RetrieveSchedule

Figure 27 retrieveSchedule Conversation Responder

Designers construct conversations based on the tasks developed in the Concurrent Task Model. Each task that defines external communication will yield at least one conversation. The analyst can trace through the task model to ensure that messages sent and received in the task model correspond to those in a conversation.

4.7 Assembling Agent Classes

This step adds the internal details to the agent classes identified in the Creating Agent Classes step. Designers can choose an existing architecture, such as Belief-Desire-Intention [Georgeff and others 1996], or can develop their own. Once the architecture is built, the designer specifies the components of each agent in the Agent Architecture Diagram.

Figure 28 is the Agent Architecture Diagram from the *UserAgent* class in the example. Four agent components combine to form this agent class. Each box represents one component and contains a list of the attributes and methods of the component. The lines represent internal communication, such as method calls, between the various components. In Figure 28, the *GenerateSchedule* component must communicate with the *DisplaySchedule* component to display the schedule once it is complete.

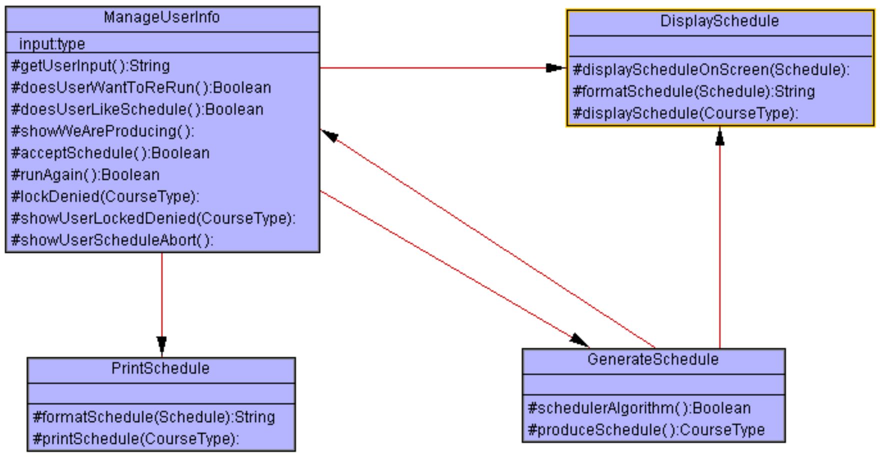

ManageUserInfo
input:type
#getUserInput():String
#doesUserWantToReRun():Boolean
#doesUserLikeSchedule():Boolean
#showWeAreProducing():
#acceptSchedule():Boolean
#runAgain():Boolean
#lockDenied(CourseType):
#showUserLockedDenied(CourseType):
#showUserScheduleAbort():
DisplaySchedule
#displayScheduleOnScreen(Schedule):
#formatSchedule(Schedule):String
#displaySchedule(CourseType):
PrintSchedule
#formatSchedule(Schedule):String
#printSchedule(CourseType):
GenerateSchedule
#schedulerAlgorithm():Boolean
#produceSchedule():CourseType

Figure 28 Example Agent Architecture Diagram

Designers can describe the internal behavior of the components with state-diagrams and formal definitions for the operations. Complex components may even have sub-components. One typical agent architecture defines a component for each task the agent performs. In this manner, the behavior of the agent is compartmentalized by tasks.

4.7.1 Special Case -- Specifying Agent Component Ontologies

Agent components will typically use the system ontology defined in Step 3, however, there are cases when the components will use a separate ontology. This research extended MaSE to accommodate these occasions. The first case occurs when the system interacts with a legacy system, such as an existing database. The multiagent system data model frequently will not match the legacy system's model. The agent component that interfaces with the legacy system must use the ontology of the legacy system. The designer can then provide a mapping from the legacy system's model to that of the multiagent system, with the component responsible for translating between the two models.

The second case occurs when an agent component is reused from a previous system. The reuse of agent components is similar to Component Based Software Engineering. Designers can build components to provide frequently needed services, which can then be built into future systems. Each component is designed with its own ontology that is then integrated into the systems. The developer must map the component data model to the system model to ensure proper performance. The data model for reusable components should be much smaller than the system data model. As discussed in Chapter III, a small ontology aids reuse because the domain representation acts as a type of precondition for the proper execution of the system. With a smaller ontology, there are fewer conditions that future system must obey. Only the information required by the component is specified in its data model.

Designers create component ontologies in the same manner as system ontologies. Using techniques discussed in Section 4.3, the analyst can fully specify the component ontology.

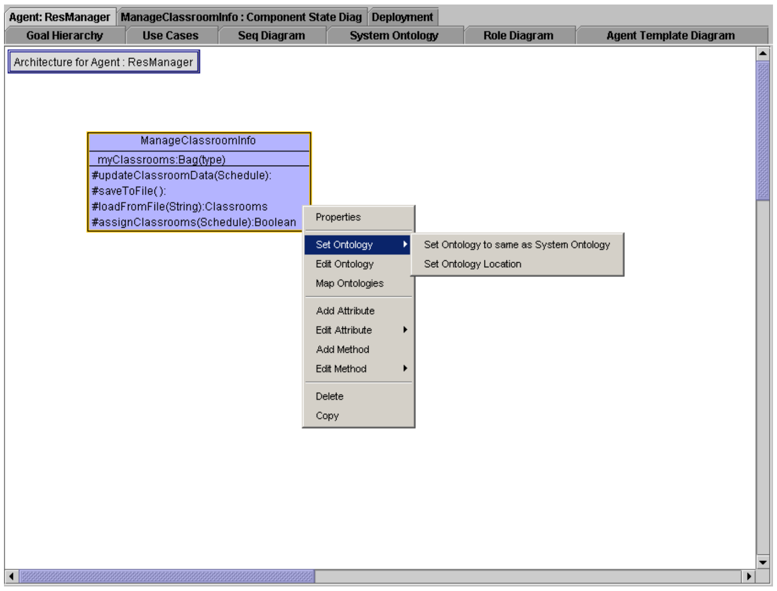
4.7.1.1 Mapping Component Ontologies to the System Ontology

The analyst must provide a mapping for any component ontology that does not match the system's ontology. During implementation, the programmers use these mappings to create translators, or other similar programs, to convert the data between the models. In distributed systems, this is frequently called marshalling the data. Each attribute and class in the component ontology should be paired to an attribute or class in the system data model. This ensures the multiagent system will convert data appropriately when passed between the agents.

Component ontologies are mapped to the system ontology to reduce the number of mappings. An alternative is to map the data model of each component to every component it interacts with. The problem with this approach is it increases the number of mappings required. If each component interacted with every other component, this could result in *n*! mappings for *n* agent components. Instead, the system data model is used as the universal language for the components to reduce the number of mappings in the system.

4.7.1.2 Component Ontologies in agentTool

In MaSE, agent components may have different data models than the system. To support this capability, this research augmented the Agent Architecture Panel in agentTool to allow the user to add, delete and modify component ontologies, as shown in Figure 29.



```
ManageClassroomInfo
myClassrooms:Bag(type)
#updateClassroomData(Schedule):
#saveToFile():
#loadFromFile(String):Classrooms
#assignClassrooms(Schedule):Boolean
```

Figure 29 Editing Component Ontologies in agentTool

Designers can set the component ontology to the system ontology or to another ontology. The developer can also edit the component ontology directly from this panel with the appropriate menu selection. When adding attributes to the components, users may choose from the object types in the specified ontology. If the component ontology is different than the system ontology, the user must map the component ontology to the system ontology.

4.7.1.2.1 Mapping Ontologies

Designers map the ontologies in agentTool through the Ontology Mapper program. This program displays the component ontology in a tree structure on the left side of the panel, as shown in Figure 30. The mapper displays the slots of the selected object in the list below the ontology. When the user selects a slot in the component ontology, the program ranks the objects in the system ontology based on the ranking model discussed in Chapter V. The mapper program then lists the ranked objects in the suggested class mapping list, as shown in Figure 30.

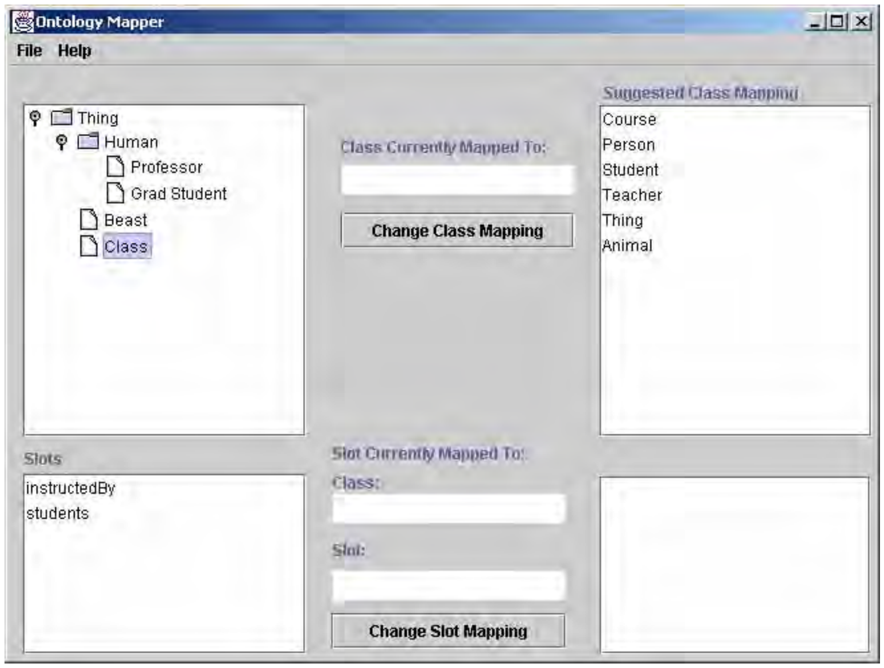


Figure 30 Ontology Mapper

In the center of the window, the mapper program displays the existing mapping for the selected object. If the user selects any slots from the object, the existing mapping for that slot is displayed in the lower middle of the window. The program displays the class and slot the attribute is mapped to, as slots from the same object may be mapped to different classes in the system ontology. This occurs when the designer represents an object in the component ontology as two or more objects in the system data model. The user can modify the mappings of objects and slots by selecting the class and slot desired in the system ontology and pressing the appropriate button. The program suggests candidate objects based on the similarity values returned by the ranking model.

4.8 System Design

The final step in the MaSE methodology instantiates actual agents from the agent classes defined in earlier steps. The Deployment Diagram shows the numbers, types and locations of the agents in the system, as shown in Figure 31. The shadowed boxes represent the agent instances and describe the agent name and type. The dashed lines in the diagram represent a single computing platform. In our example, none of the agents execute on the same platform. The *SchedManager* and *ResManager* reside on dedicated servers and the *UserAgents* can be on multiple machines. The solid lines represent the communication between the agents and platforms.

Designers use the Deployment Diagram to describe the location of agents so that information such as the hostname or network address of the platforms can be specified for implementation. The Deployment Diagram allows for multiple configurations of the multiagent system, giving designers the flexibility to adapt to various sets of resources. For example, if network bandwidth is limited, the designer should place as many agents as possible on the same platform to reduce network communication. This must balance with the fact that each agent requires a different amount of processing power and the designer must not overload individual platforms. Using the Deployment Diagram, the analyst can balance network communication with processing limitations of individual platforms. Upon completion of this step, the user is finished with the multiagent system design, having analyzed and designed the behavioral, structural and information models of the system. The development team must now code the system based on the design documents.

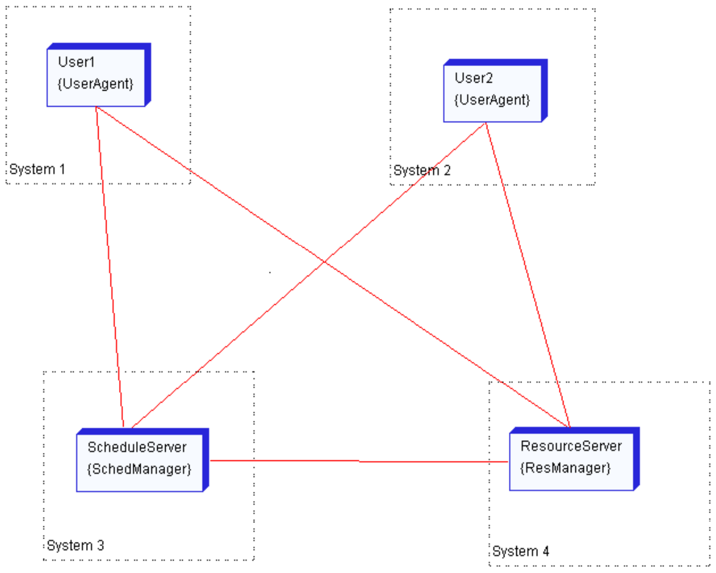


Figure 31 Example Deployment Diagram

4.9 Code Generation

agentTool generates Java code for the multiagent system based on the design documents created from the MaSE steps. This research modified the code generator so that agentTool produces Java code for every class in the ontology. These classes are then used by the agent code to pass messages in the system, as described in the MaSE design. The resultant code provides a shell for the agents consistent with the developed data, structural and behavioral models.

V. Geometric Score Reduction Model

5.1 Computing the Similarity of Objects

As discussed in Chapter III, this research uses the characteristics of the objects to compute the similarity of one object to another. A significant part of developing the ranking model is to determine which characteristics should be used. This research bases the decision of whether to include a characteristic in the similarity equation based on how significant the characteristic is in defining the semantic content of the object. This section discusses the inclusion of the characteristics of an object, as shown in Table 1, in the similarity equation for comparing objects.

Characteristic	**Included or Not Included in Ranking Model**
Name	Included
Attribute Structure	Included
Role (Concrete or Abstract)	Included
Number of Children	Not Included
Number of Axioms	Not Included
Number of Parents	Not Included
Level in the Object Hierarchy	Not Included
Description	Not Included

Table 1 Object Characteristics and Their Use in the Geometric Score Reduction Ranking Model

5.1.1 Object Characteristics Used by the Model

The similarity score of an object is calculated using Equation 7. The ranking model uses the characteristics in the equation to rank the objects in the most precise manner possible. To obtain a precise ranking, the characteristics chosen for the equation must have a significant impact on the underlying semantic content the object represents. The attribute structure, name, and role characteristics of an object all meet this requirement.

$$P(Object_1 = Object_2) \approx P(Attributes_1 = Attributes_2) * P(Role_1 = Role_2) * P(Name_1 = Name_2) \tag{7}$$

Attribute Structure. This research considers the attribute structure the most distinguishing characteristic of an object. In other words, if the attributes of two objects match, there is a good chance the objects represent the same content. There are cases when one designer may choose to include more attributes of the object than the other designer. In this case, the ranking model will identify the attributes that do match and will penalize the similarity score because not all objects match. Using the attribute structure, either in a best-value or comparator method, the ranking model can obtain an effective similarity score. Section 5.2 discusses these two methods of attribute comparison in greater detail.

Role. To further differentiate between objects, the ranking model compares the roles of the object. As discussed in Chapter III, the role of an object signifies if the class is abstract or concrete. If one object is concrete and the other abstract, the ranking model reduces the similarity score. Including this characteristic introduces the assumption that designers will design similar objects with the same role. This is an appropriate assumption under most circumstances. Because an abstract class cannot be instantiated, designers will represent abstract concepts as abstract classes. So, if the roles of two objects do not match, they are not likely to match semantically.

Name. The ranking model compares the names of the objects to differentiate between objects that match in attribute structure and role characteristics but do not represent the same semantic idea. For example, a class *Person* contains an *Age* attribute and a class *Dog* contains an *Age* attribute. Both classes are concrete, and without comparing the names, the ranking model would indicate that the objects are similar. So the ranking model reduces the similarity score when the names of the objects do not match. The fact that the names of two objects do not match does not mean that the objects do not represent the same content, so the penalty for names not matching should not be large.

Currently, the ranking model performs a string comparison to determine if the names match. A future enhancement could include a thesaurus to look for similar words that match based on a lookup in the thesaurus.

5.1.2 Object Characteristics Not Used

This research designs the ranking model for mapping from one data model to a larger data model. The ranking algorithm works for data models with a similar number of objects, but the algorithm does not include characteristics that might improve the precision of the model when used on similar sized ontologies. The characteristics not used in the similarity equation are either inappropriate when mapping from a data model to a larger one or are too computationally intensive when evaluating the possibility of a match. Each characteristic is described below along with why the ranking model does not use it in the computation of the similarity equation.

Number of Children. The number of objects that inherit from the object, also known as the number of children of the object, would be an appropriate characteristic to compare if the ontologies were the same size. In the case of system design, where the system data model will typically contain more objects than the component data model, this characteristic is not appropriate. For example, the component data model might contain an object *Airplane* while the system data model has an object *Aircraft*. *Aircraft* has *Fighter Aircraft* and *Support Aircraft* as children, while *Airplane* has no children. This can happen when the component is a take-off manager component that only cares that the object is an aircraft in order to handle it taking off. Because the system data model will frequently contain more detailed information than the component ontology, the number of child objects of a class is inappropriate for computing the similarity function in the ranking model.

Number of Parents. The number of objects that the compared objects inherit from, or number of parents of the objects, is similarly not an adequate characteristic for inclusion in the similarity equation. The system data model will contain more objects, and as the number of objects increases, the probability of additional parents for the objects in the system data model increases. Also, since most objects will normally only have one parent, this characteristic will rarely not match the same characteristic of another object. As such, the inclusion of the number of parents into the similarity equation would rarely influence the results of the model.

Number of Axioms. Comparing the axioms that pertain to the selected objects could improve the precision of the results from the ranking model, but the comparison is a difficult task. If all axioms are in first order logic, the ranking model could determine if the axioms regarding each object are contradictory. If so, the objects are not similar and the similarity score can be set to zero. The ranking model does not perform this check due to the complexity of the logic inferences, such as keeping track of all the possible assignments between which variables might correspond to the variables in the other axioms. This step alone could take as long as the computation of all the other characteristics combined. As such, it is left as a possible addition in the future.

Description. The description of each object is difficult to compare. Designers will not write the descriptions exactly, so a simple string matching does not work. One of the most accurate comparisons treats the description fields as two documents and computes the similarity of the two. The ranking model would use a thesaurus to ensure similar words are included. The model would add synonyms of words in the descriptions and then compare the two. The number of words contained in both divided by the total number of words is the similarity of the description fields. This additional computation will have minimal impact on results, however, as designers frequently leave this field blank due to their impression that the semantics of the name and attributes of the object fully describe the object. This creates problems when comparing a data model that has descriptions with one that may or may not have descriptions as objects that match might be penalized if one of them does not contain a description. The ranking model does not use the description field for all of these reasons.

5.2 Comparing the Attributes of the Objects

As discussed in the previous section, the ranking algorithm uses the attribute structure as part of the similarity score between two objects. To compute the level of similarity in the attribute structure, the algorithm must compare the attributes in some manner. When comparing the attribute structure, the ideal outcome would determine the optimum matching of attributes that maximize the similarity score, finding a set of attribute pairs where each pair consists of an attribute from each of the objects and every attribute appears in exactly one pair. The similarity value of each pair is computed using the ranking model discussed in Section 5.2.1.1. The optimal set would maximize the sum of the similarity values of the pairs.

Finding the optimal matching is a combinatorial problem that takes a non-polynomial amount of time to compute. Thus, an approximation must be found such that the running time of the model remains polynomial while the ranking results remain accurate.

To improve running time, the ranking model algorithm compares the attributes in one of two ways. The first method simply pairs each attribute from the mapped object to the attribute of the target object that has the highest similarity value. However, this method can allow attributes to appear in more than one pair. This relaxed restriction yields a polynomial time algorithm for the ranking model. The second method for comparing attributes is to sort the list of attributes from each object in a pre-defined order and then compare them one-to-one. The algorithm keeps track of the number of matching attributes and computes the score based on the number of matches.

5.2.1 Assigning the Best Value

Relaxing the requirement that the model ensures each attribute is mapped to only one other attribute simplifies calculation. The model compares each attribute to those of the other object and assigns it to the attribute that maximizes the similarity score. The model must consider several problematic situations with this best-value approach in order to maintain good performance.

The first situation is when the attributes of the mapped object match a subset of the attributes of the target object. In this case, the model could return that attributes match up exactly while there exist attributes that do not match up. Figure 32 is an example of this situation. Attributes 1-3 each obtain the best similarity score when paired with Attribute A. If each attribute received a perfect score match with Attribute A, the ranking model would return a perfect score. To avoid this problem, the ranking model looks for the best mapping for each of the attributes of the mapped object and for each of the target object.

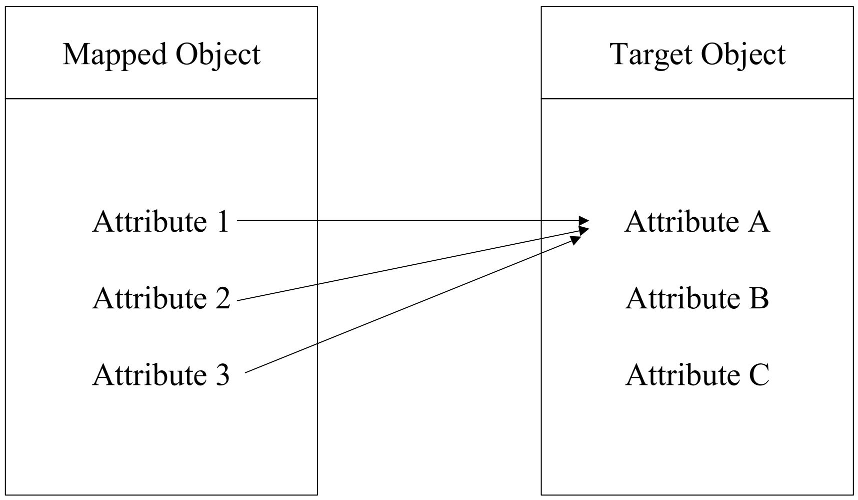


Figure 32 Imperfect Object Match with One-Way Mapping

By comparing the mapped object to the target object to each other, the model avoids returning a perfect score because the values obtained from Attributes B and C when compared to the attributes from the mapped object will not return a perfect score. The ranking model determines the similarity of the attributes by looking at the values in a bi-directional mapping.

Even though a bi-directional mapping improves the precision of the ranking model, the number of attributes must also be considered. If the ranking model did not consider this, it could return a perfect match for the situation illustrated in Figure 33. In Figure 33, each attribute can be matched to one in the other object, but multiple attributes are matched to the same object. Because there are more attributes in the mapped object, however, the target object is not an exact match.

The ranking model adjusts the similarity scores of the attributes to prevent the situation in Figure 33 from receiving a perfect score. The ranking model multiplies the probability that the attributes match by the ratio of the number of attributes in each object. For example, if one object had five attributes and the other had four, the model multiplies the score by 4/5. The ratio is always less than or equal to one, so that it never raises the score. Although the ranking model relaxes this restriction for improved speed in calculations, penalizing objects that fall in this situation will reduce the error from this restriction relaxation.

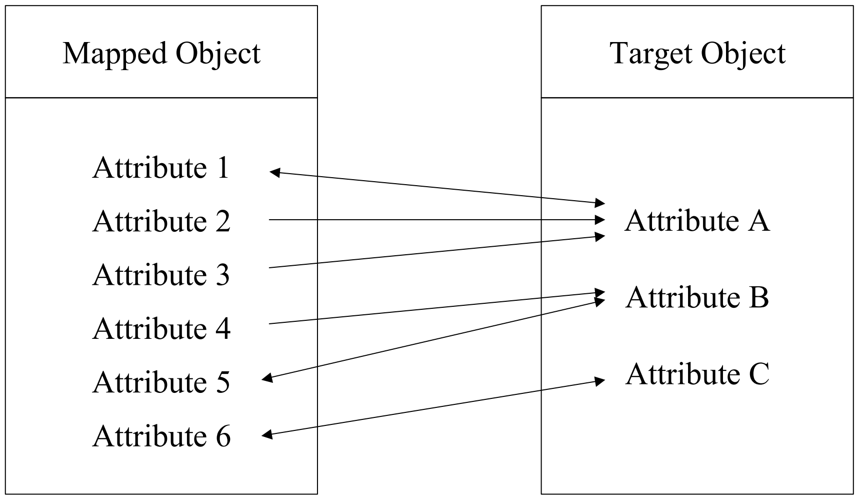


Figure 33 Imperfect Object Match with Two-Way Mapping

The benefit of comparing the attributes in this manner is that the model can consider partial matches. A mapped attribute does not have to match exactly to be considered into the score. This allows for the possibility that designers may decide to represent two attributes that are semantically the same in different ways. For this benefit, the ranking model makes additional calculations to ensure the problems discussed earlier, lack of a bi-directional mapping and differing numbers of attributes, do not improperly influence the score of the object. To decrease the number of calculations, the ranking model could choose to only evaluate exact matches, instead of the partial matches considered with this method.

5.2.1.1 Determining the Similarity Score of an Attribute

As mentioned in the previous section, the best-value approach sets the mapping for an attribute to the attribute in the target object to which it is the most similar. The approach uses a ranking model to determine the similarity score of one attribute to another in order to determine the most similar attribute.

The mathematical foundation of this model is analogous to that of the model for object similarities. The probability that an attribute matches another is approximately the probability that each of the characteristics of the attribute match. So, the equation used to compute the similarity value of two attributes is the same as Equation 3, with *attribute* substituted for *object*. Section 5.3 discusses the characteristics of attributes and why each was or was not chosen to compute the similarity of the attributes.

5.2.2 Comparing Attributes Using an Ordering

An alternative method to comparing attributes is to develop a comparator that takes two attributes and returns whether they are equal or if one is greater than the other. The ranking model uses fewer comparisons with this method, however this method relies on some assumptions regarding how the designers develop ontologies.

This method orders the set of attributes in a sequence of increasing order, based on the results of the comparator. When comparing the attributes of the object, the ranking model retains a pointer into the sequence of attributes for each object. The pointers act as placeholders in the sequences for comparison. If the attributes match, the score for the attribute comparison increases and both pointers increment. Otherwise, the pointer of the smallest attribute advances to the next attribute in its sequence. Figure 34 illustrates the possible situations and the resulting pointer modifications.

In Figure 34(a), attribute AC is less than attribute A2, so the pointer moves to AD. The subsequent comparison would then be between A2 and AD. In Figure 34(b), the attributes are equal, so both pointers are incremented. Finally, Figure 34(c) illustrates the process when A2 is less than AC. The pointer is incremented so that the next comparison will compare attributes A3 and AC.

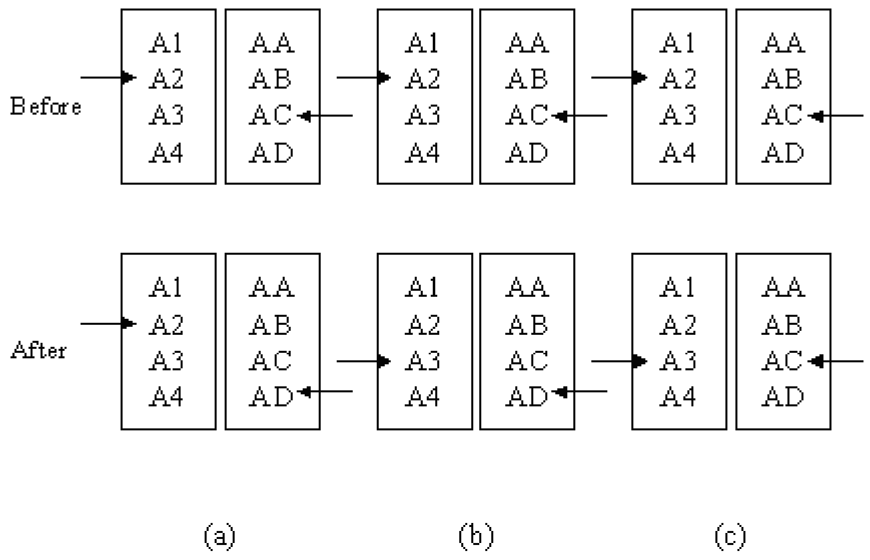


Figure 34 Possible Situations When Comparing Attributes With an Ordering

Comparing the attributes in a sorted order reduces the number of comparisons needed to compute the similarity of the attributes. In this method, the number of comparisons between the attributes of each object is $O(n+m)$, where one object has n attributes and the other has m attributes. Since the ranking model orders the attributes into a sequence, the total running time must include the sorting algorithm. If an $O(n\log n)$ search is used, the overall comparison of the similarities of the attributes is $O(n\log n + m\log m + m + n)$ which can be reduced to $O(p\log p)$, where p is the larger value of n and m. This running time is better than the $O(nm)$ running time of the best-value approach.

Although this method provides for faster computation, it does not allow for partial matches. The attributes have a strict ordering and the development of the comparator requires a decision on what constitutes equal attributes. The answer to this question depends on what assumptions are made about the designers of ontologies.

The ranking model will use two different comparators that rely on different assumptions. The first comparator simply compares the data type of the attributes. If the data types match, the comparator returns that the attributes are equal. Else, the comparator uses a pre-defined ordering on the data types to return which attribute is larger. The underlying assumption for this comparator is that designers of ontologies will select the same data type to represent the same semantic content in the domain. One problem with this approach, however, is that an attribute such as *Number_of_children* could be returned equal to *Age* because both are represented as integers, while they do not represent the same attribute.

The second comparator attempts to reduce this error by first comparing the data types and then comparing the other characteristics of the data type. By using the other characteristics, the comparator attempts to eliminate false matching. For this ranking model to work, matching attributes must have the same data type and the same values for all other characteristics.

5.3 Computing the Similarity of Attributes

Similar to the characteristics the ranking model uses to evaluate the similarity between objects, the ranking model only uses some of the characteristics of an attribute to compute the similarity between attributes. Table 2 provides a summary of attribute properties along with whether the ranking model includes each in the computation of the attribute similarity score.

Characteristic	**Included or Not Included in Ranking Model**
Name	Included
Data Type	Included
Required Attribute	Included
Multiplicity	Included
Description	Not Included

Table 2 Attribute Characteristics and Their Use in the Geometric Score Reduction Ranking Model

5.3.1.1 Attribute Characteristics Used by the Model

Name. The ranking model uses the attribute's name field in a similar manner to the name field of an object. The name is a way of improving the scores of attributes that have similar names, while not heavily penalizing the scores of attributes that do not match on their name characteristic. A case insensitive string comparison is used to determine if the names match.

Data Type. This research considers the data type of an attribute to be the most distinguishing feature when comparing attributes. If the data types do not match up, there is not a high probability that the attributes represent similar semantic ideas. To allow for the possibility that one designer may represent an attribute as a float while another designer chooses to use an integer, the ranking model reduces the similarity score of the attributes based on what the two data types are. For example, if one attribute is an integer while the other attribute is of a type *Class*, defined elsewhere in the data model, the ranking model will set the similarity score to zero. This is because the probability that designers might switch these two data types is low. When comparing a float to an integer, however, the ranking model will reduce the score less than the previous example, as it is more likely for designers to interchange floats with integers than floats with *Class* objects. Through this method, the ranking model allows for the possibility that attribute types might be represented differently by designers, while ensuring perfect matches still receive the best similarity score.

Required Attribute. A required attribute must be contained in every instance of an object. This is a significant characteristic that the ranking model uses to decrease the scores of those attributes that do not match exactly. The ranking model only penalizes the similarity score by 35% for a non-match, because designers may choose to represent an attribute as required while another would not.

Multiplicity. An object can have an attribute that contains multiple instances. The ranking model uses this important characteristic in computing the similarity score. If one attribute can be multiple and the other cannot, there is a low probability that the attributes represent the same semantic content. If the attributes do match, the ranking model compares the minimum number required and the maximum number of instances allowed. The ranking model reduces the similarity score of the attributes if either one or both of these sub-characteristics do not match. The penalty for each is less than the penalty for the multiplicity characteristic not matching.

5.3.2 Attribute Characteristics Not Used by the Model

The ranking model does not use the description field of an attribute for the same reason the description field is not used for computing the similarity of objects. The difficulty in accurately determining the similarity of this field outweighs the benefits from including this field into the similarity calculations.

VI. Results

This chapter discusses the evaluation of the additions to MaSE and the performance of the Geometric Score Reduction Model based on the measurement criteria defined in Chapter III. The first part of the chapter evaluates the use of ontologies to represent the information domain. The second part then compares the MaSE extensions to the criteria for using information domain specifications in multiagent system specifications. The final section evaluates the precision and time performance of the Geometric Score Reduction Model under various operating conditions.

6.1 Evaluation of Ontologies for Domain Representation

The characteristics and structure of ontologies described in Section 3.2.1 satisfy the requirements for an information domain representation in multiagent systems discussed in Section 3.1.1. Table 3 summarizes the requirements from Section 3.1.1 along with the ontological component that satisfies the requirement. The satisfaction of each requirement is discussed below.

Domain Representation Requirement	**Satisfying Ontology Component**
Define and Describe Objects used by the system	Classes
Specify Properties of the Objects	Slots
Specify Relationships between the Objects	Slots
Specify Axioms Regarding the Objects	Axioms
Specify Domain Metadata	Metadata
Clear, Concise, Consistent	Clear, Concise, Consistent
Only Specify Required Information	Complete

Table 3 Domain Representation Requirements and the Ontological Components that Satisfy Them

Define and Describe the Objects used by the system. The *Class* structure in an ontology defines and describes the objects in the domain. The structure allows for the object to have a name and a description of what the object represents.

Specify Properties of the Objects. *Class* objects can have multiple *Slots*. The *Slots* describe the properties and attributes of the object. For properties, the *Slot* can be named *has_property* to illustrate that an object has a particular property. The description of the slot allows the designer to explain the property even further.

Specify Relationships between the Objects. The *Instance* type of *Slots* can be used to specify relationships among objects. The instance defines a relationship by its name and description. The allowed values of the *Slot* can specify the other objects involved in the relationship. For example, the relationship *works_for* can be defined between a *Worker* class and a *Manager* class. The *Worker* class has an instance slot named w*orks_for*. The allowed values of the slot are instances of the *Manager* class. In this manner, using *Slots* with the type *Instance* allows the designer to specify relationships among objects in the domain.

Specify Axioms Regarding the Objects. This requirement maps directly to the axioms in an ontology.

Contain Metadata about the Domain. The domain representation must contain basic information about the purpose for which the representation was designed. Similarly, ontologies require the same information to facilitate future reuse. This desire for reuse in both ontologies and multiagent systems requires designers to document their projects well to allow for greater understanding in the future. Ontologies and multiagents systems both require a name, designer names, description, version number, etc. The metadata required for an ontology meets all the requirements for a domain representation.

Clear, Concise and Consistent. This requirement matches perfectly with the characteristics of an ontology. Both multiagent systems and ontologies are designed for future reuse and predictable results, so these characteristics are required in each. They ensure that the system or ontology can be understood in the future and that neither can produce inconsistent or undesired results.

Only Specify Information Necessary for System Execution. In a domain representation, it is important that the designer only specify the minimum requirements necessary for proper execution of the multiagent system. With ontologies, designers are often told "The ontology should not contain all the possible information about the domain: you do not need to specialize (or generalize) more than you need for your application" [Noy & McGuinness 2001]. The ontology should be complete to the level of granularity specified in the metadata for the ontology, so for multiagent systems the designers can set the granularity to that required for the proper execution of the system being designed. This requires the designer to describe what type of multiagent system is under development to allow future reviewers to understand the purpose for which the ontology was built. By setting the level of granularity of the ontology appropriately, ontologies ensure that only the minimum amount of information required for proper system execution is specified.

6.2 Evaluation of the Use of the Information Domain in Extended MaSE

Chapter III also defined requirements for using the information domain in multiagent systems development, and this section discusses the extensions to MaSE and how well they meet the requirements described in Section 3.1.2. Table 4 summarizes the criteria and whether extended MaSE fulfills them. The table also specifies whether Gaia and MESSAGE, the two other multiagent systems engineering methodologies discussed in Section 2.2.2, fulfill these criteria.

Domain Representation Requirement	**Met by Extended MaSE**	**Met by Gaia Methodology**	**Met by MESSAGE**
Allow for Specification of the Information Domain	Yes	No	Yes
Specification Should Occur Before Designing any Information Sharing in the System	Yes	No	Yes
Allow Designer to Specify Objects to Pass between Agents	Yes	No	Yes
Allow for Agent Data Models	Yes	No	No
Allow Designer to Specify Relationships between Data Models	Yes	No	No

Table 4 Criteria For Using Information Domain in Multiagent Systems Design

As one of the first agent oriented methodologies, the developers of Gaia address the behavior of agents that make the system development different from traditional software development. The resulting documents describe variables, but the methodology does not discuss how to specify the type of variables or the specification of the information domain. This development is left to the user, in a manner similar to the low-level design, as discussed in Section 2.2.2.1.

Section 2.2.2.2 discussed how MESSAGE uses a *domain* view to represent the information domain of the system. This view is developed in parallel with the behavioral and structural models. This parallel development allows the user to modify the information domain specification to include any object found to be necessary while designing the structural and behavioral models. While this satisfies the first three criteria in Table 4, the methodology does not allow for agents to have a different model than the system data model. Without the ability to specify agent data models, MESSAGE does not fulfill the last two criteria in Table 4.

The results for the Table 4 regarding MaSE come from the experience of using extended MaSE to develop a distributed course scheduling system. The purpose of the system is to allow for the scheduling of classes for instructors and students. Various individuals can schedule simultaneously using the distributed information sources of the system. Appendix A further describes the requirements and development of the system throughout the MaSE process, while the rest of this section discusses how extended MaSE fulfills the criteria based on the experience of developing the course scheduling system.

6.2.1 Specifying the Information Domain

The first extension to MaSE involves creating the system ontology. This extension meets the criteria for allowing the designer to specify the information domain at the appropriate time in the design process. The previous steps in MaSE, Goal Hierarchy and Use Cases, provide a set of terms for consideration as possible objects in the ontology. Table 5 shows the terms for the scheduling project derived from previous steps. Seven of these nine terms become part of the ontology, providing useful information to assist the designer in developing the data model.

Terms	Derived From
Schedule	System Requirements
Course	
Section	
CourseType	
CourseOffering	
Room	
Student	System Goals
Instructor	
User	Sequence Diagrams

Table 5 Candidate Ontology Terms

The creation of the system ontology occurs at the appropriate time in the development process. Earlier steps analyze the goals and situations the system will encounter. The step after the ontology creation passes messages and information amongst the roles. This information passing occurs with parameters that are specified as objects from the ontology. By placing ontology creation right before task creation, the methodology allows the user to analyze the problem domain thoroughly before creating the data model. Designers can determine exactly what information is necessary for the system while creating the data model before it is required for the rest of the development process, meeting the criteria to develop the model before designing information sharing in the system.

One alternative is to place the ontology creation before the creation of use cases and sequence diagrams. This placement, however, would require the designer to specify the information domain before fully analyzing the sequence of events that occurs in the system. This location allows for the objects in the ontology to be included as parameters in the sequence diagrams, but the use cases and sequence diagrams provide important details regarding exactly what information must be included in the objects, which outweighs the benefit of placing the ontology creation before the sequence diagrams.

6.2.1.1 Steps

Not only does the placement of the Create Ontology phase logically flow with the software development process, but the steps within the phase also flow from one to other. At each step, the inputs of the previous step are used to continue the development of the ontology. The steps are also an iterative process, allowing the designer to make modifications at any time during analysis, design, and development

The first step creates the class hierarchy and provides the designer with a skeleton framework for the rest of the ontology development. This framework then expands with the addition of the attributes of the objects. One development hurdle with creating attributes is that some users may find it difficult to separate between defining attributes that strictly describe the object and those that describe relationships in the information domain. The designer can combine these two steps, if desired, to reduce confusion over which type of attributes to create at each step.

Once the user specifies the information domain in terms of objects and their attributes, the next step records the requirements on the data in the form of axioms. Considering the functions the system performs on the data aids in the development of the axioms. In the course scheduling system, for example, the scheduling agents compare the *ScheduledEvents* of a *Person* to ensure that the person is not required in two different places at the same time. The ontology describes this axiom to ensure later developers know to code the system appropriately to prevent the situation from occurring.

Existing use cases simplify the final step of refining and validating the ontology. The designer iterates through the sequence diagrams or use cases, and ensures that the created data model contains all the information needed to accomplish the described events. If there are any problems with the data model, the appropriate modifications are made. Once the designer tests the data model against all the use cases, the ontology creation step is finished.

6.2.2 Using Objects from the Information Domain

Once the designer specifies the information domain for the system, the objects in the model describe the information passed throughout the rest of the MaSE process. The classes become parameters in the task and communication state diagrams, as well as attributes and parameters to methods in the components. This section evaluates the benefits of including the information domain in behavior specifications of the system, meeting the criteria of allowing the developer to use the data objects in the design specifications.

6.2.2.1 Task and Conversation State Diagrams

MaSE describes tasks and conversations using finite state automata, so the experience with using the ontology is the same for each case. With the existing data model, the designer can specify the type of objects passed between agents, as shown in Figure 35. This is an immense improvement over the previous version of MaSE, as designers can now observe the flow of information in the system. In Figure 35, the agent will receive a *Schedule* object. The developer knows the appropriate object to cast the received parameter to when this conversation is implemented in code. Without the ability to specify the object type, the person responsible for coding has to guess at the appropriate object.

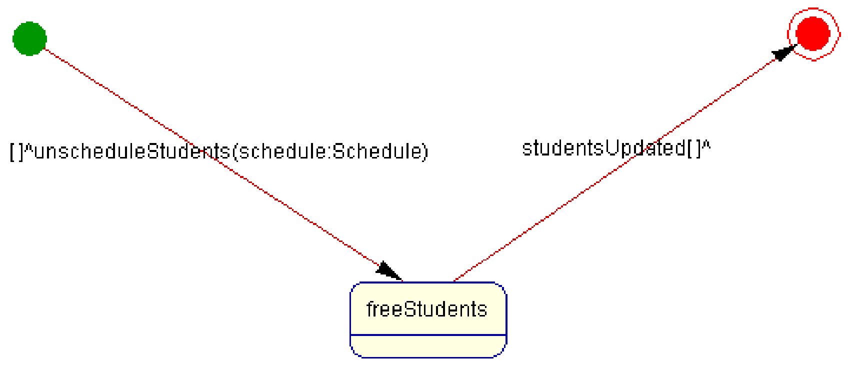
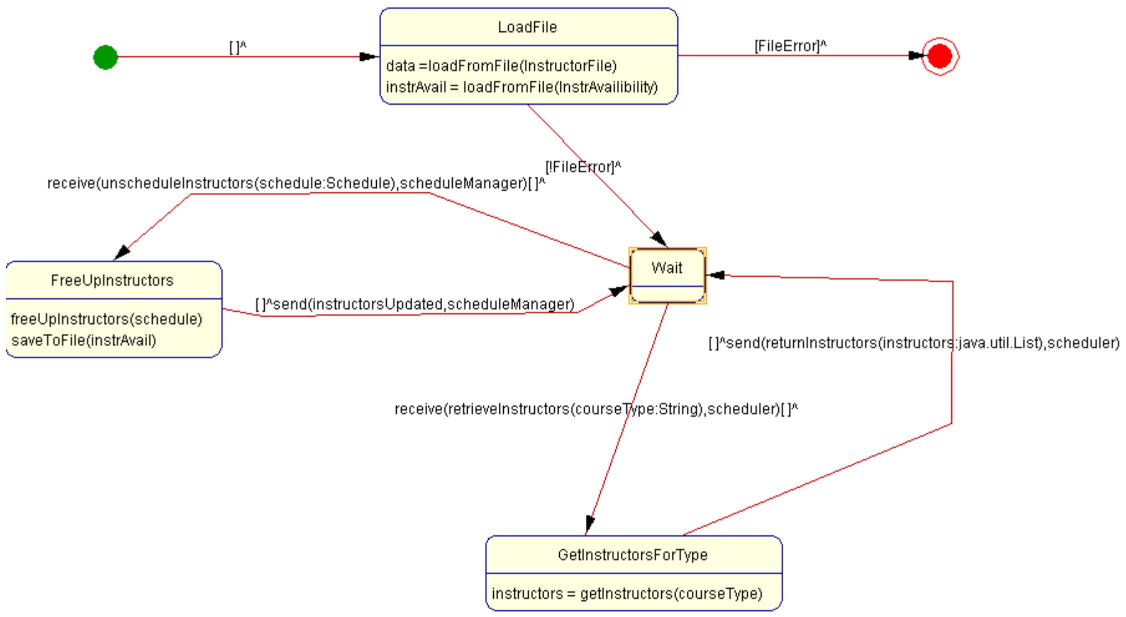


Figure 35 Sample Finite State Automata with Information Passing

With the ability to specify parameter types, designers also know the exact structure of the information passed between agents and can verify that each agent has all the information needed to accomplish its goals. Previously, the designer passed in a parameter assuming the developers would realize to code that information into that object. With a specified data model, the user can specify the object and ensure the information is an attribute of the object. If the information does not appear in the ontology, the designer adds it to the data model.

One issue that still exists within MaSE is that the diagrams use local variables, as shown in Figure 36. In the state *LoadFile*, the agent has a local variable *data*, which is not defined in previous documents. Designers do not define local variables until the next-to-last step in MaSE, Constructing Agent Components. This requires the designer to either skip ahead to define the variable in a component or remember the type of each variable for specification in the documents developed later. The fix for this problem is to include some type of section attached to the finite state automaton for the conversation or task which lists the variables and functions included in the state machine. In this manner, the designer could describe that *data* is a list of *Instructor* objects so that the variable can be carried through the rest of the design documents with a data value description.



[]^
LoadFile
data =loadFromFile(InstructorFile)
instrAvail = loadFromFile(InstrAvailibility)
[FileError]^
[!FileError]^
receive(unscheduleInstructors(schedule:Schedule),scheduleManager)[ ]^
FreeUpInstructors
freeUpInstructors(schedule)
saveToFile(instrAvail)
[ ]^send(instructorsUpdated,scheduleManager)
Wait
[ ]^send(returnInstructors(instructors:java.util.List),scheduler)
receive(retrieveInstructors(courseType:String),scheduler)[ ]^
GetInstructorsForType
instructors = getInstructors(courseType)

Figure 36 Task Diagram with Unspecified Variable

6.2.2.2 Components

The MaSE extensions also assist the user with creating the component architecture. Previously, the designer could define an attribute as any data type. If this type was not a programming language data type, such as `int`, the object type was not specified anywhere in the design documentation. Now, the designer can choose the variable's data type from the component ontology or a system type. When it is time to code the system, the developer now knows exactly how to code up the types developed for the system and which components use those types.

To fulfill the criteria of allowing agent data models, extended MaSE includes component ontologies. These ontologies further increase the usefulness of MaSE, allowing designers to integrate interface agents to legacy systems. The user can create component data models based on the system the agent interfaces with and describe how the model relates to the system data model by specifying which classes and attributes correspond to one another.

The final requirement, to allow the designer to illustrate the relationships between the data models, is satisfied by the development of a mapping between the system and component ontologies. The implementation of this step in agentTool assists the user by suggesting objects to map to based on the ranking of objects by the Geometric Score Reduction Model.

6.3 Geometric Score Reduction Model

The metrics of time and rank of the relevant object, discussed in Section 3.3.2, provide a method for comparing the performance of the ranking model against a baseline alphabetical model and the performance of the three implementations of the model. The control set is a model that returns the objects sorted in alphabetical order. This research uses this model because when mapping, an alphabetical listing is preferred to a hierarchical tree structure. A tree structure requires the designer to remember the parents of a class instead of simply the name of the class. With an alphabetical listing, on the other hand, the designer can scroll through the list to find the appropriate object, without having to navigate through a tree-like structure. This alphabetical model provides a baseline for comparing the various implementations of the Geometric Score Reduction model.

As discussed in Chapter V, the ranking algorithm uses two methods to compare attribute structure. The first method, *best-value*, computes the similarity score between attributes and matches the attributes to maximize the total similarity score. The comparator method orders the attributes based on their characteristics and then matches two attributes if they are equal. This research develops a strict and a loose comparator, where the loose simply uses the data type to decide whether two attributes are an exact match while the strict uses additional characteristics, such as the required and multiplicity characteristics of an attribute. Each experiment evaluates the performance of the baseline model along with the performance of the *best-value*, strict comparator, and loose comparator methods of the Geometric Score Reduction Model.

6.3.1 Experiment Setup

This research runs three experiments to test the models. The first experiment ensures the ranking model operates as designed by mapping an ontology to itself. The second experiment tests a general case of mapping between same-sized ontologies to evaluate possible future use of the ranking model. The final experiment tests the ranking model when mapping from a smaller ontology to a larger ontology, the condition that occurs when mapping from component to system ontologies.

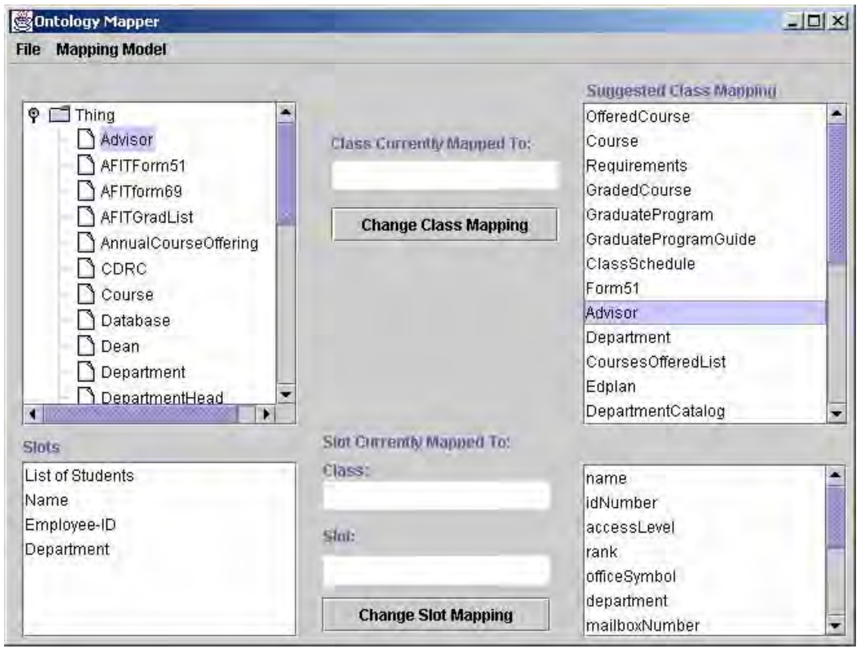


Figure 37 Example Experiment Run

Each experiment involves two ontologies: *from* and *target*. Every object in the *from* ontology is mapped to the *target* ontology. Figure 37 demonstrates a sample run in the experiment. In this sample run, the ranking model ranks the objects in the right list based on their similarity to the *Advisor* object. The relevant object in the *target* ontology is the *Advisor* object, ranked ninth in the list. The experiment records this ranking and then ranks the *AFITForm51* object. At the end of all the runs, the average rank is determined along with the standard deviation of the ranks. The ranking model ranks the objects in less than one millisecond, so each run is executed 1000 times to obtain a time measurement to compare the time metric.

Before starting the experiments, this research chose which ontologies to use. This was a complex task as most of the available ontologies are either very small or do not provide many attributes to the objects in the domain. For example, some available ontologies create a hierarchy of objects to distinguish between objects in the domain and rely on the user to understand what the object represents by the name of the object. Computer-based systems cannot use these types of ontologies, as the objects must contain attributes to actually pass information in the system. After a long search, this research found two ontologies for the smaller to larger case from the DAML Ontology Library. The similar sized case experiment uses data models created by students at AFIT for a software engineering project. The data models represent the same domain, but the students chose to represent some overlapping semantic concepts while also representing some concepts not represented by the other groups.

6.3.2 Mapping to Identical Ontology

The first experiment shows that the three versions of the ranking model work perfectly under the optimal condition of mapping to identical objects. This experiment maps the four ontologies identified above to themselves. For each run, the ranking model returned the identical object as the number one ranked object based on similarity.

This is expected because an identical object receives a similarity value of 100% due to the name, role, and attributes matching exactly. The only way the relevant object would not be returned number one is if another object receives a perfect similarity score, also. For that to happen, however, the other object would have to match the original object's name, role, and attributes. If so, the ontology contains two objects that are exactly the same, which is not possible in properly constructed ontologies. Therefore, when mapping an ontology to itself, a correct implementation of the ranking model will always return the relevant object as the number one ranked object.

6.3.3 Mapping to Similar Sized Ontology

After verifying that the versions of the ranking model perform correctly in optimal situations, this research tested them using two ontologies with the same number of objects. As discussed earlier, this thesis tunes the parameters of the Geometric Score Reduction Model to work best when mapping a smaller ontology to a larger ontology. However, testing against similar sized ontologies can illustrate performance in a broader case.

For this test, the research uses two data models developed by different teams for the same software project. Each model contains the information necessary to construct a registrar system for the term project in a software engineering course. The system ontology was constructed by Chad Harris, Nate Jensen, and Choung Kil while Eric Trias, Rick Rapallo, and Rick Day constructed the component ontology. Each data model is described in detail in Appendix B.

The experiment runs every object through the ranking model for comparisons with the objects in the target ontology. There are cases where an object was not in the target ontology, and this case was omitted from the experiment since there is no correct answer for the ranking model to find. The development teams used these ontologies to design the system, but did not actually implement it, so the teams do not describe the same set of semantic concepts. If the project included coding the system, the data models would have been refined to include the same semantic concepts.

Table 6 presents the experimental results from the various versions of the ranking model. The control case of the alphabetical model averages a rank of 13. This is no surprise as there are 26 objects in the target ontology. Since every object is tested, the expected rank is half the number of objects. Every version of the Geometric Score Reduction Model is slower, but outperforms the alphabetical model in ranking the objects.

	Best-Value	Strict Comparator	Loose Comparator	Alphabetical
Average Rank	6.238095	7	7.428571	13.14286
Standard Deviation	5.448897	6.778467	5.143651	7.525196
Average Ranking Time for 1000 Ranks (ms)	1157.3333	396.8095	340.6667	31.4286

Table 6 Similar Sized Ontologies Ranking Model Results

The *best-value* approach averages a rank of approximately 6, or 24% of the number of objects in the target ontology. This approach has the largest run time of 1.157 seconds to return the ranked objects 1000 times. This increased execution time is due to the polynomial running time of the algorithm that compares the attributes of the object, as discussed in Chapter III. Using this version of the ranking model, the designer looks at 29% of the objects in the target ontology, on average, before finding the correct mapping. This is a definite improvement over the alphabetical model.

Using a strict comparator to evaluate the attribute similarities, the ranking model averages a rank of 7, or 27% of the number of objects in the target ontology. This accuracy is slightly less than the *best-value* approach, but executes in approximately one-third the time, based on the average results from ranking objects 1000 times. The reduced running time is a product of the *n*log n* running time of the attribute comparison algorithm, as discussed in Chapter III.

The loose comparator method performs slightly lower than the strict comparator in this experiment. The method averages a rank of almost 7.5, a significant improvement over the alphabetical model. The loose comparator method performs the worst because it only looks at the data types of the variables. With this method, it can yield false positives, non-relevant objects that it thinks are relevant. Because the algorithm compares fewer characteristics, however, it executes slightly faster than the strict comparator method.

6.3.4 Mapping to Larger Ontology

The final experiment simulates the intended use of the Geometric Score Reduction model. A smaller ontology, consistent with that of a component, is mapped to a larger ontology, consistent with a system ontology. The system data model is the general concept model built by Jeff Heflin [Heflin 2000]. The component data model is the GEDCOM ontology, a genealogy-based data model built by Marti Hall [Hall 2001]. The ontologies can be found online [DAML+OIL], and the exact website of each is discussed in Appendix B.

This experiments uses these ontologies because the general concept model includes a majority of the semantic concepts of the GEDCOM ontology, while containing many more concepts than the GEDCOM model. Table 7 shows the results from the experiment.

	Best-Value	Strict Comparator	Loose Comparator	Alphabetical
Average Rank	2.166667	2	2	15.5
Standard Deviation	2.401388	2.44949	2.44949	7.342797
Average Ranking Time for 1000 Ranks (ms)	1076.1667	428.8333	352	35.3333

Table 7 Experiment Results for Smaller to Larger Ontology Mapping

Unlike the previous experiment, the *best-value* method performs slightly worse than the comparator methods. The performance of each of the methods is an average rank of two, or 6.666% of the total number of objects in the target ontology. This is a huge improvement over the traditional alphabetical listing that averaged approximately 51%. Using the ranking model, the designer looks at less than one-seventh of the number of objects required using the alphabetical ranking model.

6.3.5 Analysis of Implementation Approaches

Comparing the average rank of the implementations to the baseline model, using hypothesis testing on paired data, shows that experiments yield a *p*-value of less than one percent. This signifies that the experiments present enough data to conclude that the implementations of the Geometric Score Reduction Model have a higher precision than the baseline model. However, the experiments find no statistically significant difference among the precision of the three implementations.

Situations exist where one implementation is better suited than the others. The *best-value* method allows for partial matches when comparing attributes and is built to allow designers to specify similar attributes as different data types, as discussed in Section 5.2.1. This capability does not exist in the comparator implementations, so the *best-value* method is recommend for most situations.

When mapping to an ontology with a large number of objects, however, the user may find the *best-value* approach takes too long to rank the objects. In this situation, the comparator implementations should be used. The choice between the strict or loose comparator is determined by the designer's perception of whether or not the two ontologies are specified in the same level of detail. For example, if one model describes attributes with their multiplicities while the other model simply lists the attribute's names and data types, the user should use the loose comparator. If the characteristics are well-defined in each ontology, however, the strict comparator should be used to take advantage of these characteristics.

VII. Conclusions and Future Work

The previous chapters describe the extensions added to MaSE to address the information domain of multiagent systems and the Geometric Score Reduction Model developed to assist designers in mapping between ontologies. This chapter summarizes the results from the previous chapters and suggests possible areas for future work to enhance MaSE and the ranking model.

7.1 Summary and Conclusions

This research fulfills all the goals described in the introduction by maturing MaSE to address the information domain, providing a methodology to develop the structural, behavioral, and information models of heterogeneous multiagent systems. To fulfill the goals, the research identifies the requirements for constructing and using information domain specifications in multiagent systems. MaSE now includes the development and use of ontologies to define the information domain of the system. This research shows that ontologies can fully describe the information domain as needed in the development of heterogeneous multiagent systems.

These additions to MaSE lead the user through the creation of the system ontology and use of data model objects in agent behavior. The extensions mesh with the previous version of MaSE to ensure the MaSE steps logically flow through a software development process where the outputs of one step become inputs to the following steps. As part of the MaSE process, the designers now construct system and component ontologies by creating and structuring classes and attributes using terms extracted from the goal hierarchy, system requirements, and use cases. Agents can then use these classes to share information with one another.

By adding these steps that address the information domain, this research matures MaSE towards a complete methodology for building multiagent systems. MaSE now addresses the system's behavioral, structural, and data models, thus defining the aspects required to ensure a coded system will fulfill the initial requirements. With these models, the designer can ensure that each agent has the required information to fulfill all of the system requirements.

Part of the information domain model of the system describes the mapping between the system ontology and the ontologies of the agent components. To assist with creating these mappings, this research develops an information retrieval ranking model, the Geometric Score Reduction Model, which computes the probability that two object classes represent the same semantic content in the information domain. This research programmed the ranking model into agentTool, as a proof of concept, and showed that the ranking model can reduce the development time of these mappings to one-seventh of the normal development time. The experiments in Chapter IV also indicate that the ranking model can be expanded to assist designers in mapping any two data models while integrating systems.

7.2 Future Research Areas

7.2.1 Automatic Conversion from UML to Ontologies

The development of automatic transformations to translate UML specifications into an ontological format would help with the expansion of existing ontology libraries. With an automatic translator, designers could use existing UML data models without having to convert them manually into ontology specifications. This is particularly useful when integrating legacy systems, whose data models are specified in UML. Part of the research would include developing a standard ontology API. The current DAML+OIL API provides a starting point, but the API is new and needs to mature to input and output other ontology specification languages.

7.2.2 Creating Converters from Ontology Mappings

Currently, designers must develop code to implement the mappings between ontologies specified in MaSE. More research could develop automatic transformations to generate translators to implement the mappings. These translators would perform similar data marshalling to that done by distributed operating systems. They could take the objects from the system ontology and create the appropriate objects from the component ontology and vice versa.

The current mappings contain all the information necessary to create these translators. Future research must develop the proper architecture for the translators and develop the transformations to specify code to fit into the architecture to perform data marshalling for the multiagent systems. This research would further assist the developers by reducing the amount of code necessary for them to complete.

7.2.3 Ranking Model Enhancements

While this research shows that the ranking model performs well, user-feedback could improve the results of the model. User-feedback improves the precision of returned results in Information Retrieval systems, and can be applied to the Geometric Score Reduction Model. As the designer maps the objects of an ontology, the ranking model uses the selected objects to fine-tune the parameters of the model. These adjustments should improve the precision of any future rankings the model performs on the ontologies. To modify parameters, the model adjusts the probabilities used to compute similarity values based on the model's performance.

7.2.4 Transformations

There are also topics in MaSE for future research. Previous research by Clint Sparkman addressed the development of semi-automatic transformations to construct the design models from the analysis models in MaSE; however these transformations currently do not work consistently in agentTool. Future research should make these transformations more robust and include error catching so that any problems are properly reported to the user.

Constructing the design models from the analysis models is a straightforward transformation, so it is logical to have an automated transformation of this process. The automated process ensures that the design models represent all the concepts described in the analysis models, while a manual process might miss some concepts. The automated process also saves the designer the work of creating these design models, which can be a lengthy development process. The development of reliable, robust transformations for MaSE is the research area that provides the most benefits to users.

7.2.5 Improved agentTool Visual Interface

Another research area pertains to improving the visual interface of agentTool. The current tabbed interaction in agentTool fails to take advantage of the flow of information through the MaSE process. For example, to view an agent architecture, the user must click on the agent and then on the agent architecture tab. An improved interface would allow for the user to drill-down into the agent through a double-click, or some other method. This research area would require identifying and specifying the flow of information throughout the MaSE process and then constructing an interface to visualize this information and its flow in agentTool.

7.3 Summary

Designers will integrate many different automated systems to develop the global information grid called for by *Joint Vision 2020*. The Geometric Score Reduction model can aid this process, as this integration involves mapping the various data models of each system.

This research also addresses the need for a complete methodology for constructing reliable multiagent systems. MaSE provides designers with an engineering methodology to develop the information domain, behavioral and structural models for multiagent systems. Using MaSE, developers can construct the secure, reliable, and robust systems necessary to provide information superiority for our warfighters.

A. Appendix – Designing Distributed Scheduling System Using Extended MaSE

The goal of the project is to develop a semi-automatic course-scheduling program to allow users to organize the time and location of classes. The system is representative of a university-type environment for which the registrar's office develops a quarterly schedule. Within the university, there are multiple departments offering courses of differing types. Students may take courses from any department, and classrooms are shared resources between all departments.

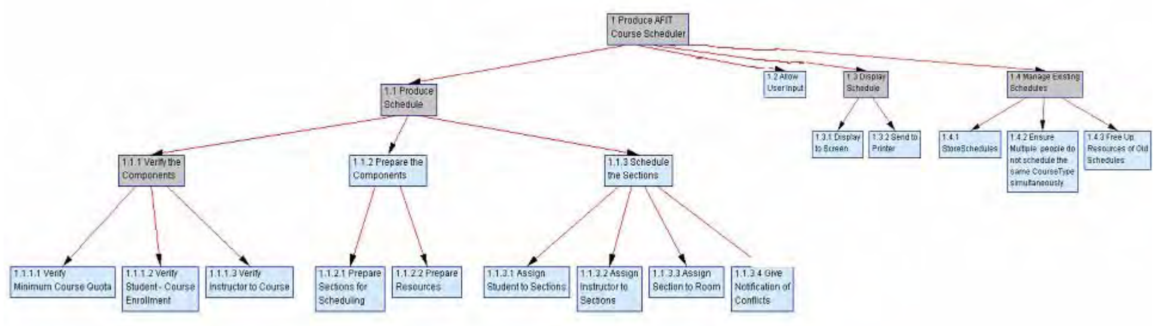
The generic network system described by the requirements has the required scheduling data resident on up to four separate computers and allows any number of concurrent schedulers from network machines. The system must fulfill the following requirements:

- Each database can be resident on any network computer, but the design must be robust enough to consider that the databases may be moved.
- There may be any number of schedulers required to schedule all the courses for a single quarter.
- There are 15 course types, each with a 4 character identifier. Each course has a three digit identification number and a two digit section number. A course may have multiple sections determined by class enrollment. A particular class offering is designated by the course type, course number and section number.
- Any scheduler can choose to schedule any one or more class types.
- Only one scheduler may schedule any given course type at once.
- Any scheduler should be able to print a copy of the current scheduling of any (one or more) course types.
- There should be only one master schedule resident on the system.
- The system should produce an integrated schedule that includes course type, course number, section number and time blocks consisting of day, start time, end time and room.

This appendix describes the development of this system using extended MaSE and discusses each step along with the results of the step and how the designer reached those results.

A.1 Capturing Goals

The first step translates the functional requirements into the goals of the system. The main goal for this system, based on the requirements, is to perform AFIT scheduling activities. The requirements then describe four subgoals: producing, displaying/outputting and managing schedules and handling user input. Figure 38 shows the complete goal hierarchy for the system. Every system requirement is met by one of these goals and the rest of this section describes each of the goals in detail.



1 Produce AFIT Course Scheduler
1.1 Produce Schedule
1.2 Allow User Input
1.3 Display Schedule
1.4 Manage Existing Schedules
1.1.1 Verify the Components
1.1.2 Prepare the Components
1.1.3 Schedule the Sections
1.3.1 Display to Screen
1.3.2 Send to Printer
1.4.1 StoreSchedules
1.4.2 Ensure Multiple people do not schedule the same CourseType simultaneously
1.4.3 Free Up Resources of Old Schedules
1.1.1.1 Verify Minimum Course Quota
1.1.1.2 Verify Student - Course Enrollment
1.1.1.3 Verify Instructor to Course
1.1.2.1 Prepare Sections for Scheduling
1.1.2.2 Prepare Resources
1.1.3.1 Assign Student to Sections
1.1.3.2 Assign Instructor to Sections
1.1.3.3 Assign Section to Room
1.1.3.4 Give Notification of Conflicts

Figure 38 Distributed Course Scheduling Goal Hierarchy

A.1.1 Produce a Schedule

This goal represents all the activities required to create the schedule for a specified course type. As the main system activity, producing the schedule is the most extensive goal. The first sub-goal, *Verify the Components*, is a partitioned goal since it is divided into sub-goals relating to particular schedule requirements. These subgoals ensure there are enough students for each course and that there are instructors to teach each course. Once the sub-goals are met, the *Verify the Components* goal is satisfied, also.

Prepare the Components encompasses all the goals relating to splitting class offerings into the proper number of sections, based on student enrollment and determining the available resources for scheduling. The *Schedule the Sections* goal contains all the goals relating to the actual scheduling of the courses; assigning a time and classroom to sections and assigning students and instructors to these sections.

A.1.2 Allow User Inputs

This goal handles all user input, allowing the system to interact with the user. The goal allows users to create and modify the schedules and to instruct the system how to handle any scheduling conflicts that may arise. Creating a separate goal for user input allows all the other goals to not have to handle the user input required for system execution.

A.1.3 Display the Schedule

This goal fulfills the requirement of allowing the user to view or print schedules. The goal contains a sub-goal for each of these two methods of viewing schedules. The goals pertain to both new and existing schedules, so the system does not handle the displaying separately for each of these cases. By adding an overall goal for displaying the schedule, the design reduces the number of goals needed when producing new or managing existing schedules.

A.1.4 Manage the Existing Schedules

Once schedules are created, the system must retain them for future modification or viewing. *Manage the Existing Schedules* handles this requirement by storing the goals and ensuring that multiple schedulers do not simultaneously schedule the same course type. Another subgoal is to unschedule the various sections in a schedule if a user decides to cancel an existing schedule. The goal is a partitioned goal, as it is satisfied by the combination of the three subgoals.

A.2 Applying Use Cases

After analyzing the system requirements, four use cases describe the scenarios in which the system participates. The first use case, *Produce Schedule*, is the scenario where the user tells the scheduler to schedule a set of course types. The scheduler displays the existing schedules for the course types and asks the user if the existing schedules should be discarded. If the user responds with an affirmative, the scheduler enters the *Erase Schedule* use case.

Erase Schedule frees up the resources used by the schedule. The system notifies the managers for the students, classrooms and instructors that the schedule is no longer valid and that the time and resources used by the schedule are now available. In this manner, the system updates all data to reflect the cancellation of the schedule.

Once the system erases the existing schedule, the *Automatically Generate Schedule* use case begins. This use case involves the scheduler first obtaining a lock on the course type to be scheduled. This allows the schedule manager to prevent simultaneous scheduling of the same course type. Once the manager grants a lock, the scheduler retrieves the data on all the sections needed for the course types, along with the information regarding the students and instructors for these classes. The system creates the schedule once it receives the availability of the classrooms. Because multiple schedulers can run at once, the system must check the new schedule with each of the managers to ensure the schedule is feasible. Each manager checks the schedule against the current state of the resources and replies whether the schedule is approved. If approved, the manager's resources are adjusted to reflect the times in the new schedule. This commit protocol can create a race condition when scheduling, as the first user to finish is more likely to have the schedule approved. This protocol balances the need for distributed scheduling with the requirement to not double-book resources. The schedulers can obtain the data, but the data may change during scheduling, requiring the system to receive the approval of the managers once the schedule is created.

Another option is to lock each resource, but then the system could only allow schedulers to execute one at a time. As a result of the need for simultaneous scheduling, the system will now have situations where the created schedule is not valid and must be rebuilt. If any of the managers disapprove the schedule because of conflicts, the system notifies any managers that have approved the schedule to free up the resources used by that schedule.

Once the system creates a valid schedule, the system shows the new schedule to the user and asks if the user accepts the schedule. If so, the schedule is passed to the schedule manager to save. If not, the schedule is passed to the schedule manager to have all the resources used by the schedule released.

The last use case, *Output Schedule*, is the scenario of the user wanting to view or print a schedule. The use case also contains the situation when the system initiates the display before and after creating a new schedule. Once the use cases describe all the system scenarios, the designer converts the use cases into sequence diagrams.

A.2.1 Sequence Diagrams

The sequence diagrams flow directly from the use cases. The *Produce Schedule* and *Erase Schedule* use cases each have one sequence diagram that fully describes their scenarios. The *Automatically Generate Schedule* has two sequence diagrams to illustrate the scenario when the schedule is approved and disapproved by the user. The last use case, *Output Schedule*, has three sequence diagrams for its three scenarios; user requests display to screen, system requests display to screen, and user requests print out.

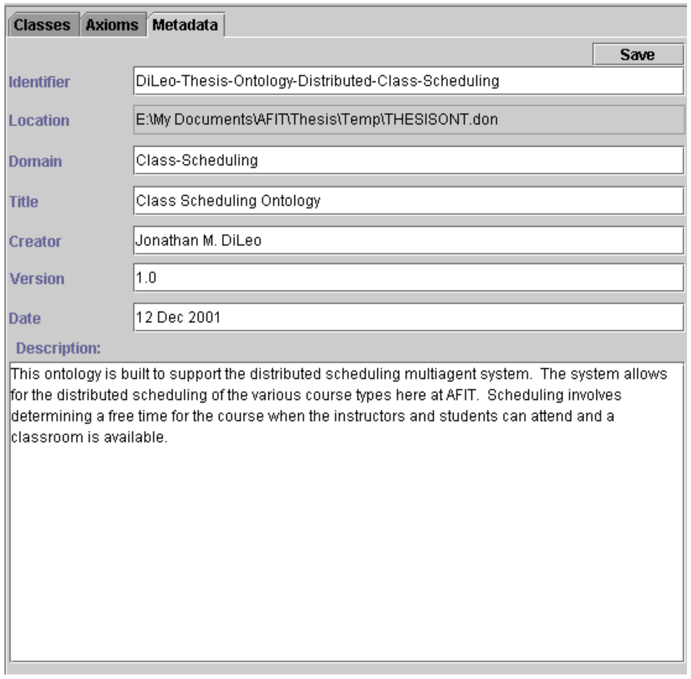
The construction of the diagrams from the use cases is straightforward and is not described in this research. The main decision in creating the sequence diagrams is the creation of roles for the system. This research developed a manager for the instructors, classrooms, students, courses and schedules. These *manager* roles come from the requirement that the source data may be located on separate servers. These roles act as an interface to the service providing the data for the individual resources. The *user* role represents the user in the various scenarios. Because the system encompasses all of the roles, the role of *scheduler* represents the part of the system that performs the scheduling algorithm. Similarly, the *output manager* handles the displaying and printing of the schedules. The analyst used these roles to construct the sequence diagrams for the use cases.

A.3 Developing System Ontology

Before further specifying the system behavior, the designer constructs the ontology for the project. The system uses this data model to share information between agents to accomplish the systems goals. This section discusses the development of a data model for this multiagent system.

A.3.1 Define Purpose and Scope

The system requirements determine the purpose and scope of this ontology, and Figure 39 shows the metadata for the developed data model. The ontology contains only the information needed to schedule classes, with this scope leaving out information such as the address of individuals or the grades received by the students. Once the scope and purpose, along with the other metadata, are accurately described, the designer can start the construction of the ontology.



Identifier	DiLeo-Thesis-Ontology-Distributed-Class-Scheduling
Location	E:\My Documents\AFIT\Thesis\Temp\THESISONT.don
Domain	Class-Scheduling
Title	Class Scheduling Ontology
Creator	Jonathan M. DiLeo
Version	1.0
Date	12 Dec 2001

Description:

This ontology is built to support the distributed scheduling multiagent system. The system allows for the distributed scheduling of the various course types here at AFIT. Scheduling involves determining a free time for the course when the instructors and students can attend and a classroom is available.

Figure 39 Metadata for the System Ontology

A.3.2 Collect Data

The first step is to create the list of candidate terms for the data model. Table 8 shows the list of candidate terms derived from the system requirements, goals, and sequence diagrams. This list acts a basis for objects and attributes of the ontology, but designers frequently expand the list throughout the development process. The next section illustrates the expansion and use of the candidate list.

Terms	Derived From
Schedule	System Requirements
Course	
Section	
CourseType	
CourseOffering	
Room	
Student	System Goals
Instructor	
User	Sequence Diagrams

Table 8 Candidate Ontology Terms

A.3.3 Construct Initial Ontology

A.3.3.1 Existing Ontologies

If the developed system was part of a larger school wide information system, it is beneficial to use the data model from that larger system. For example, if a college-wide ontology existed, using that ontology the system could integrate easily into any other information system using the college-wide model. This section builds an ontology from scratch to demonstrate the methodology, but for an actual registrar system it is recommended to build an ontology that discusses more than just the registrar portion of academia.

A.3.3.2 Build Class Hierarchy

Figure 40 shows the initial class hierarchy for the registrar system. The terms *course type* and *course offering* from the term list are not included in the ontology. *Course type* defines the type of a course and is appropriate as an attribute of the *Course* object. A *Section* represents the specific offering of a course, so a *course offering* object is redundant and not included in the data model. Along with not including some terms, the ontology includes terms not originally in the candidate list.

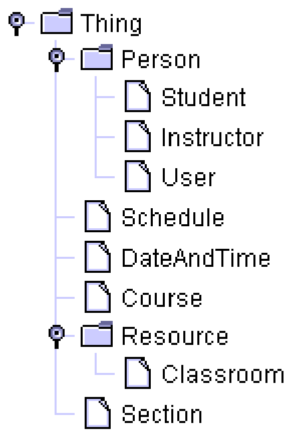


Figure 40 Initial Class Hierarchy

These additional objects stratify the objects and hold redundant information. *Students*, *Instructors*, and *Users* will all have similar attributes and so the *Person* class acts as a super class to hold these common attributes. *DateAndTime* holds the information regarding the start/end date and times of the courses. The initial ontology required that classes occur only in classrooms, but since there are other types of resources, such as lecture halls or auditoriums, a *Resource* class is added. The system will use this class to schedule sections in any type of room.

A.3.3.3 Add Attributes to Classes

Once the initial class structure is complete, the data model can add attributes to the classes. Table 9 shows the attributes of each object and the attributes' semantic representation. The question for creating attributes is *What information must the objects contain for the system to function properly*. Table 9 shows the attributes in the highest level they occur. For example, *name* and *idNum* occur in *Student*, *Instructor* and *User*, but are not shown in the table. The attributes represent the same semantic content in all of the subclasses of the objects in Table 9. The first pass in creating attributes involves those attributes that do not specify relationships among objects. The relationship attributes are added in the next step.

Object	Attributes	Attribute Description
Person	name	Name of the individual
	idNum	Social Security or other identification number of the person.
DateAndTime	date	Date of the instance
	time	Time of the instance
	duration	Duration of the object. So the stop time is this many hours from the start time/date specified in this object
Course	coursetype	The string defining the type of course (per system requirements)
	number	The course number (per system requirements)
	description	Brief description/name of the course
	numHours	Number of hours the class meets. System needs to know how many hours to schedule a week.
Resource	name	Name of the resource
Section	number	Section number (per system requirements)
Schedule	courseTypes	List of course types the schedule contains. Reduces the time necessary to determine what type of sections are in the schedule.

Table 9 Initial Attribute Listing

A.3.3.4 Define Relationships

Designers sometimes merge the previous step and this step into one, as both involve adding attributes to the objects. This step specifies the relationships between objects as attributes. A single attribute can represent a relationship, such as the fact that a *Section* is an offering of a *Course* represented as an attribute of the *Section*. In some instances, however, two attributes represent the relationship, such as a student registers for a section and the section must know all the students registered for it. This requires an attribute in both the *Section* and *Student* objects. Table 10 shows the attributes added to the objects and why the system needs the attributes. By recording the instructor, students, and classrooms in the *Section* object, the system can easily erase a schedule. Using these references the system knows exactly which objects to update. After describing all the attributes, the designer specifies any axioms regarding the attributes and objects.

Object	Attributes	Attribute Description
Student	coursesToTake	The Courses the student needs to register for. Once registered, the course is removed from this list
	registeredSections	The Sections the student is currently enrolled in. System will use to determine the available times for the student.
Instructor	canTeach	The courses the instructor is qualified to teach.
	registeredSections	Same as Student.registeredSections
Schedule	composedOfSections	The list of Sections contained in the schedule
Resource	usedBy	The sections currently using this room. Allows the system to determine the open time for this classroom.
Section	course	The Course object the section is an offering of.
	taughtBy	The Instructor object teaching the section.
	location	The classrooms that will hold the section.
	dateandTime	The times for the sections.
	registeredStudents	The Students that are registered for this section.

Table 10 Relationship Attributes for Distributed Scheduling

A.3.3.5 Define Axioms

The first system axiom requires that the sections of an instructor or student cannot overlap. In other words, a person cannot have two events scheduled at the same time. A similar requirement is that two sections must not occur in the same room at the same time. These two axioms specify the situations that cannot occur with the system data. With the axioms fully specified, the project can now validate the ontology using test cases.

A.3.4 Refine and Validate

Using the use cases, the designer simulates system behavior to ensure that the ontology contains all necessary information. If the project needs additional information, the designer modifies the ontology to meet the need. For this project, the ontology does not fulfill all the needs for the *Generate Automatic Schedule* use case. The first problem arises when the *SectionManager* creates the appropriate number of *Section* objects for the courses based on the student enrollment. The ontology does not describe the maximum number of students allowed in the course offerings. To fix this problem, the project adds the *maxNumberofStudents* attribute to the *Course* object. This attribute allows the system to know the maximum number of students that can occur in a section of the course, so that seminar classes can have smaller numbers than a lecture class.

The next problem occurs during the actual scheduling of the sections by the *Scheduler*. Instructors and students might have other events that conflict with the scheduling of sections, such as department meetings, but the ontology is incapable of representing these events. The project adds a *ScheduledEvent* object as a parent of a *Section* to represent these other events. A *Person* then has a *scheduledEvents* attribute that contains a list of all events that individual is required to attend. With these additions, the scheduler can verify that an individual has no activity scheduled during a specific time frame.

The final problem is a design issue, instead of the lack of information issues discussed above. The *DateAndTime* object currently represents the end time by the duration from the start time and date. This will require the system to compute the end time for every comparison while scheduling. This increased computation outweighs the extra memory needed to keep the end date and time. By adding an *endDate* and *endTime* attribute, the system can schedule the sections with less processor use.

A.3.5 Final Ontology

The ontology is validated once the designer is comfortable that the ontology includes all the information needed to execute the use cases. Figure 41 and Table 11 show the final class hierarchy and attribute list for the system data model. At anytime in the MaSE process, the designer may identify additional information required by the system and make the appropriate changes to meet this need. Like the other steps in MaSE, the construction of the system ontology is iterative and can be revisited at any point in the development cycle.

A.4 Refining Roles

This step starts with assigning goals to the roles from the sequence diagrams and then constructing tasks for the roles to accomplish the goals. This section discusses the goals and tasks of each role and why the analyst chose each assignment.

A.4.1 CourseManager

The course manager creates the appropriate number of sections for each course of a specified course type, based on the number of students enrolled. Because of this function in the sequence diagram, the role is responsible for accomplishing the *Prepare Sections for Scheduling* goal. The role accomplishes this goal through the *ManageSectionInfo* task, shown in Figure 42.

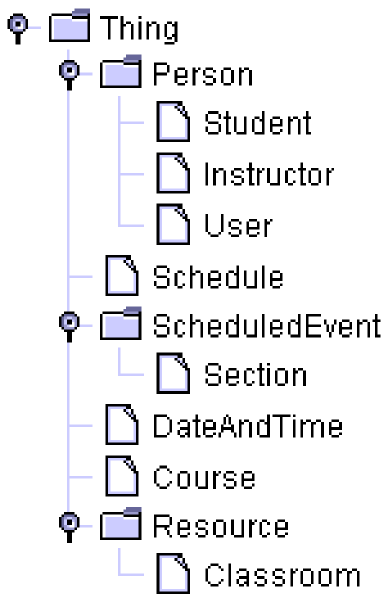


Figure 41 Final Class Hierarchy for the Registrar Ontology

The task begins by loading the data for the courses into memory from persistent memory. Persistent memory is necessary in case of improper shutdown of the system, ensuring the data in the system remains consistent. Once the data is loaded into memory, the task then listens for requests from other tasks.

When the task receives a request for the sections of a specific course type, the role requests the student data from the *StudentManager*. The task then creates the appropriate number of sections for each course and returns the list of sections to the requesting *Scheduler* role.

Object	Attribute	Description
Person	name	Name of the individual
	scheduledEvents	List of events the person participates in. System uses this list to term free times for the individual.
	idNum	Social Security or other identification number of the person.
Student	registeredSections	The Courses the student needs to register for. Once registered, the course is removed from this list
	coursesToTake	The Sections the student is currently enrolled in. System will use to determine the available times for the student.
Instructor	canTeach	The courses the instructor is qualified to teach.
Schedule	composedOfSections	The list of Sections contained in the schedule
	courseTypes	List of course types the schedule contains. Reduces the time necessary to determine what type of sections are in the schedule.
ScheduledEvent	dateandTime	The time window the event occurs in.
	location	The location (Resource Object) where the event occurs.
Section	course	The Course object the section is an offering of.
	taughtBy	The Instructor object teaching the section.
	number	Section number (per system requirements)
	registeredStudents	The Students that are registered for this section.
DateAndTime	startDate	Start date of the event
	endDate	Ending date of the event
	startTime	Start time of the event
	endTime	Ending time of the event
Course	courseType	The string defining the type of course (per system requirements)
	number	The course number (per system requirements)
	description	Brief description/name of the course
	numHours	Number of hours the class meets. System needs to know how many hours to schedule a week.
	maxNumberofStudents	Needed to calculate the necessary number of sections based on enrollment
Resource	name	Name of the resource
	usedBy	The sections currently using this room. Allows the system to determine the open time for this classroom.

Table 11 Final Attribute List for the Registrar Ontology

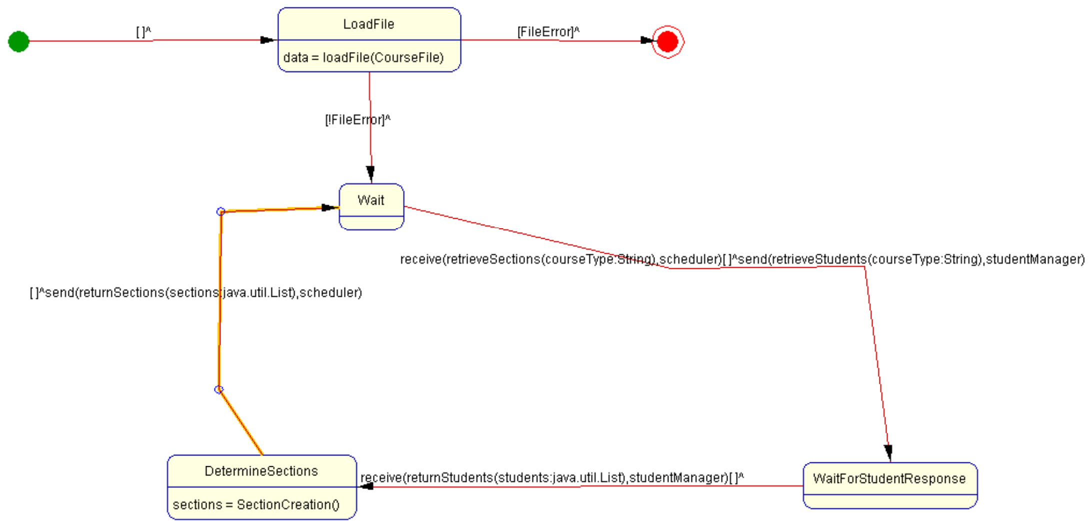
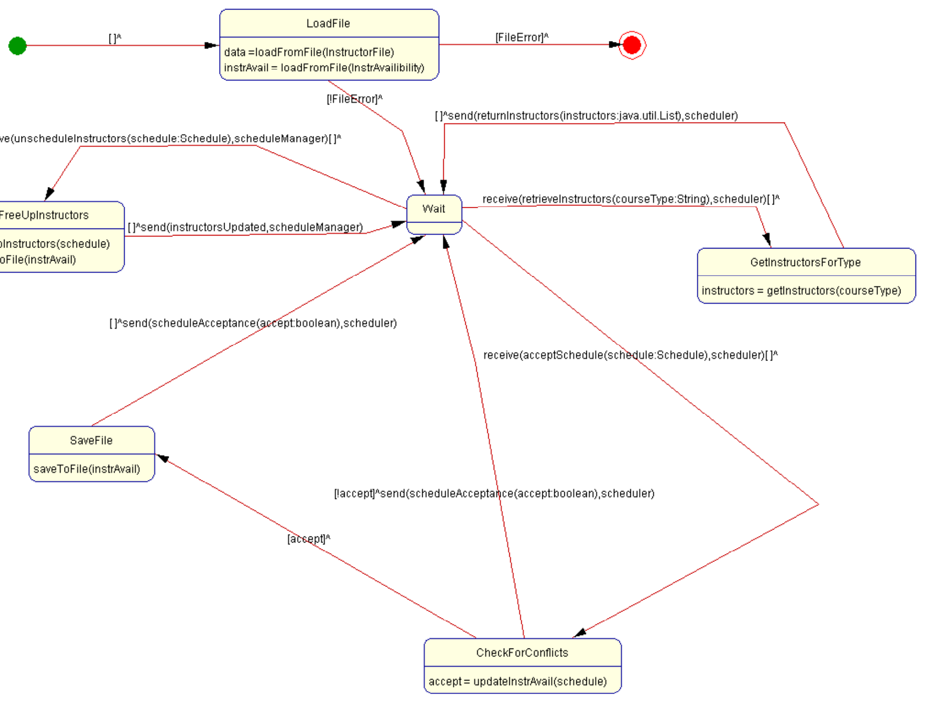

[ ]^
LoadFile
data = loadFile(CourseFile)
[FileError]^
[!FileError]^
Wait
receive(retrieveSections(courseType:String),scheduler)[ ]^send(retrieveStudents(courseType:String),studentManager)
[ ]^send(returnSections(sections:java.util.List),scheduler)
DetermineSections
sections = SectionCreation()
receive(returnStudents(students:java.util.List),studentManager)[ ]^
WaitForStudentResponse

Figure 42 ManageSectionInfo Task

A.4.2 InstructorManager

The *InstructionManager* role is responsible for retaining the instructor information and the availability of each instructor. The analyst assigned the *Verify Instructor-Course* goal because this role will handle the information that tells what instructors can teach which courses. The role must also ensure that instructors are never scheduled to be in two different places at the same time. To accomplish its goal, the role performs the *ManageInstructorInfo* task shown in Figure 43.

The task begins by loading the instructor information and availability from persistent storage. Once loaded, the role will respond to any one of three requests from the other roles. The first request is for the information about instructors for a specific course type. The manager constructs a list of all instructors qualified to teach courses of that type and returns the list. The second request is to free the resources used by a schedule. The manager uses the schedule to determine what instructors should be modified. The manager changes the instructors so that they are no longer unavailable during the times of classes in the schedule. The manager then saves the instructor availability to persistent memory and returns an acknowledgement to the role that sent the message.



LoadFile
data =loadFromFile(InstructorFile)
instrAvail = loadFromFile(InstrAvailibility)
[]^
[FileError]^
[!FileError]^
[]^send(returnInstructors(instructors:java.util.List),scheduler)
receive(unscheduleInstructors(schedule:Schedule),scheduleManager)[]^
FreeUpInstructors
freeUpInstructors(schedule)
saveToFile(instrAvail)
[]^send(instructorsUpdated,scheduleManager)
Wait
receive(retrieveInstructors(courseType:String),scheduler)[]^
GetInstructorsForType
instructors = getInstructors(courseType)
[]^send(scheduleAcceptance(accept:boolean),scheduler)
receive(acceptSchedule(schedule:Schedule),scheduler)[]^
SaveFile
saveToFile(instrAvail)
[!accept]^send(scheduleAcceptance(accept:boolean),scheduler)
[accept]^
CheckForConflicts
accept = updateInstrAvail(schedule)

Figure 43 ManageInstructorInfo Task

The final request asks the manager to approve a schedule. The manager checks for any conflicts with the instructors in the schedule and their current free-time, and returns whether the schedule is valid or not. If valid, the manager updates persistent memory to reflect the new schedule.

A.4.3 OutputManager

This role is responsible for handling the output of schedules in the sequence diagrams. This function pertains to all the sub-goals of the *Display Schedule* goal. The design includes a task for each of the sub-goals. Although one task could handle both the printing and the displaying to the monitor, different tasks are used to simplify the internal task states.

The *DisplaySchedule* task waits for a request to display a new schedule or an existing schedule to the monitor. If an existing schedule is requested, the task requests the schedule from the *ScheduleManager*. The schedule is formatted to fit on the screen and then displayed.

The *PrintSchedule* task waits for a request to print an existing schedule to the printer. The task requests the schedule from the *ScheduleManager*, formats it for the printer and then places the schedule in the print queue.

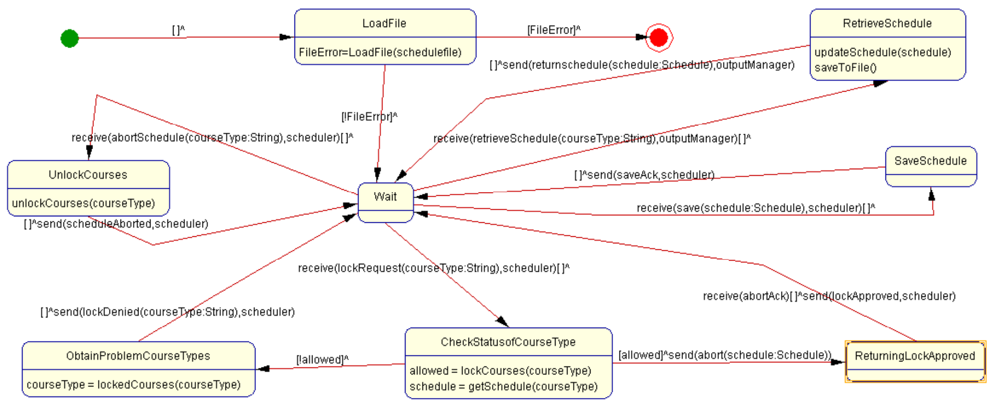
A.4.4 ResourceManager

The *ResourceManager* maintains all the information regarding the classrooms in the system. This function maps to the *Prepare Resources* goal. To maintain when the classrooms are available, the manager performs the *ManageSectionInfo* task. This task is analogous to the *ManageInstructorInfo* task, only the task handles classroom, instead of instructor information.

A.4.5 ScheduleManager

This role retains all accepted schedules in persistent memory and ensures that only one person is scheduling a course type at any one time. This role logically contains all the sub-goals of the *Manage Existing Schedule* goal. This research designed two tasks, as the freeing of resources is separate from storing and adding schedules.

The *ManageSchedules* task, shown in Figure 44, loads the schedules from persistent memory and then handles messages from the other roles. If the task receives a request for a schedule of a specific course type, the manager returns the appropriate schedule to the role. If the request is to save a schedule into the system, the manager adds the schedule to the master schedule and saves it to memory.



[ ]^
LoadFile
FileError=LoadFile(schedulefile)
[FileError]^
RetrieveSchedule
updateSchedule(schedule)
saveToFile()
[ ]^send(returnschedule(schedule:Schedule),outputManager)
[!FileError]^
receive(abortSchedule(courseType:String),scheduler)[ ]^
receive(retrieveSchedule(courseType:String),outputManager)[ ]^
SaveSchedule
UnlockCourses
unlockCourses(courseType)
[ ]^send(saveAck,scheduler)
Wait
receive(save(schedule:Schedule),scheduler)[ ]^
[ ]^send(scheduleAborted,scheduler)
receive(lockRequest(courseType:String),scheduler)[ ]^
receive(abortAck)[ ]^send(lockApproved,scheduler)
[ ]^send(lockDenied(courseType:String),scheduler)
ObtainProblemCourseTypes
courseType = lockedCourses(courseType)
[!allowed]^
CheckStatusofCourseType
allowed = lockCourses(courseType)
schedule = getSchedule(courseType)
[allowed]^send(abort(schedule:Schedule))
ReturningLockApproved

Figure 44 ManageSchedules Task Diagram

The task can also handle lock requests. Locks are used to ensure that only one agent in the system is scheduling a course type at any one time. Before scheduling a set of courses, the *Scheduler* must obtain a lock on that course type. The *ScheduleManager* checks if that course type is currently locked. If not, it approves the lock request. If a lock already exists, the manager returns that a lock on that course type already exists.

If after receiving a lock the *Scheduler* wants to cancel trying to schedule the course type, the role passes an abort message to the *ScheduleManager*. The manager updates its memory to show that that course type is no longer locked.

The *FreeScheduleResources* task returns all the resources used by a schedule into an available state. The task contacts the *InstuctorManager*, *StudentManager*, and *ResourceManager*; asking each to update their memory based on this schedule being removed from the system.

A.4.6 Scheduler

The *Scheduler* role is responsible for creating a schedule for a course type when requested. Thus, this role meets all goals related to the creation of a schedule and performs the *GenerateSchedule* task to meet all of these goals.

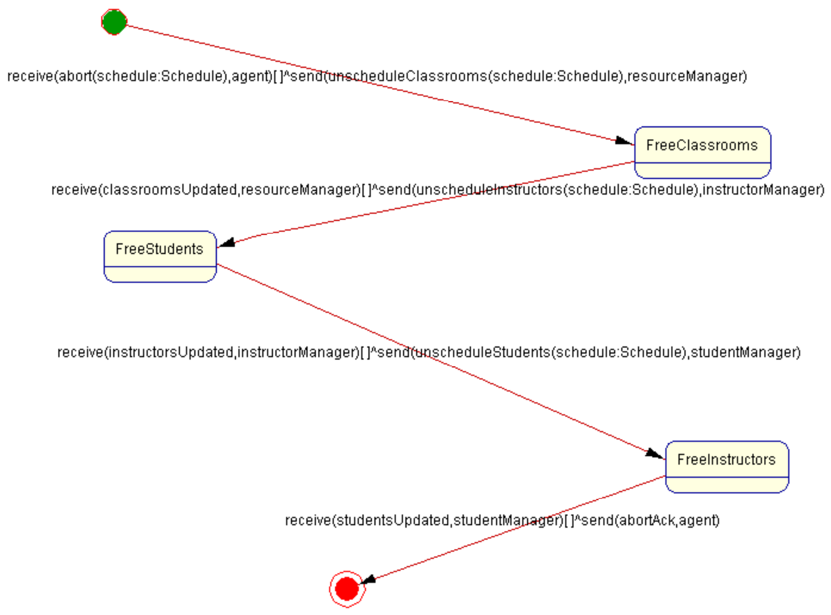

receive(abort(schedule:Schedule),agent)[ ]^send(unscheduleClassrooms(schedule:Schedule),resourceManager)
FreeClassrooms
receive(classroomsUpdated,resourceManager)[ ]^send(unscheduleInstructors(schedule:Schedule),instructorManager)
FreeStudents
receive(instructorsUpdated,instructorManager)[ ]^send(unscheduleStudents(schedule:Schedule),studentManager)
FreeInstructors
receive(studentsUpdated,studentManager)[ ]^send(abortAck,agent)

Figure 45 FreeScheduleResources Task Diagram

The *GenerateSchedule* task is the most involved of all the tasks in the system. The task waits for a request from the *User* to schedule a specific course type. The task obtains a lock for that course type from the *ScheduleManager*. If the lock is denied, the task notifies the *User* role and goes back to waiting. If the lock is granted, the task obtains the sections needed and all the information regarding the classrooms, students and instructors for courses of that type.

Now that the task knows all the information, it attempts to create a schedule. If the system cannot create a schedule, due to the existing schedules in the system, it notifies the *User* and aborts creating the schedule. If a schedule is found, the task asks the *InstructorManager*, *ResourceManger* and *StudentManager* to approve the schedule. If any of the managers do not approve the schedule, those managers that have approved it are notified to free up the resources used by the schedule and the task begins from the point where the lock was granted earlier.

Once a schedule is found and approved by the *manager* roles, the schedule is passed to the *OutputManager* to display to the user. The system then asks the user to accept the schedule. If accepted, the schedule is saved in the *ScheduleManger*. If not, the schedule is passed to the *FreeScheduleResources* task and the whole process starts over.

A.4.7 StudentManager

The *StudentManager* maintains all the information regarding the students in the system, which maps to the *Prepare Resources* goal. To maintain when the classrooms are available, the manager performs the *ManageSectionInfo* task. This task is analogous to the *ManageInstructorInfo* task, only the task handles classroom, instead of instructor information.

A.4.8 User

The *User* role is responsible for handling all inputs from the user and translating them into the proper requests to the various roles. The *Allow User Input* goal is met by the *HandleUserInput* task. This task waits for user input, parses it and performs the appropriate system call.

A.5 Creating Agent Classes

The next step involves taking the roles and turning assigning them to agent classes. For this project, the design uses one agent class for each of the manager roles. The design combines the roles of *User*, *Scheduler* and *OutputManager* into a single agent class. These roles reside on the same system to reduce network traffic and so including them in the same agent class reduces the number of agents in the system. The *InstructorManager*, *StudentManager*, *ResourceManager* and *ScheduleManager* will reside on separate systems and must have separate agent classes. Figure 46 shows the agent classes and their assigned roles.

A.6 Constructing Conversations

Figure 46 also shows the conversations between the agent classes. This research constructed these conversations using the tasks for each role. In the tasks, the role sends and receives messages. Once the design assigns a role to an agent class, these messages become internal or external messages. For every *send* in a task diagram, the designer finds the corresponding *receive* transition. If the roles reside in the same agent class, no conversation is needed. If the roles are in separate agent classes, the designer creates a new conversation.

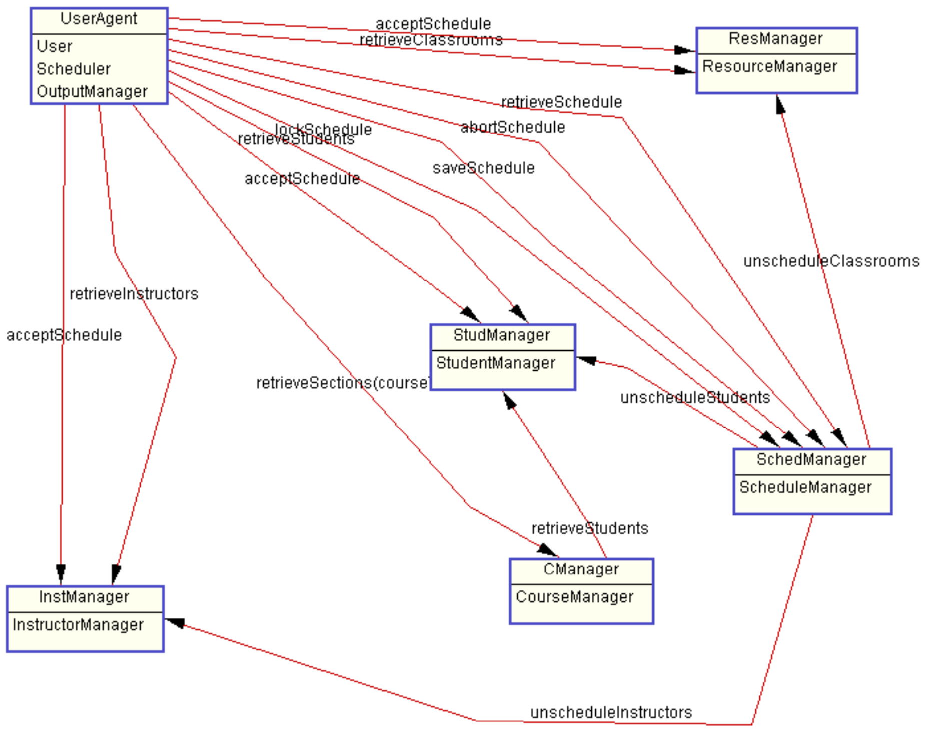


Figure 46 Agent Template Diagram

For example, in the *FreeScheduleResources* task there are four send messages. The first three are to the *InstructorManager*, *StudentManager* and *ResourceManager*, which are all in separate classes. A conversation exists for each of these sends: *unscheduleStudents*, *unscheduleInstructors* and *unscheduleClassrooms*. The *SchedManager* is the initiator for each of these conversations, as the agent starts off the conversation by requesting that the other agent free up the resources used by that schedule. The other agent replies once complete and the conversation is finished. In this system, most conversations are: send a message and wait for the response.

Some conversations, such as *abortSchedule*, are more complex. After the start of the conversation, the responder must start additional conversations, as shown in Figure 47. This conversation accounts for the last send from the *FreeScheduleResources* task. When asked to abort the passed-in schedule, the agent starts the three conversations described earlier to free the instructor, student and classroom time. Once the conversations are complete, the agent responds that the schedule has been aborted.

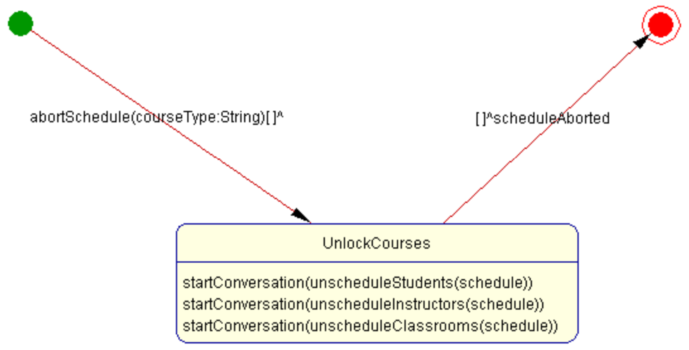

abortSchedule(courseType:String)[ ]^
[ ]^scheduleAborted
UnlockCourses
startConversation(unscheduleStudents(schedule))
startConversation(unscheduleInstructors(schedule))
startConversation(unscheduleClassrooms(schedule))

Figure 47 abortSchedule Conversation Responder Diagram

Using the procedure of matching *sends* to *receives* and ensuring every message sent to a task located in a separate agent class occurs in a conversation, the designer specifies every needed conversation.

A.7 Assembling Agent Classes

The designer must now specify the agent behavior by describing the internal components of each of the agent classes. As the conversations came from the tasks, so do the internal components. This system uses the typical MaSE design of having one component for each of the tasks an agent performs. Each component is then responsible for accomplishing the task it derives from.

This simplifies the design of the agents by compartmentalizing the agent behavior into parts that contain a single task to accomplish. Without separating the tasks, the designer must integrate the behavior of the two tasks into one single thread of control. This is a difficult and unnecessary process. With each component performing a single task, the designer can easily transform the task behavior into the component behavior. Now, however, the component starts specific conversations instead of sending messages.

Figure 48 is the component state diagram for the *FreeScheduleResources* component of the *SchedManager* agent and shows how the task is transformed into the component behavior. In the task diagram, the role received a request and then sent a message to each of the managers. These send messages are included in conversations, so when the agent receives the request, the agent starts the conversations and then completes the initial conversation.

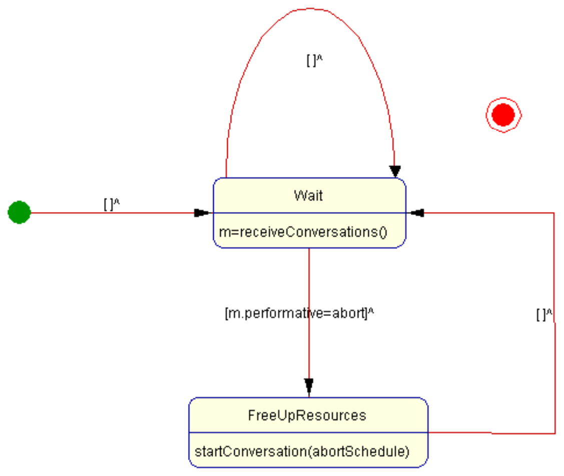


Figure 48 FreeScheduleResources Component State Diagram

The designer specifies the attributes and functions of each component after completing the state diagrams for the component. These methods come from the actions performed in the state diagrams. In the *ManageSchedules* task, one state has the action *allowed = lockCourses(courseType:String)*. This action became part of the *lockSchedule* conversation, started by the *ManageSchedule* component of the *SchedManager* agent. As such, the component must contain an attribute *allowed* of type Boolean and a function *lockCourses* that returns a Boolean and receives a String as a parameter. This project completes this step by specifying the behavior, attributes, and methods of all system components.

A.8 System Design

The system design assigns agents to different computing platforms on the network. The distributed course scheduling system has the design of one instance of each agent, except the *UserAgent* that has multiple instances. Each agent resides on its own platform, as the system requirements dictated that each of the resources could be on a separate computer. Since each agent is responsible for one of those resources, the agents must be on different platforms. Once the agents are placed on platforms, the communication between platforms flows from the conversations between the agents.

With all of the MaSE steps complete, the analysis and design documents describe the system's behavior, architecture and information domain. The plans describe the agent classes, conversations and their location in the system, providing the details necessary to code the multiagent system.

B. Experimental Raw Data

This section contains the raw data from the experiments on the Geometric Score Reduction Model. Each table has the object tested on, the correct object for mapping and the position in the ranked list of that object.

B.1 Mapping Same-Size Ontologies

This section lists the results from the mapping of the Trias, Day and Rapallo version of the Registrar Ontology to that of Harris, Jensen and Kil.

B.1.1 Best-Value

From	To	Rank	Time (ms) to Rank 1,000 Times
Thing	Thing	1	145
Advisor	Advisor	9	1032
AFITform51	Form51	8	2824
AFITform69	From69	10	611
AFITGradList	GraduationList	2	841
Course	Course	13	2258
Database	Database	1	140
Dean	Dean	11	375
Department	Department	1	1452
DepartmentHead	DepartmentHead	11	626
DeptCatalog	DeptCatalog	21	1442
DeptGradList	GraduationList	1	846
Edplan	edplan	1	2599
Grade	GradeList	2	831
GradProgram	GradProgram	2	2528
Instructor	Instructor	12	611
QuarterCourseOffering	CoursesOfferedList	6	1081
Registrar	Registrar	8	1032
Requirements	Requirements	4	641
Schedule	ClassSchedule	1	646
Student	Student	6	1743
	AVG=	6.238095	1157.333333
	STD=	5.448897	802.4071494

B.1.2 Strict Comparator

From	To	Rank	Time (ms) to Rank 1,000 Times
Thing	Thing	1	281
Advisor	Advisor	10	385
AFITform51	Form51	5	621
AFITform69	From69	6	320
AFITGradList	GraduationList	2	340
Course	Course	2	517
Database	Database	1	284
Dean	Dean	6	330
Department	Department	1	443
DepartmentHead	DepartmentHead	5	350
DeptCatalog	DeptCatalog	25	425
DeptGradList	GraduationList	2	344
Edplan	edplan	2	520
Grade	GradeList	16	331
GradProgram	GradProgram	4	596
Instructor	Instructor	12	351
QuarterCourseOffering	CoursesOfferedList	3	390
Registrar	Registrar	8	393
Requirements	Requirements	20	320
Schedule	ClassSchedule	2	351
Student	Student	14	441
	AVG=	7	396.8095238
	STD=	6.80441	95.85855155

B.1.3 Loose Comparator

From	To	Rank	Time (ms) to Rank 1,000 Times
Thing	Thing	1	250
Advisor	Advisor	15	331
AFITform51	Form51	9	553
AFITform69	From69	11	272
AFITGradList	GraduationList	5	301
Course	Course	5	461
Database	Database	1	245
Dean	Dean	16	278
Department	Department	1	362
DepartmentHead	DepartmentHead	15	294
DeptCatalog	DeptCatalog	9	375
DeptGradList	GraduationList	6	301
Edplan	edplan	5	454
Grade	GradeList	1	292
GradProgram	GradProgram	4	481
Instructor	Instructor	14	293
QuarterCourseOffering	CoursesOfferedList	7	328
Registrar	Registrar	15	326
Requirements	Requirements	5	286
Schedule	ClassSchedule	3	294
Student	Student	8	377
	AVG=	7.428571	340.6666667
	STD=	5.143651	82.66579301

B.1.4 Alphabetical

From	To	Rank	Time (ms) to Rank 1,000 Times
Thing	Thing	25	30
Advisor	Advisor	1	40
AFITform51	Form51	13	30
AFITform69	From69	14	30
AFITGradList	GraduationList	19	30
Course	Course	4	30
Database	Database	6	40
Dean	Dean	7	30
Department	Department	8	30
DepartmentHead	DepartmentHead	10	30
DeptCatalog	DeptCatalog	9	30
DeptGradList	GraduationList	13	30
Edplan	edplan	19	30
Grade	GradeList	15	30
GradProgram	GradProgram	17	30
Instructor	Instructor	20	30
QuarterCourseOffering	CoursesOfferedList	5	30
Registrar	Registrar	22	30
Requirements	Requirements	23	30
Schedule	ClassSchedule	2	30
Student	Student	24	40
	AVG=	13.14286	31.42857143
	STD=	7.525196	3.585685828

B.2 Mapping Smaller to Larger Ontologies

This test involved mapping objects from the genealogy-based ontology at http://orlando.drc.com/daml/Ontology/Genealogy/current/ to the general concept ontology found at http://www.cs.umd.edu/projects/plus/DAML/onts/general1.0.daml. The following tables show the results using each of the versions of the ranking model.

B.2.1 Best-Value

From	To	Rank	Time (ms) to Rank 1,000 Times
Thing	SHOEEntity	1	120
Person	Person	1	1733
Individual	Person	7	2483
PhysicalAddress	Address	1	938
Family	SocialGroup	2	752
Event	Event	1	431
	AVG=	2.166667	1076.166667
	STD=	2.401388	879.2358993

B.2.2 Strict Comparator

From	To	Rank	Time (ms) to Rank 1,000 Times
Thing	SHOEEntity	1	210
Person	Person	1	581
Individual	Person	7	851
PhysicalAddress	Address	1	310
Family	SocialGroup	1	361
Event	Event	1	260
	AVG=	2	428.8333333
	STD=	2.44949	243.5137915

B.2.3 Loose Comparator

From	To	Rank	Time (ms) to Rank 1,000 Times
Thing	SHOEEntity	1	185
Person	Person	1	476
Individual	Person	7	636
PhysicalAddress	Address	1	285
Family	SocialGroup	1	285
Event	Event	1	245
	AVG=	2	352
	STD=	2.44949	169.9105647

B.2.4 Alphabetical

From	To	Rank	Time (ms) to Rank 1,000 Times
Thing	SHOEEntity	22	36
Person	Person	18	35
Individual	Person	18	35
PhysicalAddress	Address	2	35
Family	SocialGroup	23	35
Event	Event	10	36
	AVG=	15.5	35.33333333
	STD=	8.043631	0.516397779

Bibliography

ARPA Knowledge Sharing Initiative. "Specification of the KQML agent-communication language," ARPA Knowledge Sharing Initiative, External Interfaces Working Group working paper. Available as http://www.cs.umbc.edu/kqml/papers/kqml-spec.ps, December 1992.

Berners-Lee, T., R. Fielding, and L. Masinter. "Uniform Resource Identifiers (URI)," Generic Syntax. IEDTF Draft Standard (RFC 2396), August 1998.

Crawford, J and Kuipers, B. "Toward a theory of acess-limited logic for knowledge representation," Proceedings of the First International Conference on Principles of Knowledge Representation. Morgan Kaufmann, 1989.

DAML+OIL. DAML ontology library. http://www.daml.org/ontologies

DeLoach, S., M. Wood, and C. Sparkman. "Multiagent Systems Engineering," The International Journal of Software Engineering and Knowledge Engineering. Volume 11 no. 3, June 2001.

DeLoach, S. "Analysis and Design using MaSE and agentTool," 12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001). Miami University, Oxford, Ohio, March 31 - April 1, 2001.

DeLoach, S. and M. Wood. "Developing Multiagent Systems with agentTool," Intelligent Agents VII. Agent Theories, Architectures, and Languages – 7th International Workshop, ATAL-2000, Boston, MA. July7-9, 2000. Lecture Notes in Artificial Intelligence. Springer- Verlag, Berlin, 2001.(2000a)

DeLoach, S. and M. Wood. "Multiagent Systems Engineering: the Analysis Phase." Technical Report, Air Force Institute of Technology, AFIT/EN-TR-00-02, June 2000. (2000b)

Evans, R., P. Kearney, J. Stark, G. Caire, F. Garijo, J. Gomez Sanz, F. Leal, P. Chainho, and P. Massonet. "MESSAGE: Methodology for Engineering Systems of Software Agents." EURESCOM Project P907. September 2001.

Ferber, J. Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Harlow, England: Addison-Wesley, 1999.

Fernández, M., A. Gómez-Pérez, and N. Juristo. "METHONTOLOGY: From Ontological Art Towards Ontological Engineering." Ontological Engineering: Papers from the 1997 AAAI Spring Symposium. Technical Report SS-97-06, AAAI Press, 1997.

Fernández, M., A. Gómez-Pérez, J. Sierra, and A. Sierra. "Building a Chemical Ontology Using Methontology and the Ontology Design Environment." IEEE Intelligent Systems, Vol.14, No.1, pp.37-46, 1999.

FIPA TC B. "FIPA Agent Management Specification." http://www.fipa.org/specs/fipa00023/. August 15, 2001.

Fikes, R. and T. Kehler. "The role of frame-based representation in reasoning." Communications of the ACM, 28(9):904-920, 1985.

Genesereth, M. "The Epikit manual." 1990.

Genesereth, M. "Knolwedge Interchange Format: draft proposed American Standard." NCITS.T2/98-004. 1998.

Georgeff, M., D. Kinny, and A. Rao. "A Methodology and Modeling Technique for Systems of BDI Agents," in Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW '96. Lecture Notes in Aritificial Intelligence, vol. 1038. Speringer-Verlag, Berlin Heidelberg, 1996.

Gómez-Pérez, A. "Some ideas and examples to evaluate ontologies." Proceedings of the Eleventh Conference on Artificial Intelligence Applications. IEEE Computer Society Press, 1995.

Gómez-Pérez, A. "Guidelines to verify completeness and consistency in ontologies." Third World Congress on Expert Systems. 1996.

Gruber, T. "Toward Principles for the Design of Ontologies Used for Knowledge Sharing." IJHCS,43(5/6):907-928.

Guarino, N. "Understanding, Building, and Using Ontologies." Technical report, LADSEB-CNR, National Research Council, 1996.

Gruber, T. "Ontolingua: A Mechanism to Support Portable Ontologies." 1992.

Gruninger, M. and M. Fox. "Methodology for the design and evaluation of ontologies." Workshop on Basic Ontological Issues in Knowledge Sharing. International Joint Conference on Artificial Intelligence, 1995.

Hall, M. GEDCOM data model. http://orlando.dr.com/daml/Ontology/Genealogy/current/. 2001

Heflin, J. General concept ontology. http://www.cs.umd.edu/projects/plus/DAML/onts/general1.0.daml. 2000

Huhns, M. and M. Singh. "Ontologies for Agents." IEEE Internet Computing, November-December 1997. 81-83.

KBSI. "The IDEF5 Method Report." KBSI Report, 1994, Texas.

Kendall, E. A., Palanivelan, U, and Kalikivayi, J. "Capturing and Structuring Goals: Analysis Patterns," Proceedings of the Third European Conference on Patter Languages of Programming and Computing, Bad Irsee, Germany, July 1998.

Lenat, D. and R. Guha. Building Large Knowledge-based Systems: Representation and Inference in the Cyc Project. Addison-Wesley, 1990.

MacGregor, R. "The evolving technology of classification-based knowledge representation systems." Principles of Semantic Networks: Explorations in the Representation of Knowledge. Morgan Kaufmann, San Mateo, CA. pp.385-400, 1991.

Noy, N. and D. McGuinness. "Ontology Development 101: A Guide to Creating Your First Ontology." 2001

Noy, N., M. Sintek, S. Decker, M. Crubezy, R. W. Fergerson, and M. A. Musen. "Creating Semantic Web Contents with Protege-2000." IEEE Intelligent Systems 16(2):60-71, 2001

Ontolingua. Online ontology library. http://ontonlingua.stanford.edu

Russel, S. and P. Norvig. Artificial Intelligence: A Modern Approach. Prentice-Hall, Inc. New Jersey, 1995.

Shelton, H. "Joint Vision 2020". US Government Printing Office. Washington, DC, 2000.

Skuce, D. "Conventions for reaching agreement on shared ontologies." Proceedings of the 9th Knowledge Acquistion for Knowledge Based Systems Workshop, 1995.

Sparkman, C. Transforming Analysis Models Into Design Models for the Multiagent Systems Engineering (MaSE) Methodology. MS thesis, AFIT/GCS/ENG/01M-12. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, 2001.

Uschold, M. and M. Gruninger. "ONTOLOGIES: Principles, Methods and Applications." Knowledge Engineering Review. Volume 11 Number 2, June 1996.

W3C. "Extensible Markup Language (XML) 1.0". W3C Recommendation. Feb, 1998.

W3C. "Resource Description Framework (RDF) Model and Syntax Specification". W3C Consortium Recommendation. 1999

Wooldridge, M., N. Jennings, and D. Kinny. *The Gaia methodology for agent-oriented analysis and design*. Journal of Autonomous Agents and Multi-Agent Systems, vol. 3(3), 2000.

VITA

First Lieutenant Jonathan M. DiLeo was born on 3 June 1976 in Hahn, Germany. He graduated from the International Baccalaureate Program of Saint Petersburg High School in Saint Petersburg, Florida in June 1994. He entered undergraduate studies at Duke University, Durham, North Carolina where he graduated with a Bachelor of Science in Computer Science in May of 1998. He was commissioned through the Detachment 585 AFROTC at Duke University where he was recognized as a Distinguished Graduate.

In July 1998, Lt DiLeo attended the Basic Communications and Information Officers Training Course at Keesler AFB, Mississippi. Upon graduating as a Distinguished Graduate, he was assigned to the 6th Communications Squadron, MacDill AFB, Florida. He served as the Squadron Section Commander and Chief of the Network Control Center. In August 2000, he entered the Graduate School of Engineering and Management, Air Force Institute of Technology. Upon graduation, he will be assigned to the College of Aerospace Doctrine, Research and Education, Maxwell AFB, Alabama.