Extensible Multi-Agent System for Heterogeneous Database Association Rule Mining and Unification.

THESIS

Presented to the faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Systems)

Christopher G Marks, B. S.

Captain, USAF

March, 1999

Acknowledgments

Table of Contents

Table of Figures

Extensible Multi-Agent System for Heterogeneous Database Association Rule Mining and Unification

1 Introduction

With the ever-increasing availability of information, the methods of encoding and storing the information grows as well [KA97][SING98]. Available information sources include traditional databases such as relational, flat files, knowledge bases, programs, object-oriented, text documents, HTML, and proprietary formats that are some variant of a traditional format [KA97]. As the number of information sources grows, the problem of how to combine these distributed, heterogeneous data repositories becomes more and more critical. Even within one organization or company, this information can be stored in separate geographic locations and in varying formats. When this is coupled with rising storage capacities and the dropping cost of gathering information, we are left with an overwhelming amount of data.

One method of extracting useful trends from data is through data mining *association rules*. In this research we present a methodology and a tool for mining association rules from multiple heterogeneous data sources and then unifying the results for future incorporation into a knowledge base.

The main purpose of this research is to provide an extensible architecture that provides flexibility in the addition of a data source for data mining operations. Data mining and association rules are reviewed first. The relatively new field of Agents and multi-agent systems is then reviewed. A methodology for developing a multi-agent system is presented in Chapter 3.

Application of this methodology to the problem presented here is detailed in Chapter 4. Chapter 5 discusses and provides an implementation of extending the system for a new data source. Finally, conclusions and future work is discussed in Chapter 6.

1.1 Background

Research by Capt. Daniel Stein has shown how data mining association rules could be used to repair knowledge base incompleteness. This work was in support of the *Probabilities, Experts System, Knowledge, and Inference* (PESKI) System, an integrated framework for expert system development [STEIN96]. This system utilizes a knowledge representation known as a *Bayesian Knowledge Base (BKB)* to provide flexibility and an ease of understanding that is lacking in many representation schemes [STEIN96]. Stein has shown how a goal-directed data mining approach can automate the process of automatically solving incompleteness in a BKB with the implementation of DBMiner. Association rules can be used to discover one or more relationships that are missing from a BKB without human intervention. To date, this implementation supports only one format of knowledge base and is thus limited.

PESKI is the physical realization of an integrated knowledge-based system framework that combines the functions of natural language interface, inferencing, explanation and interpretation, and knowledge acquisition into a single consolidated application [STEI6][STEI20]. As mentioned above, the knowledge representation scheme used for PESKI is the BKB. BKBs utilize Bayesian probabilities to represent the statistical causal relationship of one random variable to another. Figure 1 represents a single piece of information in BKB format. It represents the fact that given it is sunny, there is a 75% chance the sidewalk is dry. The PESKI System utilizes this representation for inferencing over the knowledge base.

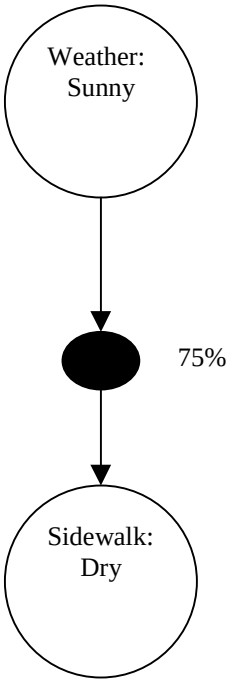

Figure 1 Sample BKB Relationship Representation

1.2 Problem Statement

This work addresses an architecture that not only allows mining data sources of multiple formats, but also allows extensibility for future formats. It also presents a methodology for unification of association rules prior to incorporation in a BKB. It will use an existing agent development tool to establish a multi-agent based framework and define the communications between those agents. The framework will be designed to accept a request from PESKI for one of three possible data mining operations. These operations are discussed in more detail in Chapter 4. Once the system accepts the request, it will determine which data sources can fulfill the request and tasks the agents responsible for those sources to begin data mining. Once results have been obtained, they will be unified to eliminate redundant or conflicting results and returned to PESKI. Two new data source formats will be introduced into the PESKI schema and they will be mined for association rules, the results unified into a unique list of results, and then passed back to PESKI for incorporation into the BKB. The process will be automated and will use

existing message passing formats to communicate with PESKI. One limiting assumption is that we will mine association rules, based on the research by Capt. Stein, as opposed to other methods of acquiring necessary support conditions for the states.

1.3 Thesis Overview

Chapter 2 provides a review of literature and research that is relevant to the problem including data mining, agents, and existing information retrieval systems. In Chapter 3, a methodology is specified that moves from problem definition to detailed design. The application of the agent development framework and methodology to the problem presented here is covered in Chapter 4. Chapter 5 discusses how the resulting architecture could be extended to include other data mining algorithms and data source formats and includes an implementation of anew source. Finally, Chapter 6 looks at the conclusions of the research and future work that may be accomplished.

2 Background

2.1 Overview

This chapter reviews the different technologies and literature that is related to this research. Section 2.2 provides a detailed explanation of what *data mining* and *association rules* are as well as how data mining association rules is accomplished in general. This research is focused on how this can be accomplished over heterogeneous data sources, which are described in Section 2.3, along with some of the problems inherent in data mining heterogeneous sources. Section 2.4 defines the term *agent*, and details various agent attributes and the categories of agents most commonly used. The next section reviews existing projects that focus on the use of agents for information retrieval tasks over heterogeneous data sources. The common architecture used by these projects is outlined and discussed in Section 2.6. Finally, Section 2.7 covers several Java-based agent development frameworks.

2.2 Data Mining

Data mining is a broad term that describes the search to extract some meaningful information from data that is unformatted and either unstructured or partially structured [RA95]. Similarly, Fayyad et. al. described it as "The nontrivial process identifying valid, novel, potentially useful, and ultimately understandable patterns in data" [FU95]. Data mining is also known as knowledge discovery, knowledge extraction, information harvesting, data archeology, and data pattern processing. Although most algorithms provide some unique implementation of each phase, there are several common steps to achieve the goal of identifying patterns in data.

The first step in data mining is data cleaning, or pre-processing. All input data must meet certain conditions to ensure optimal performance including:

1. The data must be in a usable form.

2. There must be sufficient data to derive a solution.

The next step is data reduction. Data reduction eliminates those variables that are not of interest to the problem domain. Variables that are not of interest are termed '*non-useful*' variables. The data mining process is time consuming and eliminating such non-useful variables may provide some speedup. The third step is to choose a data mining *goal*. The goal of the data mining process is largely based on the application in which the results will be used. Some typical application goals include classification, regression, clustering, and summarization [FU95]. This background and the subsequent implementation will focus on *link analysis* or *association rule* data mining (see Section 2.2.1).

Once a goal is chosen, a data mining algorithm must be selected. Selecting the methods used for searching for the patterns is critical. There are different and more efficient algorithms depending on the goal, as well as the format of the data. The next step is to perform the actual data mining. This is simply executing the algorithm chosen on the processed data. Finally, once the data is mined, the mined patterns must be interpreted. This may include a return to previous steps to refine the results or focus the search on other areas.

One popular goal of this process is to find trends in the data that show associations between domain elements. This is generally focused on transactional data such as a database of purchases at a store. This goal is known as association rules and is described in more detail next.

2.2.1 *Association Rules* An association shows some relationship between two values in the form of an implication ($X \Rightarrow Y$). An association rule is an association in which one or more items in the antecedent (X) of an implication is correlated with one or more items in the consequent (Y) with some acceptable level of confidence and support [SA96]. The support for

the rule $X \Rightarrow Y$ is the conditional probability that a transaction (database entry) contains X, given that it contains Y. The confidence is the percentage of all transactions with X that also include Y. An example of this type of rule is the statement that 90% of transactions in which chips and dip were purchased, soda was also purchased, and 3% of all transactions contain all three items. The antecedent of this rule (X) consists of chips and dip and the consequent (Y) is soda. The 90% represents the confidence factor of this rule and the 3% is the support for the rule. The rule can then be specified as chips $\cap$ dip $\Rightarrow$ soda. Both the antecedent and consequent can have sets of items, or can be a single item.

The algorithms for mining association rules generally follow three main steps [SA95]. First, the database is scanned for all *itemsets*, or sets of items whose support is greater than some user specified minimum. Those itemsets meeting the minimum support are called *frequent itemsets*. The second step in most algorithms is to use the frequent itemsets to generate the desired rules based on confidence levels. This can be accomplished by breaking the itemsets into their individual components and establishing relationships between them. The general idea is that if ABC are frequent items, then it can be determined that $AB \Rightarrow C$ if the support for the relation meets the minimum support and confidence levels. Finally, all *uninteresting* rules are pruned (removed) from the resulting rules. In the context of data mining, uninteresting refers to any rule that the user or expert system does not need or is not useful. These steps outline a general approach, and as such, there are some common problems.

Despite the relatively straightforward mining of association rules and even after pruning, the usefulness of the results is sometimes questionable. This is largely due to two main reasons [SA96]. First, most association rule mining algorithms use generic criteria to prune uninteresting rules. They do not consider the domain of the problem and can eliminate potentially important or useful rules. Second, rules are presented in a disjoint manner, without regard to relationship

between them. This can place the burden of finding the truly useful rules on the user. Again, this is related to the problem of domain independence.

Research focusing on more specific algorithms shows how this general process can be refined to eliminate some of these problems and help optimize an otherwise I/O intensive process. There are algorithms that focus on the types of association rules mined, as well as those focused on the format of the database. This process is focused on the mining of an individual data source. Unfortunately within domain of interest the data sources may be of varying formats and different locations. This is discussed along with the impact on the traditional data mining theory.

2.3 Heterogeneous Sources

As mentioned before, the growing availability of information has led to multiple formats and methods of encoding and storing the information. The decision on which format to use is largely based on the needs of the users and the structure of the data. Data mining becomes an issue as the associations that exist in the various heterogeneous sources may be of interest. More importantly, if the data in the various sources is related, the association rules mined may be related and can be integrated.

Research has uncovered several problems with the integration of distributed sources in similar, as well as heterogeneous formats. Singh mentions several of these that must be considered when unifying such data [SING98]:

1. Different sources can use different words for the same object.
2. Different sources can use different words for similar concepts.

3. Information is typically created to serve a local purpose and often omits parts that are always the same in the local context. This information is often essential to remove ambiguity at a higher level than the local source.

The problem associated with integration of the data from heterogeneous sources has driven a multitude of projects. One promising approach is to provide access to a large number of information sources organized into a network of information agents [KA97]. By evaluating agents and the proposed uses in data mining, we can get a better idea of how they can be used to solve the problems presented above.

2.4 Agents

The term *agent* has been used to describe a multitude of software - from simple batch processing to systems displaying intelligence, social ability, and pro-activeness [BRAD98]. Because there is a lack of standard definition, any research surrounding agents must clearly spell out how they define the term within the scope of that research. There are some generally agreed upon definitions or qualities of an agent that are discussed in Section 2.4.1, but even they are subject to debate. This in turn leads to question over the definition of individual agent types as well. To avoid confusion, the agent classes are discussed in Section 4.2.2 and the agent type definitions used by this research are explained in Section 2.4.3. Finally, Section 2.4.4 provides an overview of agent communication and the importance of *speech acts*.

2.4.1 *Definition* It is because agents are relatively new and encompass a wide variety of work that it is difficult for a standard definition to be agreed upon. One researcher went as far as saying [NWA96]: “We have as much chance of agreeing on a consensus definition for the word agent as AI researchers have of arriving at one for artificial intelligence itself - nil!”

A definition many researchers find acceptable is one provided by Shoham. Shoham defines an agent as a software entity that continuously and autonomously operates in an environment that may be occupied by other agents and processes [SHO97]. This is more accepted partly because it is a very high level and general definition. More detailed definitions such as the one presented next are the subject of the debates surrounding agents.

Woolridge and Jennings distinguish two general uses of the term agent: one is a weak usage, the other stronger and potentially more contentious [WJ95]. They contend a hardware or software based computer system with four key properties can be weakly classified as an agent. First it must be *autonomous*. An autonomous agent can operate without the direct intervention of humans or others, while exercising some kind of control over their actions and internal state. The internal state and goals should drive the agent to move its autonomous actions towards completion of the users or systems goals.

Next is s*ocial ability* or the ability to interact with other agents (or humans) by way of some agent-communication language. Nwana claims this cooperation is “paramount: it is the *raison dietre* for having multiple agents” [NWA96]. Without cooperation or communication, the benefit of having multi-agent systems is lost. Third, an agent must be *reactive*. A reactive agent can perceive their environment (which can be the physical world, a user through a graphical user interface, a collection of other agents, the Internet, or some combination of these), and respond in a timely fashion to changes that occur. The last quality is *proactiveness*. By being proactive an agent does not simply act in response to environmental changes, but is able to exhibit goal-directed behavior by taking some initiative.

Some have classified this attribute as a part of autonomy and do not consider it unique [NWA96]. This weak notion of agency is used as the basis for each project presented in this Chapter. The properties are not strict guidelines to base agent classification on however, as Dr. Hyacinth Nwana wrote, these are attributes "which agents *should* exhibit" [NWA96]. As mentioned, there is a more contentious, stronger notion of agency, sometimes termed 'secondary attributes'.

In this stronger notion of agency, it is quite common to characterize an agent using mentalistic notions, such as knowledge, belief, intention, and obligation as well as emotional aspects [SHO97][BATES]. Characterizing an agent using this stronger notion can include properties such as mobility, veracity, benevolence and rationality.

One of the more common secondary attributes is that of *mobility*. Mobility is the ability of an agent to move around in an electronic network, whether it is the Internet or a LAN [BRAD97]. *Veracity* is the assumption that an agent will not knowingly communicate false information. It is often useful to instill this attribute in a closed system of agents. Next is *benevolence*, the assumption that agents do not have conflicting goals and will perform the tasks asked of it. Finally is the attribute of *rationality*, or the assumption that an agent will act in order to achieve it goals and will not act in a way that would prevent its goals from being achieved.

These attributes are not an exhaustive guide to agent attributes, but display the wide range of potential attributes and traits any single agent can have. Various research may define agents differently though they seem to perform similar tasks. As agents are developed that have one or some of these traits, general classes begin to form based on which traits are used.

2.4.2 *Classes of Agents* Agents that use one or more of the traits above have begun to be grouped into high level classes. These "Agent Classes" are not agreed upon standards, but

rather commonly used classifications. The following detail the more common classifications according to Nwana [NWA96]

Most widely accepted are *Autonomous Agents.* These agents can sense and act autonomously in an environment. Although they are autonomous, their actions work towards a goal. The environment can be simple and static or complex and dynamic [WJ95].

Information Agents are agents that can access, retrieve, and manipulate information obtained from any number of information sources. They also can answer queries about the information that they can access [WJ95].

Another common agent is an *Intelligent Agent.* These are agents that act on the behalf of the user or another program to carry out a set of operations. They do so with some degree of independence and autonomy.

Interface Agents are agents that support and provide assistance to a user through observing and monitoring the users actions in an interface. The agent learns from the actions and suggests or implements more efficient or easier ways of accomplishing tasks.

Collaborative Agents rely on the social ability of agents in any system to cooperate and autonomously perform tasks for the user. They have some common interface language in order to cooperate and communicate with other agents.

Finally, *Mobile Agents* are capable of movement between computers across a local area network (LAN), wide-area network (WAN), or any other communication medium. Typically they gather information for a user and report results either by traveling back to the user or transmitting them

Again, this is not an exhaustive list of agent classes, but rather some of the most widely used and agreed upon generalizations. Some other classes of agents which are not explicitly covered here are hybrid agents, reactive agents, behavioral agents, and entertainment agents

[BRAD97]. Additionally, this list is not mutually exclusive. For instance, a mobile agent can be intelligent and collaborative as well.

2.4.3 *Information Retrieval Agents* To this point this Chapter has reviewed the definition of an agent in Section 2.4.1 providing four properties which can define an agent, as well as several secondary attributes. Different combinations of these properties yield several classes of agents that are discussed in Section 2.4.2. Even within these classes there exist multiple agent types. This section covers those that can be grouped under the *information agent* class.

Information agents are agents whose goal is "to provide information and expertise on a single topic by drawing on relevant information from other information agents" [KAH94]. Systems designed using such agents allow an abstraction of each heterogeneous source to be made and a common interface defined [KAH94]. The projects discussed next in Section 2.4 use these agents or forms of these agents in different architectures to perform heterogeneous information retrieval. Because of the multitude of possible definitions that exist in literature, the agent definitions presented below will be considered standard for this research. This list is far from complete and others have mentioned several other agents and offer different definitions for these general categories.

A *User Agent* accepts queries by the user and provides an interface into outside applications. The agent must understand outside data formats and be capable of converting them into a format other agents can understand [KA97]. It also is responsible for displaying results to the user. An *Ontology Agent* maintains and provides overall knowledge of ontologies and answers queries about the ontologies. It may simply store the ontology as given, or it may be as advanced as to be able to use semantic reasoning to determining the applicability of a domain to any particular data mining request [NWA96]. Another common agent is a *Broker Agent*. A

Broker Agent maintains all information on the capabilities of individual agents. It also responds to queries from agents as to where to route specific requests. In general, any new agents in a system using a Broker Agent must advertise their capabilities through the broker in order to become a part of the agent system. A *Resource Agent* provides the map from the common ontology to a specific database schema and is knowledgeable about the language required to interface with the resource. This agent is critical to any information retrieval system [SHOH97]. A *Data Analysis Agent* is a Resource Agent specialized for data analysis/mining methods. It is mentioned separately from the Resource Agent as it is becoming more commonly used in agent systems. A *Task Execution Agent* coordinates the execution of high-level information-gathering subtasks required to fulfill scenarios. It remains in close contact with the Broker Agent to determine what agents in the system are capable of fulfilling any given task, and then tasking those agents it deems useful [SHOH97].

The definitions and individual agent functions begin to show how the agents may be useful and interact with each other. Not every system may require all agents or may use agents in a different capacity. In order to see how they can be utilized to fulfill a system goal, several of the key agent-based information retrieval systems are presented.

One common property of most agents is the ability to communicate with other agents. While some agents can perform their goals without this, most multi-agent systems rely on the ability of agents to communicate to fulfill their goals.

2.4.4 *Agent Communication (Speech-Act)* Multi-agent systems rely on the ability of individual agents to communicate with each other to fulfill system goals. By nature, multi-agent systems are generally distributed, making interaction more difficult. Interaction in multi-agent system has two key components. First is the language or interaction protocol, and second is the use of *performatives.*

In a conversation-centric system, the actions of an agent are driven by the communications it has with other agents in the system [CHAU97]. When dealing with distributed agents, it is important to develop a common language that any agent can understand. The use of a common language ensures that any new agent can receive a message, and based on the language protocol in use, extract the information it requires. This is accomplished through a common message format. A common format allows agents to interface with other agents regardless of the agents internal structure [CHAU97].

Once a common communications protocol has been established, *performatives* must be developed that can give receiving agents direction or direct agent actions towards a system goal. As such, performatives are the speech-act component of the language [CHAU97]. Performatives are specific to each system and are dependent on the functions and goals of that system. For instance, a system may use a "command" performative to indicate a request.

All projects reviewed work from the most common speech-act agent language used, the *Knowledge Query and Manipulation Language* (KQML). KQML handles the interface protocols for transmitting queries, returning the appropriate information, and building the appropriate internal structures [BAY96]. Every KQML message consists of an *operation type* and any information containing parameters required for the operation. The operation-type simply indicates the type of communication (tell, ask-if, ask-one) and can either be a fixed KQML operator, or a system specific performative. KQML is indifferent to the format of the information itself and relies on the system to specify or understand format. As such, it can be used as a shell

to contain messages in various languages and also allow agents to route messages, even if they do not understand the syntax or semantics of the content message [BAY96].

2.5 Agent-Based Information Gathering Frameworks

The following projects are representative of a multitude of agent-related, heterogeneous information projects in the literature. They contain unique features that set them apart from the other research that may be applicable to this research. Of most interest is the agent architectures used and the similarities seen between the systems.

2.5.1 *CARNOT*

Initiated in 1990 at the Microelectronics and Computer Technology Corporation (MCC), it was one of the first large-scale attempts at unifying distributed, heterogeneous information [BAY96]. Carnot executes queries in a distributed environment by dispatching autonomous computing agents to remote sites where they access databases and cooperate among themselves to properly merge resulting data into understandable information.

Carnot provided two key technological advances. First, they developed knowledge representation techniques for capturing and maintaining an enterprise model as well as the operations that map that model to the physical databases. The second advance, and of most interest here, is the use of intelligent, autonomous agents to retrieve the enterprise information and control enterprise processes [KAH94].

Intelligent agents are used to take a query, with reference to the common model from a client application, and retrieve the information [BAY96]. They first consult the repository to find which databases needed to be accessed, then create other agents to execute the required accesses. Each individual agent also contains the mappings needed to translate information from an individual database into the correct format [WT95].

2.5.2 InfoSleuth Carnot was not designed to operate in a dynamic environment where information sources change over time and where new information sources can be added autonomously and without formal control [BAY96]. InfoSleuth extends the Carnot technology into this dynamically changing environment. Because sources can be added without formal control, information-gathering tasks are defined generically, and results are sensitive to available resources [BAY96]. Using the agent-based architecture developed by Carnot, the InfoSleuth Project developed and demonstrated technology that expedites the search for pertinent information in a geographically distributed and constantly growing network of information resources. The InfoSleuth architecture consists of a set of collaborating agents that work together at the request of the user to:

1. Gather information via complex queries from a changing set of databases and semi-structured text repositories distributed across the Internet.

2. Perform rudimentary polling and notification facilities for monitoring changes in data.

3. Automatically route location-independent requests to update individual data items.

4. Analyze information using statistical data mining techniques and/or logical inferencing.

In the InfoSleuth environment, information is advertised by describing its information content in terms of a network-wide distributed taxonomy. This taxonomy is similar to a dictionary or directory but contains more information concerning the meaning of an entry and its relationship to other entries [WT95]. Figure 2 shows the basic structure of the InfoSleuth system. Together, an *Ontology Agent* and *Broker Agent* provide the basic support for enabling the agents to interconnect and intercommunicate.

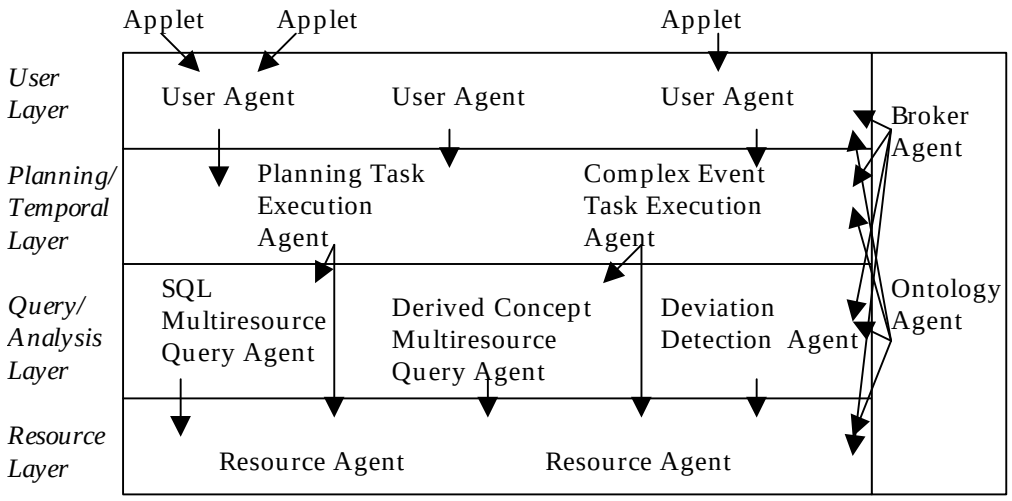

Figure 2 InfoSleuth Architecture Layers

The *Broker Agent* maintains a knowledge base of information that all the other agents advertise about themselves, and uses this knowledge to match agents with requested services. Thus, technically, the broker does semantic matchmaking. When an agent comes on-line, it advertises itself to the broker and makes itself available for use. When an agent goes off-line, the broker removes the agent from its knowledge base.

Several different types of agents are utilized for processing information within InfoSleuth. They provide more specific definitions of the agents mentioned before. First, *User Agents* act on behalf of users to first formulate their requests and pass them on for execution, and then match the responses with the requests and pass them back to the requesting applet. *Resource Agents* provide the interface to the various databases and other repositories of information as required. If a query does not require a particular database, that database is not used.

Task Execution Agents plan how the request should be processed within InfoSleuth, including result caching. Result caching involves storing the results of a query in case the same query is processed again. Task Execution agents may also be specialized to monitor for complex events that include changes in the data sources over time and simple events detected within individual resources.

Within the InfoSleuth system, the agents themselves are roughly organized into layers as shown in Figure 2, with the Broker and Ontology Agents serving all of the other agents. Users access resources via a middle set of layers that acquire and process the information from the resources as requested. Within the two middle layers, the upper, *planning/temporal* layer, deals with processes that occur over time, such as the planning of tasks and the detection of complex events that may be composed of sequences of simpler events. The lower, *query/analysis* layer, executes one-time subtasks such as the retrieval of a current snapshot of some related information or the detection of an anomaly in the data stream as it occurs.

InfoSleuth introduced several key technologies different from Carnot. First, it had the ability to execute complex queries from a *changing* set of data sources. By monitoring sources, it can provide improved query processing, utilizing sources that will provide more reliable and useful results. It also extended the Carnot architecture with these mobile agents to provide information analysis. Carnot did not offer statistical data mining techniques within its framework.

2.5.3 SIMS The Services and Information Management for decision Systems (SIMS) exploits a semantic model of a problem domain to integrate information from various information services [ACHK93]. In SIMS, the goal of information agents is "to provide information and expertise on a specific topic by drawing on relevant information from other information agents" [KAH94]. Every SIMS agent contains a detailed model of its domain of expertise and models of the information sources available to it. Given an information request, the agent selects the appropriate set of information sources, generates a plan for retrieval, uses its knowledge of the sources to reformulate the plan for efficiency, and then executes it.

Sources are modeled in each agent by the description of the classes contained within that source. The relationships between the classes and the classes in the domain model are maintained as well. Each agent contains a model of its own domain, as well as models of the other agents that can provide relevant information [KA97]. The domain model is an ontology representing the domain of interest of the agent. The agent also has an information-source model that describes both the contents of information sources and their relationship to the domain model. In this way, an agent only maintains the portion of the ontology and information sources relevant to it.

Every information agent is specialized to one application domain and provides access to all available information sources within that domain. The domain model provides the description of the information available from that agent to other agents or human users.

SIMS differed from the other projects by using an advanced semantic model of the problem domain. By performing more processing initially, it avoided expensive I/O access that would not be useful. It went beyond simple semantic modeling by modeling relationships between the classes of a source and classes in the existing domain model. This was one of the first projects to provide advanced ontological services providing relationships.

2.5.4 *TSIMMIS* The goal of the TSIMMIS project is to provide tools for accessing, in an integrated fashion, multiple information sources [MHIP95]. The TSIMMIS architecture is shown in Figure 3. Above each source is a translator (or wrapper) that logically converts the underlying data objects to a common information model. To do this logical translation, the translator converts queries over information in the common model into requests that the source can execute, then converts the data returned by the source into the common model.

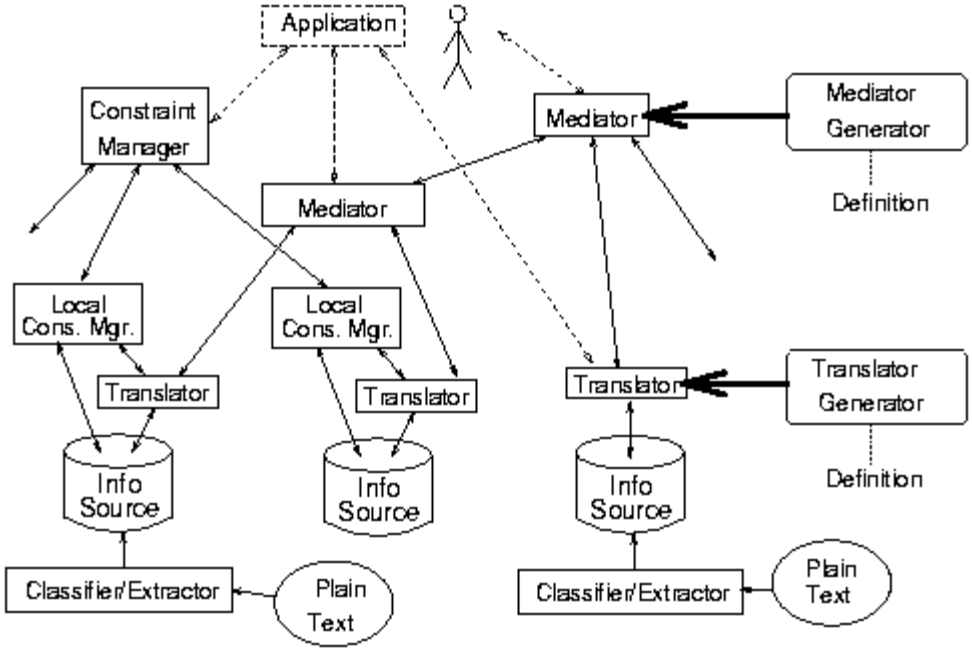

Figure 3 TSIMMIS Architecture

Above the translators in the architecture are the *mediators*. A mediator is a software module that refines in some way information from one or more sources [MHIP95]. A mediator embodies the knowledge that is necessary for processing a specific type of information. For example, a mediator for "current events" might know that relevant information sources are the AP Newswire and the New York Times database. When the mediator receives a query, such as for "articles on Bosnia," it will know to forward the query to those sources. The mediator may also

process answers before forwarding them to the user, for example, converting dates to a common format or eliminating articles that duplicate information.

There are a number of differences between integration of information sources in the TSIMMIS project and other database integration efforts [MHIP95]. First, TSIMMIS focuses on providing integrated access to very diverse and dynamic information. The information may be unstructured or semi-structured, often having no regular schema to describe it. The components of objects may vary in unpredictable ways (e.g., some pictures may be color, others black and white, others missing, some with captions and some without). Furthermore, the available sources, their contents, and the meaning of their contents may change frequently.

Second, while not particularly beneficial from an automation standpoint, integration does require more human participation. In the extreme case, integration is performed manually by the end user. For example, a stockbroker may read a report saying that IBM has named a new CEO, then retrieve recent IBM stock prices from a database to deduce that stock prices will rise. In other cases, integration may be automated by a mediator, but only after a human studies sample of the data, determines the procedure to follow, and develops an appropriate specification for the mediator generator.

Finally, TSIMMIS assumes that information access and integration are intertwined. In a traditional integration scenario, there are two phases: an integration phase where data models and schemas (or parts thereof) are merged and an access phase where data is fetched. In the TSIMMIS environment, it may not be clear how information is merged until samples are viewed, and the integration strategy may change if certain unexpected data is encountered.

In summary, the goal of TSIMMIS is not to perform fully automated information integration that hides all diversity from the user, but rather to provide a framework and tools to assist humans (end users and/or humans programming integration software) in their information processing and integration activities.

2.6 Common Information Retrieval System Architecture

While each project presented was unique in some aspect of its implementation, they share several commonalties. These commonalties have become a template for most information retrieval systems. They setout three important concepts for systems – a*gent technology, domain models,* and *information brokerage*. *Agent technology* introduced collaborative agents which comprise a network, communicating by means of a high level query language KQML (Knowledge Query and Manipulation Language). *Domain models* or ontologies, give a concise, uniform description of semantic information, independent of the underlying syntactic representation of the data. Finally, information brokerage utilized specialized Broker Agents to match information needs with currently available resources, so retrieval and update requests can be properly routed to the relevant resources.

Developing a system using specialized agents with the ability to communicate with a single information source, as well as with other agents, allows for a great deal of flexibility [KAH94]. For instance, adding a new information source merely implies adding a new agent and advertising its capabilities. In doing this, the systems reviewed all utilized a general approach that is outlined below.

The general system operates as follows - When a query is made, the first step is to select the appropriate information sources. There are several areas of thought here. Singh proposes using metadata compiled at the time of the query to determine what sources to use [SING98]. In this case, dynamic changes to knowledge sources are captured and reflected with each query. The InfoSleuth project initializes the Ontology Agent at start-up, and all domain related queries are routed to it, so dynamic changes in data are not necessarily captured unless the system is restarted [BAY96].

The next step is to produce a plan to implement the required retrieval. Planning schemes vary from system to system, but generally involve coordination of retrievals require ordering and

assignment to the appropriate agents. Overcoming the problem of redundant data in different sources is handled by minimizing the number of different information sources used to answer the query [KAH94].

The steps in the plan are partially ordered based on the structure of the query. This ordering is determined by the fact that some steps make use of data that is obtained by other steps, and thus must logically be considered after them. Next, the plan produced is inspected and, if possible, data retrieval steps that are grounded in the same information source are grouped. Finally, the system reformulates a query plan into a less expensive, yet semantically equivalent plan.

Metadata descriptions can be used to infer relationships between objects, unify heterogeneous data representations into a common object data model and rapidly evolve applications. By using metadata specifications for information sources, user query models, and business logic rules, a system can decide dynamically how to handle requests at run time.

By making use of metadata at run-time, any changes in the information are reflected immediately in a user query. This is in contrast to a procedural approach, in which a sequence of steps must be prescribed to answer each query (the plan). A change in a source may require changing all procedures that can very time-consuming. Once an extensible system is designed, the individual agents must be built. Building agents is generally done with the help of various tools and agent development frameworks. Two Java-based frameworks are discussed next.

2.7 Multi-Agent Development Frameworks

In general, an agent development framework provides a set of templates and code that facilitates or implements basic communication. It may also provide templates for various types of agents or constructs that agents can use. Basic communication can be as simplistic as e-mail or

as advanced as direct communication. The key differences between most development framework lies in the implementation and architecture of the provided communication and agent functionality. Both JATLite and JAFMAS are described here and the methods of implementation are discussed. Both are Java based frameworks that allow directed communication between agents.

2.7.1 *JATLite* JATLite provides a set of Java templates and a Java agent infrastructure that allows agents to be built from a common template. The template for building agents utilizes a common high-level language and protocol [JAT97]. This template provides the user with numerous predefined Java classes that facilitate agent construction. The classes are also provided in layers so that the developer can easily decide what classes are needed for a given system. In this way, if the developer decides not to use KQML for example, the classes in the KQML layer can be omitted. However, if that layer is included, parsing and other KQML-specific functions are then automatically included in any agent developed from the JATLite base classes.

The key difference between JATLite and the other systems is the agent communication infrastructure packaged with it [JAT97]. Traditional agent systems use some type of Agent Name Server (ANS) for making the required connections between agents. An agent uses an ANS to look up the IP address of another agent and then make a TCP socket connection directly to that agent for the purpose of exchanging messages.

With such an ANS, if the IP address of the other agent changes, the first agent finds out when the next attempt to send a message fails. If the second agent "crashes" in any way, it is the responsibility of every other agent with whom it was communicating to properly save the failed messages and resend them later.

JAT*Lite* uses the Agent Message Router (AMR) to act as the “server” and receive messages from the registered agents and routes the message to the correct receiver [JAT97].

Received messages are also queued to the file system to ensure a resend can be accomplished if a failure should occur. This provides more assurance a message will be successfully transmitted but also places the burden of communication on a central machine. If a crash or other error occurs in the AMR, no communication can occur and all queued messages are lost.

2.7.2 *JAFMAS* The Java-based Agent Framework for Multi-Agent Systems (JAFMAS) is a Java-based development framework that also provides a set of Java templates and a Java agent infrastructure to allow agents to be built from a common template [CHAU97]. The core classes provided by JAFMAS provide for both directed and multicast communications. Borrowing heavily from COOL, a language for representing, applying, and capturing cooperation knowledge in multiagent systems, JAFMAS defines the social behavior of agents. Like COOL, JAFMAS defines all interactions between agents as "conversations" and information exchange is performed through the conversation in the way of performatives or through messages between agents involved in a conversation.

The key difference between JAFMAS and other systems is the use of multicast messaging to establish an agent's identity [CHAU97]. Multicast is a Java provided datagram socket class that allows joining "groups" of other multicast hosts on a network. It differs from broadcasting in that messages are sent to all members of the "group", not the entire network. This ensures bandwidth is conserved and only agents which are affected by a message actually receive it. More importantly, it frees up a multi-agent system from relying on a central registry for agent identity and message routing. This ensures a system can function even if an agent should fail.

2.8 Summary

This Chapter presented an overview of the different technologies that provide a foundation for this thesis. It first covered both generally accepted definitions of what defines an agent, as well as some attributes that are more heavily debated. As agents are developed with combinations of these traits and attributes, general classes have begun to form. These were described and potential uses were covered. Information Retrieval Agents were then covered in more detail, including some specific types. The importance and methods of accomplishing agent communication in a multi-agent system was then discussed. Several projects that focused on new or unique implementations of information gathering frameworks utilizing agents and were presented. Each was reviewed because of some unique aspect in which it used the agents or retrieved the information. Finally, two agent development frameworks were covered, and potential trade-offs of each were covered. One area that was found noticeably lacking in literature was data mining of heterogeneous sources for association rules, then unification of results.

3 Methodology

3.1 Overview

Much like any software development process, developing a multi-agent system (MAS) should follow a logical design process tailored to the goals of the target system. The frameworks discussed in Section 2.7 provide the Java code necessary for representing and developing the coordination knowledge and protocols required for a multi-agent system, but do not provide guidance in the design of the system and determination of what agents may be required. In order to apply any framework in the design and development of a system, a general methodology must be applied. This chapter describes a five-step methodology similar to Object Oriented Analysis (OOA) working from problem analysis to detailed agent design.

3.2 Methodology

The purpose of this research is not to develop a new agent development methodology, however none was found that was adequate for the system being developed. Because of this, the following methodology was developed. This section presents this methodology for development that, similar in form to top-down Object-Oriented development methodology. This methodology assumes that the decision to use an agent-based framework has already been made. It will not aid in the decision of whether or not to use agents. By developing a methodology similar to that already used, existing tools and ideas can be leveraged It is important to remember that we are not simply defining agents that can communicate, but an entire system that has defined goals based on a problem description. In each step, the goals, scope, and level of granularity is discussed. The following scenario is used as an example:

A program takes a request to find which distributed database a particular table is currently stored in. The program is located on the user's computer. The available databases are geographically separated and have different formats.

The overall flow of the methodology is shown below in Figure 4

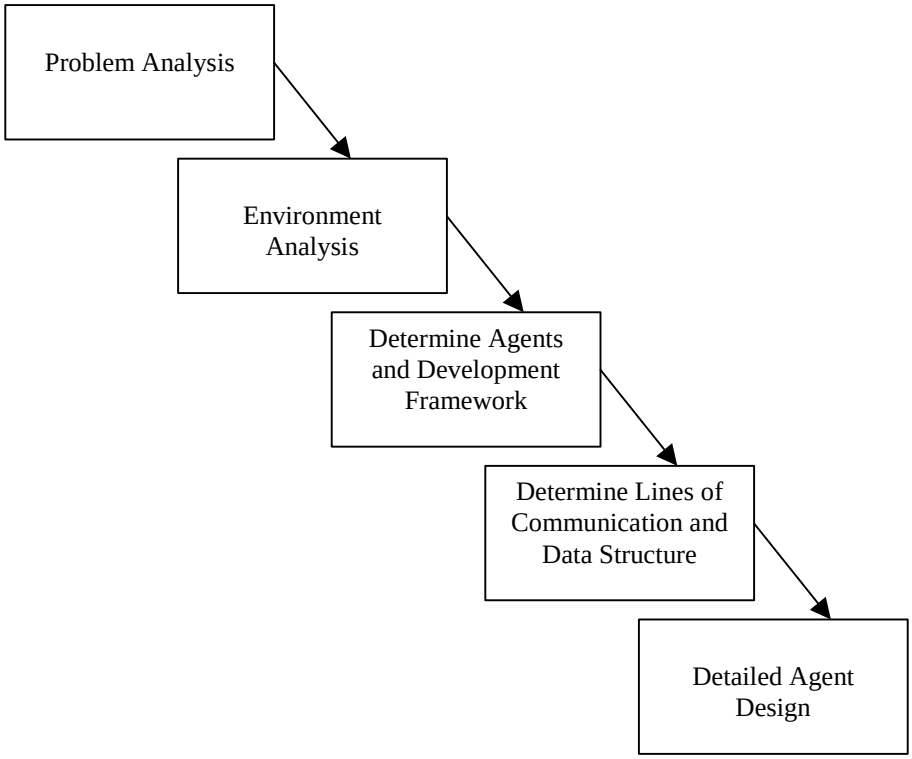

Figure 4 Methodology Flow Diagram

3.3 Problem Analysis

The first step is to define the system based on the original problem description. This entails specification of the inputs that the system can expect to receive, as well as expected outputs from the system. At this level, the objects should be specified at a granularity no more specific than domain level concepts. In agent development, such domain level concepts can include, but are

certainly not limited to, interface and processing. Inputs into and out of each object should be specified at a high level of granularity as well. The goal is to show system flow and clearly define expected system input and output streams. By breaking the system into domain level concepts, we also begin to scope what agents may be used. An example of this is shown below in Figure 5

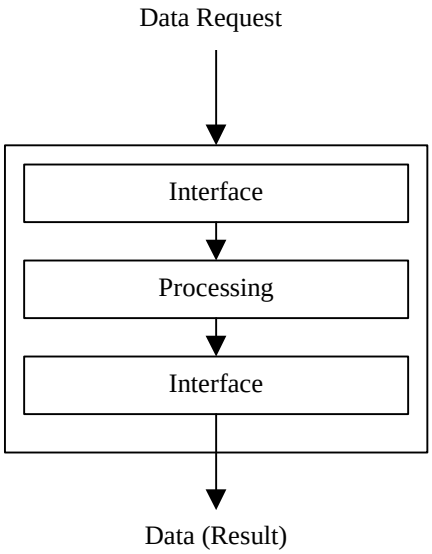

Figure 5 Sample Analysis

At this point, an agent development framework **should not** be considered. Further decomposition must be done in order to properly evaluate which framework would be most beneficial.

3.4 Environmental Analysis

Once analysis is complete, the environmental analysis is accomplished. The environment includes not only the agents themselves, but must take into consideration hardware issues as well. Some potential hardware issues may be memory requirements or the use of distributed computers

vs. a single machine. Definition of the potential sources of information for input must also be accomplished. Similarly, all outputs should be directed to another application or component of the system. The problem definition and the existing sub-systems (computers) largely drive the system design. For instance, if the task is to search distributed databases for instances of a particular piece of data, the location of those databases may be static and dictate a heterogeneous operating system, multi-computer, LAN-linked system. A sample system design is shown below in Figure 6.

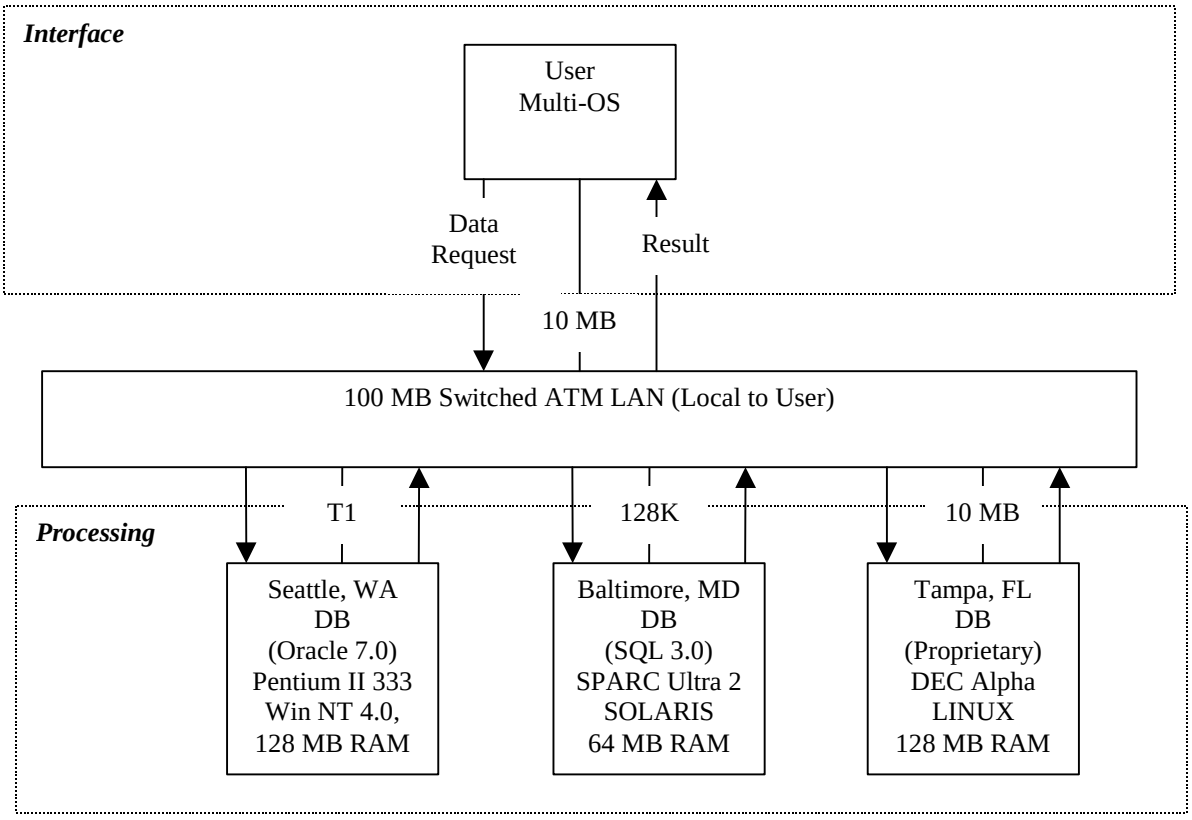

Figure 6 Sample Problem Analysis

It is important to be as specific as possible given the information known. While the above sample does not provide detailed information, it does reflect a WAN-to-LAN linked,

heterogeneous operating system, multi-database system. It also shows where each system component falls within the domain level concepts decided on in the analysis phase. While the above figure only reflects the graphic representation of the system, careful documentation of each component and the high-level requirements of that component should be accomplished as well.

3.5 Determine Agents and Development Framework

Once the system level requirements and high-level objects have been determined, identification of agents can occur and a framework can be selected. Agents should be based on either generally accepted definitions of agents or specialized agent definitions that are clearly spelled out in the system documentation. Some generally accepted agent definitions have been provided in Section 2.4. Once agent definitions have been decided upon, the agents should be laid out with respect to the high level objects found in the analysis phase. The system level inputs and outputs should be shown as they flow from agent to agent. The goals of each agent and the services they will provide in the system should also be clarified.

Once the goals and services are clarified, a framework can be selected. The framework should be based on the goals and services desired. If a framework has already been mandated, it should be evaluated for potential problems related to the frameworks' strengths and weaknesses. For example, if a TCL based framework was originally mandated, but the agents identified will only ever reside on one machine and communicate through the network, another framework may be more useful and more efficient. Continuing the example from the previous phases, Figures 7 and 8 show how agents are determined and fit into the previous results.

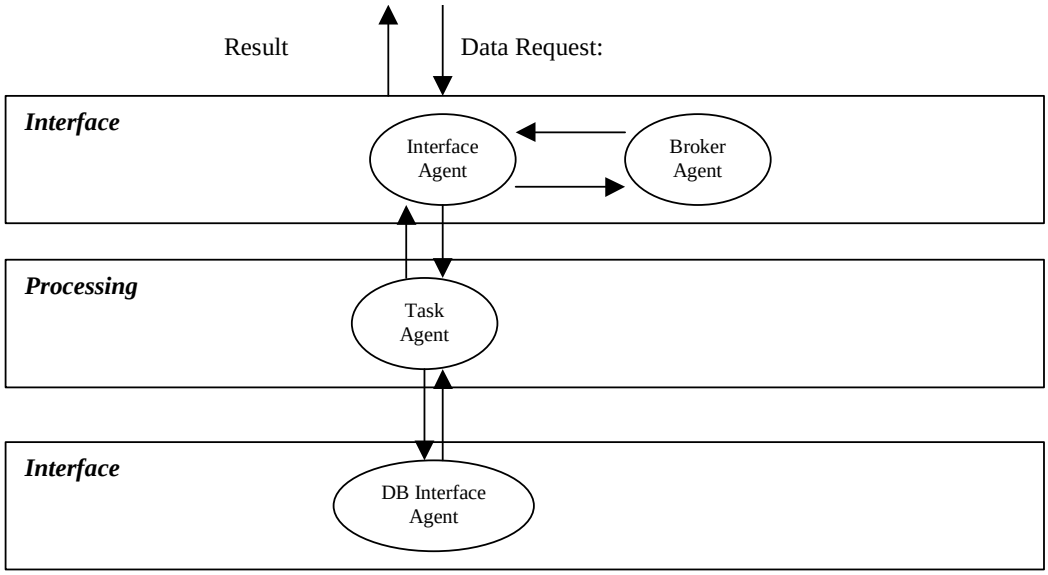

Figure 7 Determining Agents and Framework

Figure 7 shows what agent will reside on each component identified in the enviromental analysis phase. In the case of Mobile Agents, the path the agent can travel must be reflected. In the example shown, the program takes a request to find which database a table is from then it checks the broker to find all available databases. The Task Agent passes the task to the databases and monitors their progress. Each database must have an interface agent to check the database for a matching record and then report back to the task agent.

Figure 8 depicts the known communications infrastructure that must be utilized. Again, it depicts the fact that the system must work across a distributed environment and may have some bandwidth constraints that should be considered. The location of each agent is also shown.

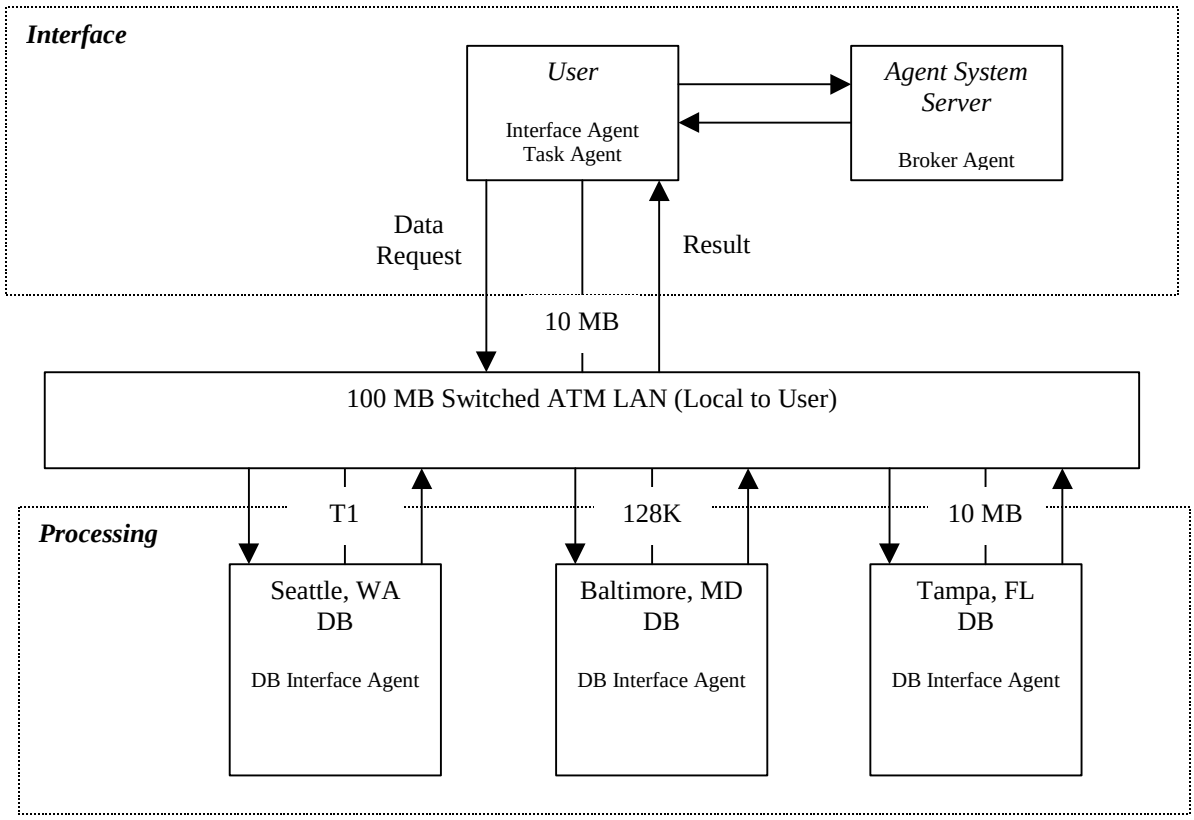

Figure 8 Sample Problem Framework Analysis

3.6 Identify Lines of Communication and Data Structures

For each line of communication to and from an agent, data structures should be identified for the information that is being passed. In an agent framework such as JAFMAS, these lines of communication will ultimately become conversations. Based on the agent framework selected, the information that should be specified for each line of communication may be different. For frameworks using performatives, as most do, the performatives should be clarified, along with the data structures each message will contain. In a communication-centric language such as JAFMAS, finite automata should be developed for each conversation, as well as for system level

states. This ensures proper information flow and reduces the risk of infinite wait states. In the example being used, the interface to broker conversation is modeled below in Figure 9.

Performatives:

- Avail Agent Request
- No Agents Avail
- Agents Exist

Data Structures

- AgentList : Array of Agents
- Agent : Record - Name, Location, tasks performed
- Request : Table to find

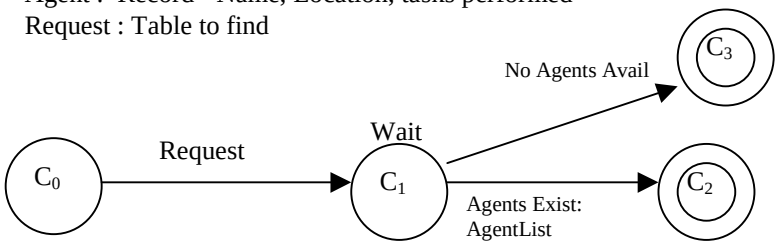

Figure 9 Sample Conversation Diagram

Each conversation should be identified, then modeled. For each model all states should be identified and all information passed should be reflected.

3.7 Detailed Agent Design

Once each agent knows exactly what information it will receive or generate, specific algorithms can be developed to perform the proper agent tasks. The agent framework selected should already provide core agent functions such as communication and perhaps even planning. The algorithms developed should be specific to the individual agent task, but utilize the data structures that were identified in the specification of the lines of communication. Since the information and data structures that will be passed to the agent have already been identified, as

well as the expected output, the algorithms selected should simply perform the processing required to go from input to expected output.

3.8 Summary

Development of a communication-centric, multi agent system should be not accomplished in an ad hoc manner. This chapter outlined a five-step methodology that can be applied to develop a system for a specific problem. It does not aid in the decision on whether a system is best-solved using agents. The methodology starts looking at the system from a domain level view and moves to detailed agent design. It provides for selection of an agent development framework, but only after specification of as many components as possible. By applying these five steps to the problem presented in this research, a multi-agent system is evaluated and formed. This process and specific application is described in the next chapter.

4 Proposed Agent Architecture

4.1 Overview

The previous chapter explained the methodology that can be applied to the problem described in Chapter 1. Following this methodology allows for a logical thought process to be applied to the process of individual agent design and system integration. This chapter applies this methodology and describes the process at each phase. This includes and potential design decisions and trade-offs associated with the decisions.

4.2 Problem Analysis

Chapter 1 describes the problem to be solved in detail, however a short excerpt is included below for review:

> This system will use an existing agent development tool to establish a multi-agent based framework and define the communications between those agents. The framework will accept a request for one of three possible data mining operations. Once the system accepts the request, it will determine which data sources can fulfill the request and tasks the agents responsible for those sources to begin data mining. Once results have been obtained, they will be unified to eliminate redundant or conflicting results.

Analysis of the problem shows that there are two main domain level concepts that must be utilized – interface and processing. The new system must interface with an outside application to receive the data mining tasking. Once it receives the tasking, it must *process* the data and determine the proper data sources. Data sources are then mined (still under processing) and results are unified, then presented back to the application (interface). This is represented pictorially below in Figure 10:

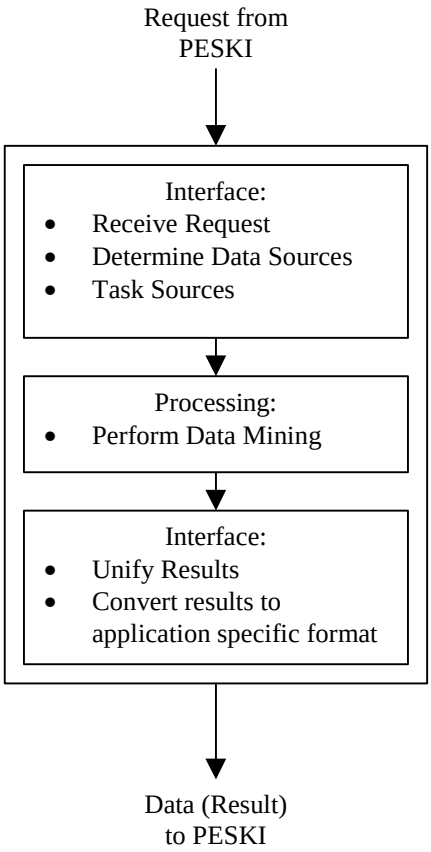

Figure 10 Domain-Level Problem Analysis Diagram

4.3 Environment Analysis

The problem description in Chapter 1 designates some of the environment and system properties. First, the system can have several data sources on various machines. Data sources can be of the same or heterogeneous formats. There is no mention of operating system (OS) requirements so it is assumed that they could operate on any major operating system. There is also no mention of geographical location so it is assumed that each data source could be located on a separate machine in any geographic location, but will have access to some method of network communication. The environment is presented pictorially in Figure 11. The arrows

represent (as yet unspecified) information being passed. No information is given about expected computer specifications or transmission rates so none is specified in the figure. It is possible a data source could reside on the same LAN as the DBMiner program, in which case the intermediate LAN or WAN would not be required.

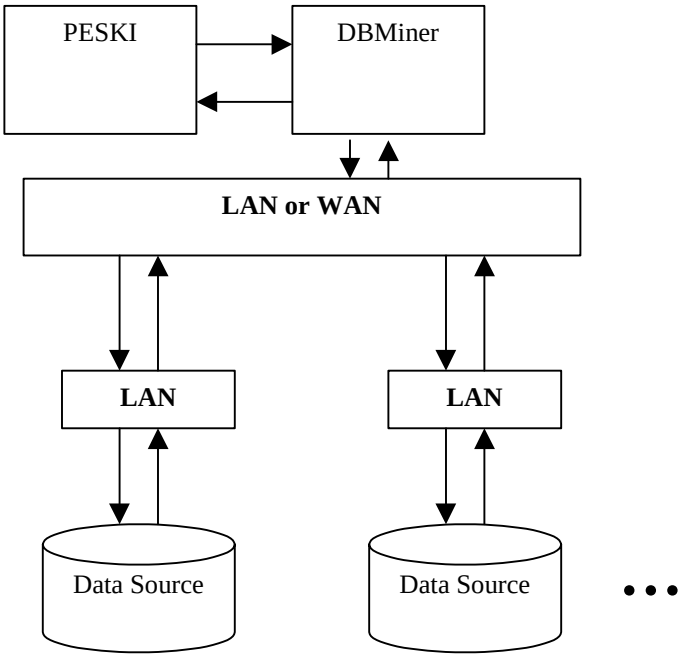

Figure 11 Environmental Problem Analysis Diagram

4.4 Determine Agents and Framework

Based on the problem and environment analysis, there are several types of agents in this system. Each adheres to the general agent definitions described in Section 2.4. They are grouped based on the similarities of the tasks they perform and their individual goals. The seven main categories of agents in this system are User, Task, Broker, Ontology, Data Analysis, Unification and Registration. Each falls within the domain level concepts specified in the problem and environment analysis phase. The User, Task, Broker, Ontology, and Registration Agents are all

interface agents. They all provide interfaces to either an outside system, agents within the system, or data sources. They do not process the data in any way. The Data Analysis and Unification Agents are processing agents which manipulate the request or data within the system. All agents are discussed in more detail in Section 4.4 and Section 4.6. The overall system, with lines of communication, is depicted in Figure 12. The specifics of the conversations are covered in Section 4.5.

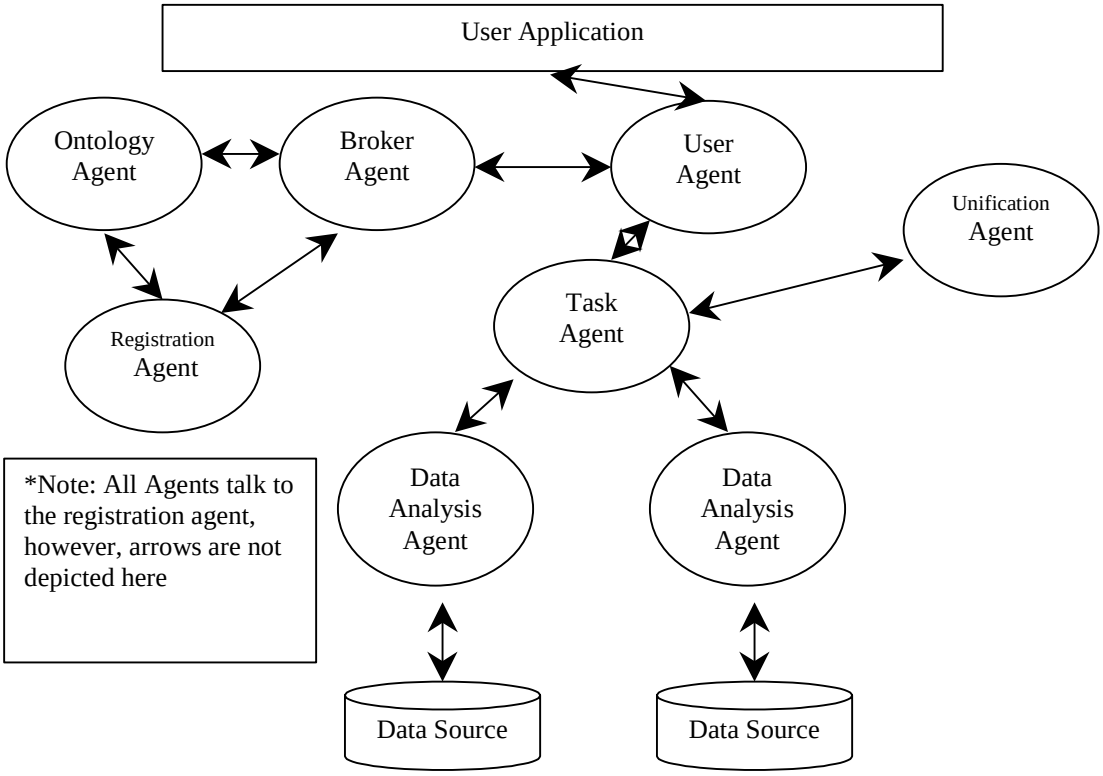

Figure 12 Overall Agent System Diagram

The next subsections will describe the function of each agent from a high level perspective. It also discusses design decisions that were made and the reasons that they were made. The agents are presented in the same order as they might be utilized to process a request in order to see how each agents function fits into the system and supports the other agents.

4.4.1 *Registration Agent* When any new agent is introduced into the system, it must first inform the Registration Agent that it has entered. The function of the Registration Agent is to inform all appropriate agents that are already in the system of a new agents arrival. Since the system is designed to be relatively static in terms of new data sources and data source types, the Registration Agent will be the least utilized agent. It should also be the first agent created and started in the system. Because the system used here cannot operate without a Broker Agent as well, the Registration Agent will initiate all methods, then await a Broker Agent to enter the system.

Once notified a broker has entered the system, it completes initialization and waits for a registration request from any new agents. When a new agent enters the system, it sends a registration request message to the Registration Agent. When it receives notification of a new agent, the Registration Agent will determine the functions the agent can perform by the information transmitted in the registration message. Based on the class or functions, it will then determine who should be informed. The Broker Agent will be informed of all classes of agents entering the system. In the case of a Data Analysis Agent (the most common type of new agent), the Ontology Agent must also be informed of the entry. Once all appropriate existing system agents have been notified, the new agent will be informed it is active in the system.

4.4.1.1 *Registration Agent Design Decisions* The Registration Agent could be eliminated and its functionality be shifted to the registering agent. Two approaches could be taken. First, any new agent could simply be required to determine what agents need to be contacted then contact them directly. By forcing new agents to perform extra processing and include additional system specific information, the system loses some extensibility. The second option is to have each new agent simply broadcast the fact it has arrived, and any existing agents that need information from it can then request it directly. This is also undesirable because of the

additional overhead a broadcast message consumes. It requires agents who may not be affected to commit processing to the message as well as consumes bandwidth. Finally, the Registration Agent provides for future expansion of the system to include various metrics or monitoring agents. By utilizing a Registration Agent as a central point of information exchange in the Registration Agent, any future tasks such as dynamic data mining algorithm assignment can be easily included.

4.4.2 *User Agent* The system has one point of entry into the end application. At this point of entry all unified results must be presented and all requests for data mining retrieved from the application in a format it understands. This dictates three essential operations the user agent must be able to undertake. First, it must be able to pick up and understand requests. Second, it must be able to present results in a format the application can understand, and finally, it must be able to pick up such asynchronous events such as a *stop mining* or *end operations* from the application. The user agent has all the knowledge required to translate information from the application to a format the agents can understand and vice-versa.

The first task, requests for data mining, will be one of three possible requests. First, the user can select to discover all trends with a given statistical significance. Statistical significance with respect to mining association rules consists of a specification of a value for confidence as well as support as presented in Section 2.2 The value of the statistical significance must be set either by the user or set in the system. Second, the user can specify an item (X) and ask for either sets or individual items (Y) that are involved in transactions enough to be “of interest”. Again, “of interest” refers to items above some statistical significance level set by the user or system. Finally, they can specify an X and Y and ask simply if there is a statistically significant trend between the two. All of these options will initiate the search for association rules of the form X $\Rightarrow$ Y across the data sources available.

4.4.3 *Task Agent* Once the user agent has retrieved the data, it passes it to the Task Agent. The Task Agent must determine, based on the information passed to it by the User Agent, what agents to task to fulfill the request. The information received can dictate one of two possible requests. The first is for a cancellation of the current operation. Such a request from PESKI may occur if a user feels the current operations are taking too long, or are no longer needed. In this case, the Task Agent must send the cancel message to all agents currently tasked and performing work.

The other possible request from the application is for one of the three data mining operations. No matter which of these three tasks it must undertake, it will ask the *Broker Agent* for all agents which can fulfill the desired tasking (This is covered in more depth in the next section). If the user wants all possible rules meeting minimum support and confidence levels, across all available data sources, then the Task Agent must task every data agent possible for all association rules. If the user wants to find all items Y which have statistical significance for a given X, the Task Agent must task only those agents which have information about X. It would be time consuming and wasteful to task agents which have access to data not containing X, as no rules would be generated. Finally, if the user specifies an X and a Y and asks for the level of support and confidence between the two, the agent must again only task those agents that have information about both X and Y.

Once data mining is completed, the Task Agent accepts all the results from the individual Data Analysis Agents. When all Data Analysis Agents are finished, the Task Agent passes the results to the Unification Agent. When the Unification Agent is completed it returns the results and the Task Agent passes them onto the User Agent.

4.4.4 *Broker Agent* To fulfill a tasking the Task Agent must talk to the Broker Agent, asking which agents can fill the request. The Broker Agent maintains all information on

the capabilities of individual agents in the system and responds to queries from agents as to where to route specific requests. By requesting only those agents who may have relevant information, the Task Agent can eliminate tasking any agents that could not possibly discover any useful rules. However, the Broker Agent does not maintain ontological information about the agents in the system, only their high level functionality and where they are located. In order to determine which agents could have the information the Task Agent will need, an Ontology Agent is used. The Ontology Agent (discussed later) maintains and provides overall knowledge of ontologies and answers queries about the ontologies.

The Broker is also notified whenever a new agent enters the system. Each time the Registration Agent notifies the Broker Agent of a new agent, it must add the agent and its capabilities to the list of available system agents. The Broker Agent interacts with the Task Agent, the Ontology Agent, and the Registration Agent.

4.4.4.1 Broker Agent Design Decisions The alternative to using a broker and Ontology Agent is to use the multicast capabilities offered by JAFMAS. When a Task Agent receives a request from the User Agent, it could simply send a multicast message to all Data Analysis Agents requesting data mining for a particular X or Y or both. The individual agents can then check their domains to see if they have the X or Y and respond appropriately. This was not done for several reasons. First, while JAFMAS offers this capability, other frameworks do not, and to offer a truly extensible architecture, this system should not be too closely tied to the features of JAFMAS that could not be implemented under another framework. The multicast ability is closely tied to the JAVA RMI registry, and is not a universally implemented feature. Other features, such as the direct message capability, are implemented in other ways in other frameworks, so this system could be more easily ported.

4.4.5 *Ontology Agent* As the Broker Agent determines an agent may be useful for any given task, it queries the Ontology Agent to determine if the agent has the information required. This would be useful if the user has specified an X and wants all Y, or has specified both an X and Y. The Ontology Agent maintains all the random variables for the data source a Data Analysis Agent is responsible. By comparing an X or Y value against this list of random variables, the Ontology Agent can determine if the value will be found in that Data Analysis Agents source. When a new that Data Analysis Agent is added to the system, the Ontology Agent is notified by the Registration Agent. Once it receives notification, it adds the agent and its domain to the list of all agents that have domains, and the respective domain.

The Ontology Agent only maintains a list of the random variables in that Data Analysis Agents data source. Currently it does not maintain any semantic information about those random variables

4.4.5.1 *Ontology Agent Design Decisions* The Ontology Agent function could be easily integrated into the Broker Agent as they currently use the same data structure and maintain the same information. This is only because of the simplistic nature of the Ontology Agent for this system. The Ontology Agent can perform much more advanced domain checking through semantic interpretation of random variables. In the future, if expansion or revisions occur, the Ontology Agent will emerge as a necessary separation from the Broker Agent. It is for these reasons that it is separated now, rather than later.

4.4.6 Data Analysis Agent Once the Broker Agent has determined what agents can fulfill a given task, it passes the information back to the Task Agent. The Task Agent then tasks each useful Data Analysis Agent, passing it the relevant information. “Useful” in this case refers to a Data Analysis Agent that is responsible for a data source that includes either the X or

Y value of the request in its domain. The Data Analysis Agent encapsulates two key classes. First, it includes a resource interface for data source specific retrieval and also has an instance of a data mining algorithm to operate over the resource interface associated with the Data Analysis Agent.

The Data Analysis accepts a request from the Task Agent and initiates the data mining algorithm using the values contained in the request. As the algorithm runs, it makes requests for data directly to a *resource interface*. The resource interface should be the only format dependent portion of the system. The Data Analysis Agent continues until it has completed its task and found all statistically significant trends, then returns the results to the Task Agent.

4.4.6.1 *Resource Interface* The resource interface is encapsulated by the Data Analysis Agent and holds all the information needed to interface with the specific format data source it is responsible for. If it is flat-file, the resource interface must be specialized for flat-file access and could not talk or retrieve data from a relational data source. Similarly, a resource interface responsible for a relational data source knows how to specifically retrieve information from a relational data source, no other format. These agents are not specific to data mining but rather are able to answer any query into its data source. If a new data source is introduced into the system, it must include a Resource interface that is capable of communicating with Data Analysis Agents. Additionally, Resource interfaces must be able to respond to queries from the Ontology Agent concerning the random variables within the data source (its domain). Upon entry into the system, a Resource interface must announce itself to the Registration Agent so that it can be recognized by the system and the Broker Agent can add it to the list of system agents. Once it has registered, the Ontology Agent will query the resource interface for the domain of the data source for which it is responsible.

4.4.7 Unification Agent When the Task Agent has received all the results from the Data Analysis Agents, it passes them to the Unification Agent. The Unification Agent contains all algorithms for unifying the data. It performs this unification on the results before passing them back to the user agent. Initially, the Unification Agent will look at results for rules that are the same but from different data sources. It combines the results from each source into one rule that blends the support and confidence levels based on number of transactions from each source. For instance, if data source A has a rule $X \Rightarrow Y$ meeting minimum support and confidence levels based on 10,000 transactions, it should get more weight that data source B's rule stating $X \Rightarrow Y$ based on 100 transactions.

4.4.8 Determining the Development Framework There are key differences between JAFMAS and JATLite that must be considered before selecting one development framework over the other. The biggest difference is the use of a centralized router or server for agent identity and message routing. JATLite uses Java's Server socket and Socket classes while JAFMAS uses Java's Remote Method Invocation (RMI) and MulticastSocket class. JATLite uses the sockets to talk with the centralized Agent Name Server for establishing agent identity. This allows for a single point of failure in the system. It also uses a centralized router for communication between agents. Again, this creates a single point of failure for the system. If either the ANS or router goes down, the system cannot function.

On the other hand, JAFMAS uses multicast messaging to establish agent identity. This allows an agent to fail without affecting the agent system as a whole. Additionally, JAFMAS does not use a centralized router for agent communication, but rather uses the JAVA RMI Registry that is located on with each agent. Again, this removes the single point of failure from the communication architecture. Both, through the use of Java, allow for a multi-platform

framework, as well as use of key Java features such as Java Database Connectivity (JDBC) for extensibility.

This research uses the JAFMAS framework for robustness reasons. A system should not be reliant on a singular component. JAFMAS provides this, as well as the ease and extensibility of Java. The following table compares key features of JATLite and JAFMAS [JAF97]:

	JATLite	***JAFMAS***
Java Version	JDK 1.2	JDK 1.2
Each agent has it's own thread?	Yes	Yes
Communication between agents?	Centralized Router	Directed or multicast
Means of directed communication?	Uses Java's Serversocket and Socket classes	Uses Java's RMI and MulticastSocket
Peer-to-Peer communication?	No	Yes
Agent identity established through?	Registering with a centralized Agent Name Server (ANS)	Multicast messaging
Subject-based addressing supported?	No	Yes
Speech-act type supported?	Yes	Yes
Security features	User name and password check provided apart from using the Java security features	Relies on Java security features.

4.5 Define Agent Conversations and Data Structures

Once the primary functions and high-level interactions are defined for each agent, the conversations required to fulfill these functions must be defined. Each agent is discussed and all conversations required for that agent are presented.

Each conversation is graphically depicted in a State Transition Diagram (STD). Each diagram can be labeled one of two ways. First, if it depicts a conversation for an agent making a request or initiating a conversation, it will be labeled 'Receiver:' and includes the type of the agent to which it is sent. If it is a conversation to which the agent is reacting to, it is labeled 'Initiated by:' and includes the type of agent that sent the conversation-initiating message. The double circle (state S_2 in Figure 13) is used to indicate a final state for any particular conversation. A transition occurs on a message receipt or send action. Each transition can include any of the values specified in the following list:

Start: indicates the start of a separate conversation. Conversation is launched within the existing conversation, which then waits until the conversation just launched completes.

Send: the performative being sent. If the agent is the initiator, this will be the first transition. It is represented by the *Type* field of a message.

Receive: the performative received. If the agent has not initiated the conversation, this will be the first transition. Again, contained in the Type field.

Content: Optional. May contain some data structure or information.

Intent: Optional. May contain some value or even a data structure.

SuchThat: Optional. Used only from intermediate states in a conversation. Transition occurs only if the *suchThat* evaluates to True.

At each intermediate state, there can exist a *do* action. This is a method or action to be performed upon reaching the given state. It is performed by the agent for whom the conversation is modeled. Any state with a *do* action must have at least one transition out with a *suchThat*

clause to allow for completion of the action. In some cases, a cancellation could cause a transition without completion of the method or action.

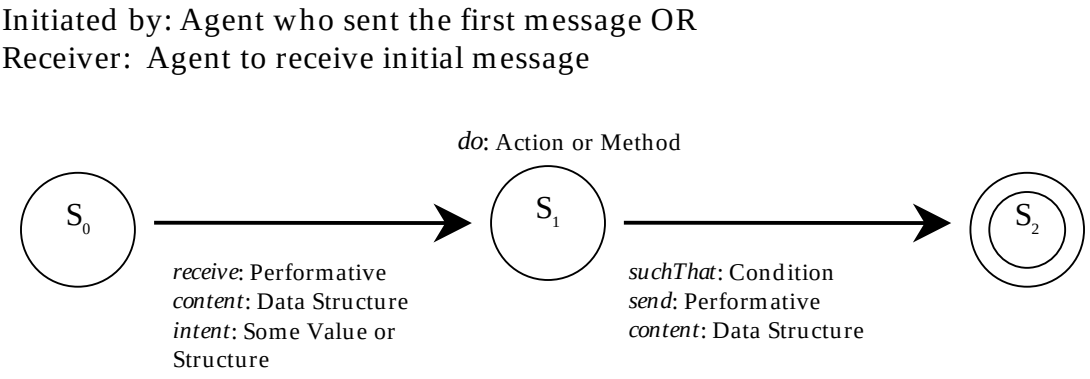

Figure 13 Generic STD for a Conversation

This section first covers the JAFMAS provided abstract class Agent and the methods it requires to be implemented in each agent instantiation. It then outlines the common conversations that any agent can undertake. The first of these is making a request to the Broker. Section 4.5.2 outlines how the Broker handles this and how it was made into a generic process. It includes a generic STD that is representative of the conversation in which any agent engages when making a request to the Broker. The second common conversation is the Registration Conversation. Again, this is overviewed and a generic STD is shown that reflects the conversation in which any registering agent engages. Finally, it discusses each individual agent and the conversations in which it engages and shows a STD from that agents viewpoint.

4.5.1 *Using the Abstract Agent Class* All agents in the system are extensions of the JAFMAS provided Agent Class. This is an abstract class that provides the implementation of all communications an agent requires. An abstract class is a class that encapsulates a concept,

but does not allow instantiation. For instance, food represents the abstract concept of things that we all can eat. However, it doesn't make sense for an instance of food to exist. What we would like to exist is instances of classes of food, such as cake, apples, and oranges. An abstract class may contain *abstract methods*, or methods with no implementation. In this way, an abstract class can define a complete programming interface, providing its subclasses with the method declarations for all of the methods necessary to implement that programming interface. However, the abstract class can leave some or all of the implementation details of those methods up to its subclasses.

The abstract agent class provides the methods and implementation to allow initialization and communication, however it requires any subclass to implement the *startConversation* and *addSubjects* methods. These methods will be specific for each subclass of agent that can exist. When a new agent is created it will then inherit all methods of the abstract agent class.

4.5.2 Utilizing the Broker Agent Before each conversation is initiated, the sending agent must discover exactly what agents should receive the message. To do this, it asks the Broker Agent to pass it the names and locations of all agents of a certain type. This occurs every time an agent initiates a conversation to ensure the proper agents receive the message and enables new agents to be immediately recognized by all agents in the system. Because this is a repeatable process, a generic conversation class was created that any agent can use to query the Broker. This ensures any new agent can immediately talk with the broker by simply instantiating the conversation class and forces proper formatting of any requests. Improperly formatted requests will not be recognized by the broker and hence go unfulfilled. It also creates an extensible framework for any system that may use a Broker-Centric hierarchy. The generic conversation is modeled below (Figure 14) and is not reflected in the conversation STD's for each individual conversation. Figure 23 shows the conversation from the Broker Agent perspective.

Initiated by: Any requesting agent

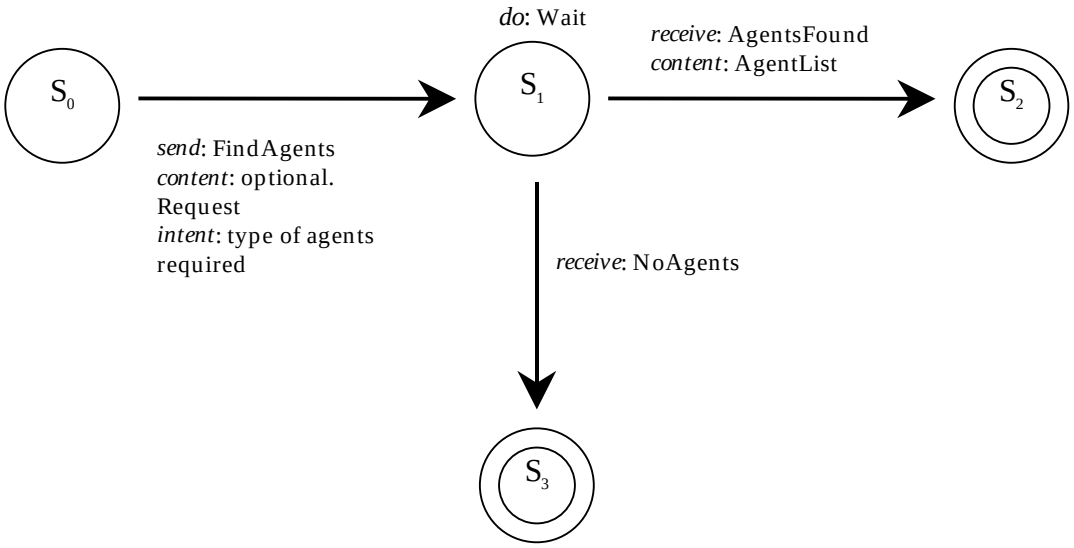

Figure 14 Agent to Broker Conversation

All agents must have the capability to register in the system. This section discusses how this is accomplished in more detail from a conversation viewpoint, including some design decisions that were made. One goal in this area was a common registration conversation that could be instantiated by any agent (existing types or new). In order for this to occur, the common information required to register must be made available through methods associated with the registration conversation. The agent registration process is covered first, including what agents are notified of any new registration and the information required from the registering agent. Next is a look at the conversations that must occur for a successful registration, including the values each message must contain. Finally, why and how this process was made a default part of each agent is discussed.

4.5.3 *Registration Agent* The Registration agent is responsible for ensuring every agent that needs to be aware of a new agent, is made aware. The functions described in Section 4.4.1 dictate three possible agent conversations. First is the new agent conversation (Figure 15). The conversation in Figure 15 is from the registering agents perspective and is shown as a generic STD since every agent will use the same conversation when it enters the system.

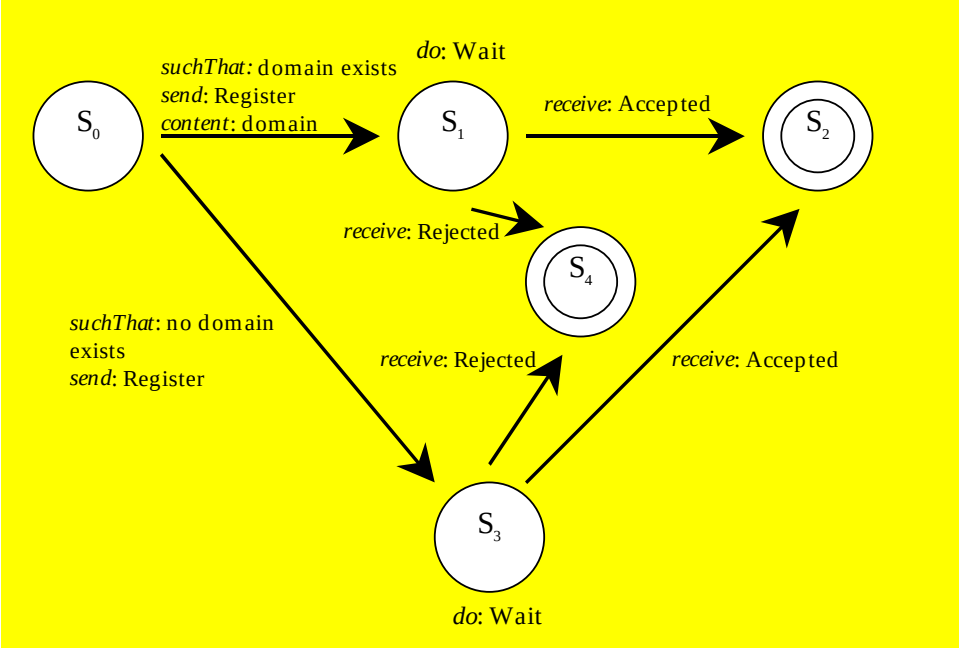

Figure 15 Agent to Registration Conversation

When a new agent is created, it automatically locates the Registration Agent by means of a broadcast request to the system. The directed communications module of JAFMAS automatically responds to this request with the Registration Agent's full name. The new agent then uses this information to send a "Register" performative message to the Registration Agent. The Registration Agent must recognize this request and, based on the type of agent requesting registration, initiate conversations with the Broker and/or the Ontology Agents. Figure 16 shows this conversation from the Registration Agents perspective.

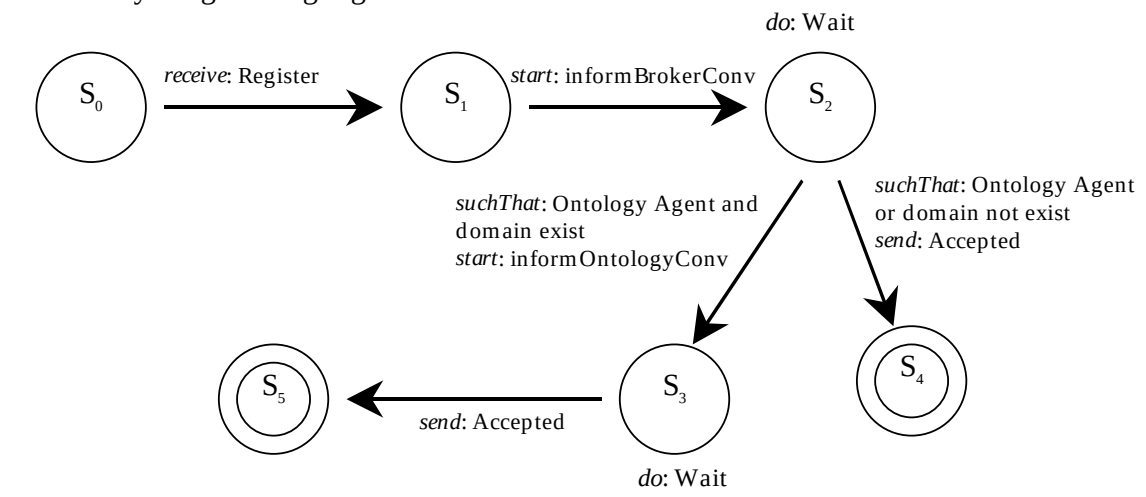

Figure 16 Registration to Agent Conversation

Figure 17 shows the conversation the Registration must initiate with the Broker Agent to inform it of a new agent s arrival. As previously mentioned, the Ontology Agent should only be informed when an agent which has a domain associated with it enters the system. In this case the Registration Agent initiates a conversation with the Ontology Agent (Figure 18), passing the applicable information (domain, name and type). Once the Registration Agent has received an acknowledgement from the Ontology Agent, it ends the conversation.

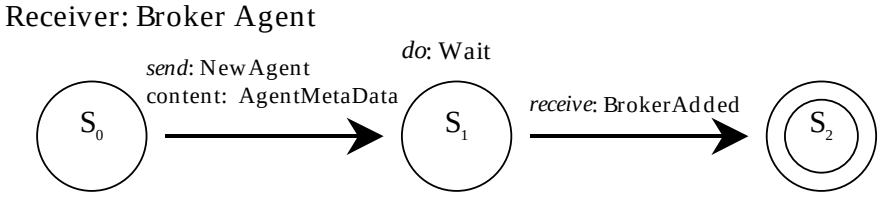

Figure 17 Registration to Broker Conversation

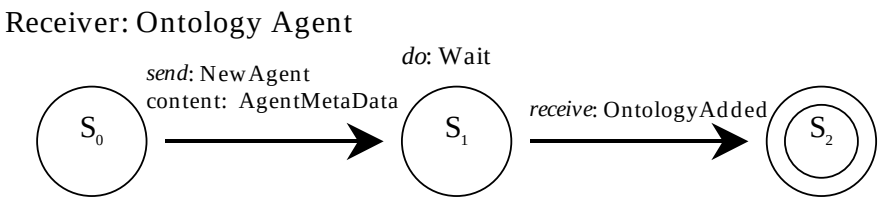

Figure 18 Registration to Ontology Conversation

In order to ensure all agents have the ability to register and all required message fields are set properly, the Abstract Agents class was extended to include the method register(). Since all new agents in this system are based on this class, they inherit this function and only need to ensure that the method call is included in the appropriate place. Most likely this will be in the agent constructor.

4.5.4 User Agent

4.5.4.1 *Conversations* The User Agent is unique in that it must understand conversations in two forms – from other agents, as well as from the requesting application itself. Because the communication with the application will only consist of getting requests and passing back results, it is not modeled here, but it must be considered a line of communication nonetheless. The user agent interacts with only one agent, the Task Agent. Figure 19 shows the STD for the conversation. Once a request from the application is received, it initiates a request for mining based on the values passed to it by the application with a *DoMine*. The Task Agent will acknowledge receipt and the User Agent waits until either results are returned, or it is informed there are no data sources can provide information (with a *NoAgents*). It then passes the results back to the application. In the case of no results it sends back an empty results list.

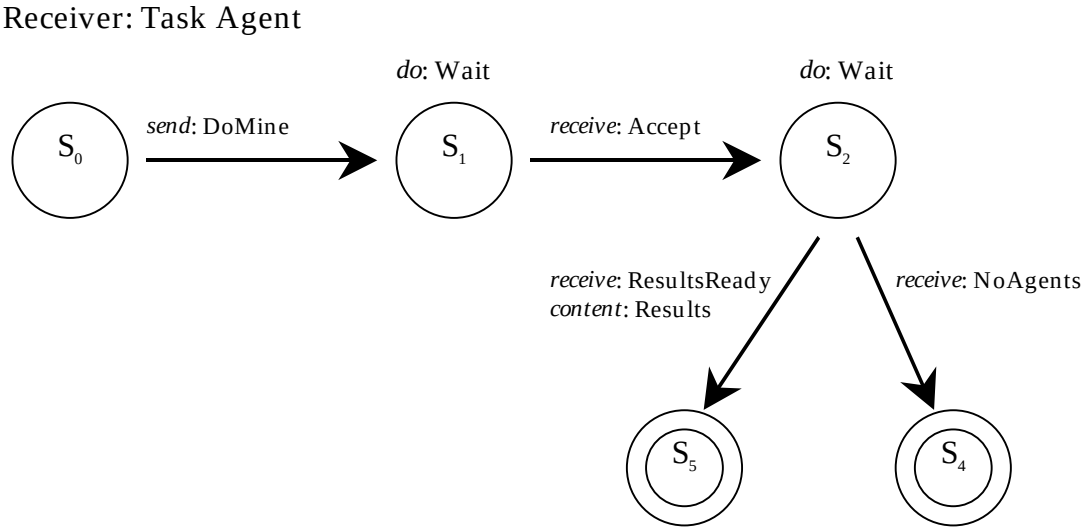

Figure 19 User to Task Conversation

4.5.4.2 *Data Structures* The conversations in which the User Agent engages dictates a number of data structures. When the User agent first picks up the request from the application it must convert it to a request the agent system can understand. The Request Class is used to store this request as it passes throughout the system. It will have the following structure:

Request Class: Includes Support, Confidence, X, and Y, where X and Y are either values or null, depending on the request being made, i.e. if X = (sidewalk,dry) and Y = null, then we are looking for any Y value in which (sidewalk,dry) $\Rightarrow$ Y is true for the support and confidence values.

In order to store the results of the data mining process, it must also utilize a class that allows for a list of individual rules. This class is RulesList and has the following structure:

RulesList: Array of unspecified length with each element consisting of an individual Rule

Rule: Each rule has a Support, Confidence, Antecedent, and Consequent.

4.5.5 Task Agent

4.5.5.1 *Conversations* The Task Agent interacts with the user agent, the Broker Agent, the Data Analysis Agent, and the Unification Agent. Ultimately the Task Agent initially reacts to a conversation initiated by the user agent as depicted in Figure 20.

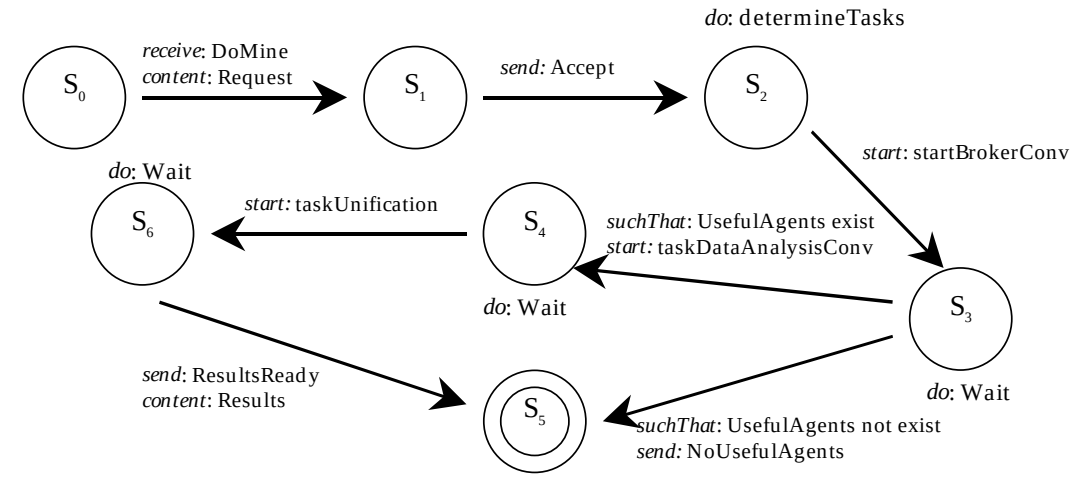

Figure 20 Task to User Conversation

The user agent will request that some data mining operation be completed with a *DoMine* and will pass the appropriate variables dependent on the type of operation as discussed in 3.4.1.2. In response to a request for completion, the Task Agent determines the type of request being made from the variables sent. Once it has determined the type of the request, it informs the User Agent it has all the information it needs with an *Accept* and begins processing.

The Task Agent then initiates a conversation with the Broker Agent to determine what Data Analysis Agents it should task for the given request (Figure 21). It awaits the results, then ends the conversation. It can receive one of two possible messages as results. It can receive a message of *AgentsFound* and list of useful agents, or a *NoAgents* message, indicating there are no data sources that could mine association rules for the variables given. If *NoAgents* is received,

the Task Agent sends a *NoUsefulAgents* message to the User Agent and ends the User Conversation.

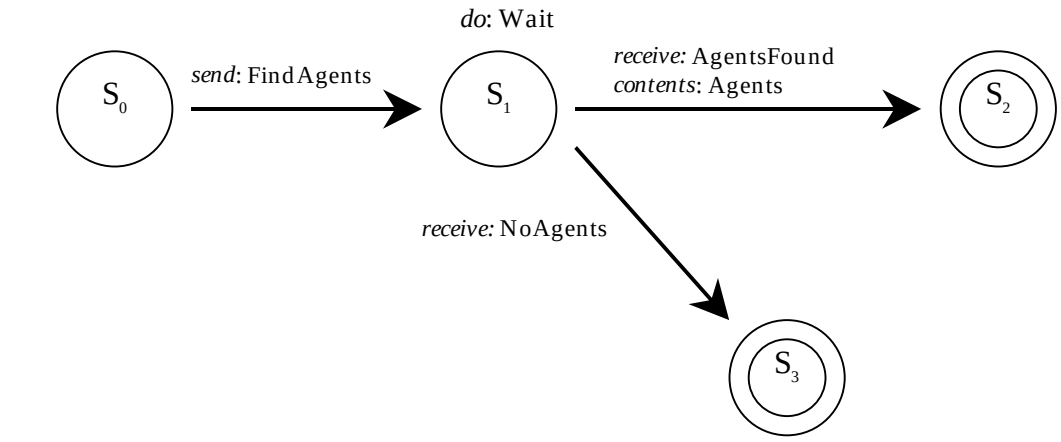

Figure 21 Task to Broker Conversation

If agents were found (*AgentsFound* received), it then initiates a conversation with each of the Data Analysis Agents returned to it from the Broker Agent. It requests that the agents begin data mining for the values in the original request. It can pass none, one, or two random variables with associated support and confidence levels for the Data Analysis Agent to mine. This is again dependent on the type of request the user has selected as discussed in 4.4.2. Once it receives the confirmation from the Data Analysis Agent, it awaits either results or a conversation initiated by the user agent requesting a termination of the current data mining operation.

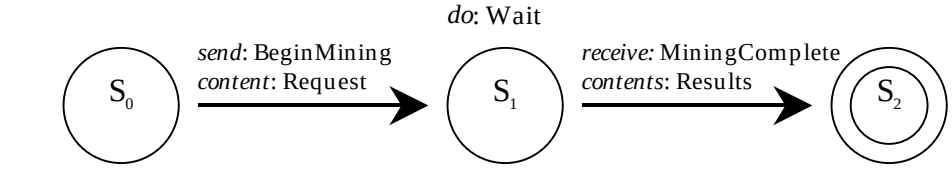

Figure 22 Task to Data Analysis Conversation

After terminating all conversations with the Data Analysis Agents, the Task takes the results and passes them to the Unification Agent. It initiates a conversation with a *Unify* performative and waits until it receives a *UnificationComplete* message with the unified results included (Figure 22). Once it receives the results and terminates the Unification Conversation, it sends a *ResultsReady* message to the User Agent and ends the conversation. Once the User Conversation is terminated, the Task Agent waits until another *DoMine* message is received.

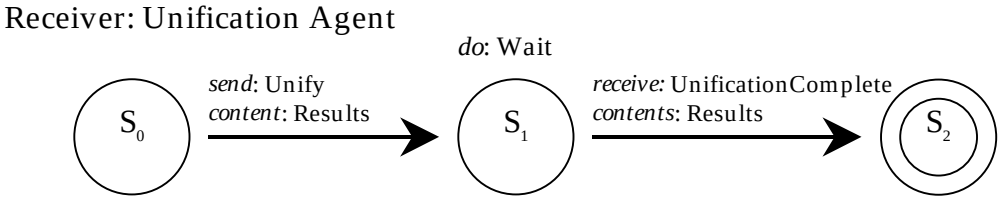

Figure 23 Task to Unification Conversation

4.5.5.2 *Data Structures* The Task Agent must handle virtually every data structure the system has. It uses the structures from the User agent (Request and RulesList) as well as several others. First, it must be able to handle a list of agents returned from the Broker

Agent. It also must be able to handle a list of RulesList in order to pass them to the Unification Agent. These structures are shown below:

AgentList : Array of unspecified length with each element containg AgentMetaData

AgentMetaData: Record that contains agent name, function or tasks performed, and a domain if one exists.

AllRules: Array of unspecified length with each element containing a *RulesList* (see 4.5.4.2)

4.5.6 Broker Agent

4.5.6.1 *Conversations* The Broker Agent primarily interacts with the Task, Ontology, and Registration Agents, but should ultimately respond to any properly formatted request for agents that can fulfill a given task. As mentioned before, the primary functions of the broker are to (1) maintain a list of all agents in the system and the tasks they perform and (2) answer queries requesting agents that can fulfill any given task.

To perform the first function, the Broker must be able to communicate with the Registration Agent and receive new agent's information. This conversation is initiated by a *AddAgent* message that contains the new agent's full name and task list (Figure 24).

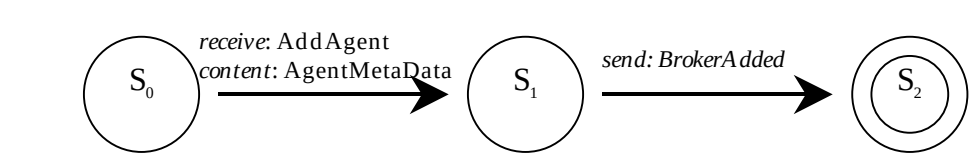

Figure 24 Broker to Registration Conversation

The new agent's information is taken from the message content, the global list of agents is updated and an acknowledgement is sent to the Registration Agent. The acknowledgement is simply a message entitled *BrokerAdded.* Once the message is sent the conversation is terminated.

The second task is more complex. When an agent makes a request for information, the Broker must get the request, process it, then determine if the Ontology Agent should be utilized. This conversation is initiated by a request message titled *FindAgents* from an agent in the system and is shown in Figure 25.

When this message is received the broker processes the information and then, based on the result of the content of the message, either begins another conversation with the Ontology Agent, or sends the results to the requesting agent.

Initiated by: User Agent

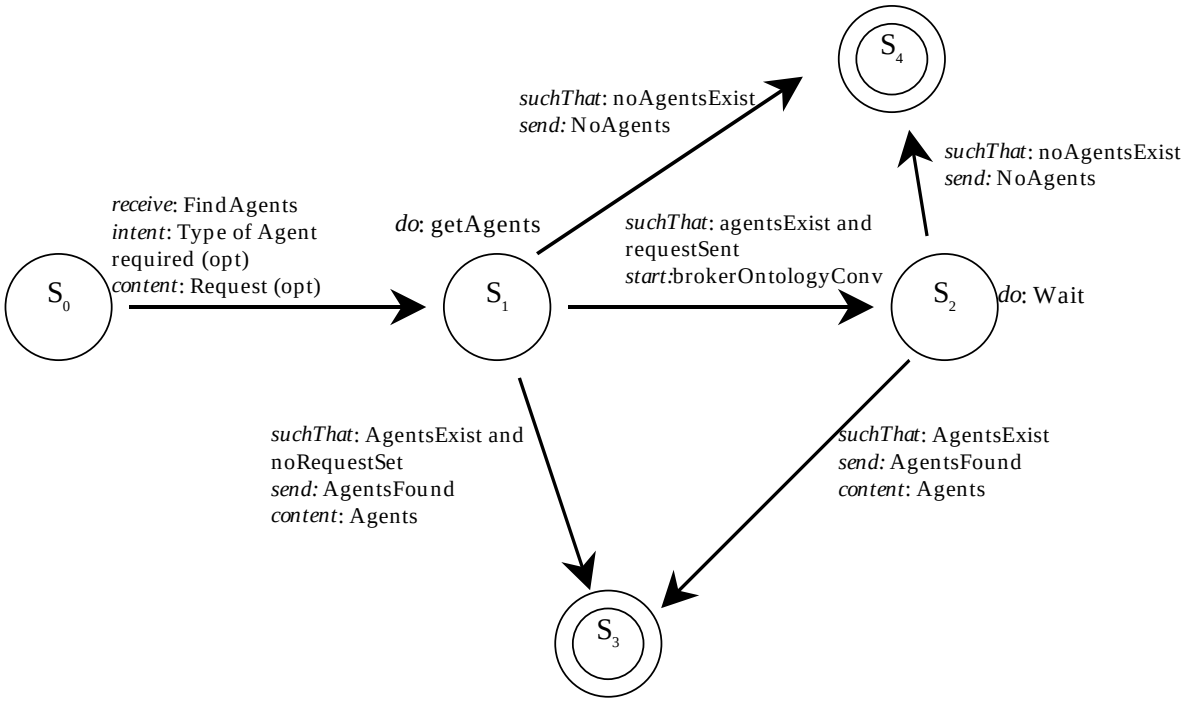

Figure 25 Broker to Requesting Agent Conversation

If the content was set and included a request with the X, Y, or both values set, then the Ontology Agent must be utilized. The Broker initiates a conversation with the Ontology Agent with a *CheckDomains* message. This conversation is shown in Figure 26 below.

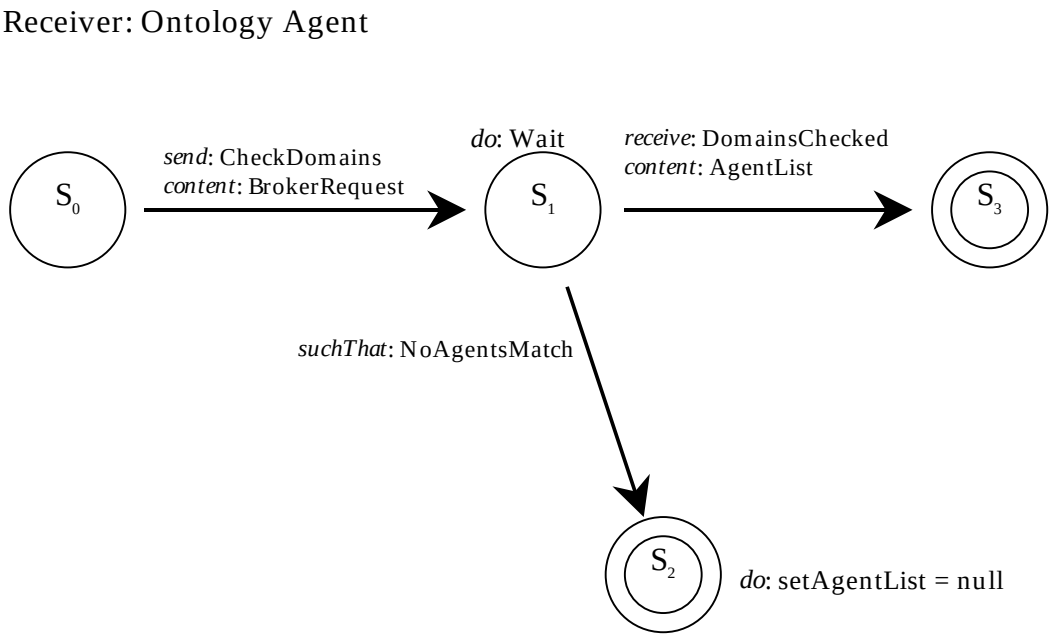

Figure 26 Broker to Ontology Conversation

Once the Ontology Agent has checked the domains it sends a *DomainsChecked* message and returns a list of all useful agents for the given request values or a *NoAgentsMatch* message. A *NoAgentsMatch* message implies none of the agents' domains are applicable to the original data mining request.

The results, regardless of if the Ontology Agent was consulted, are sent back to the requesting agent. If no agents were found, a *NoAgentsFound* message is returned. If there are useful agent, then an *AgentsFound* message is sent and the agent list is included as the content. Once the agent list is returned, the conversation is terminated.

4.5.6.2 *Data Structures* The Broker Agent does not use any "new" data structures. First, it utilizes the Request Class (see Section 4.5.4.2) for storing a request to pass on to the Ontology Agent. It also uses the AgentList class (see Section 4.5.5.2) to store the list of agents currently in the system as well as to store the agents that it found to fulfill a given function.

4.5.7 Ontology Agent

4.5.7.1 *Conversations* The Ontology Agent interacts with the broker and the Registration Agent. It is mainly responsible for helping the Broker Agent determine what agents will be able to provide useful association rules. The Broker Agent will initiate a conversation requesting that the Ontology Agent check a list of Data Analysis Agents to see if the domain contains the X, Y or both values. The Ontology Agent will receive a *CheckDomains* message with a list of agents as the content as shown in Figure 27. It checks the list, then reports the results back to the Broker. If no agents were found to be useful, then a *NoAgentsMatch* message is returned containing a null agent list. If there are Data Analysis Agents that are useful, the Broker is informed with a *DomainsChecked* message and passed the list of useful agents as the content.

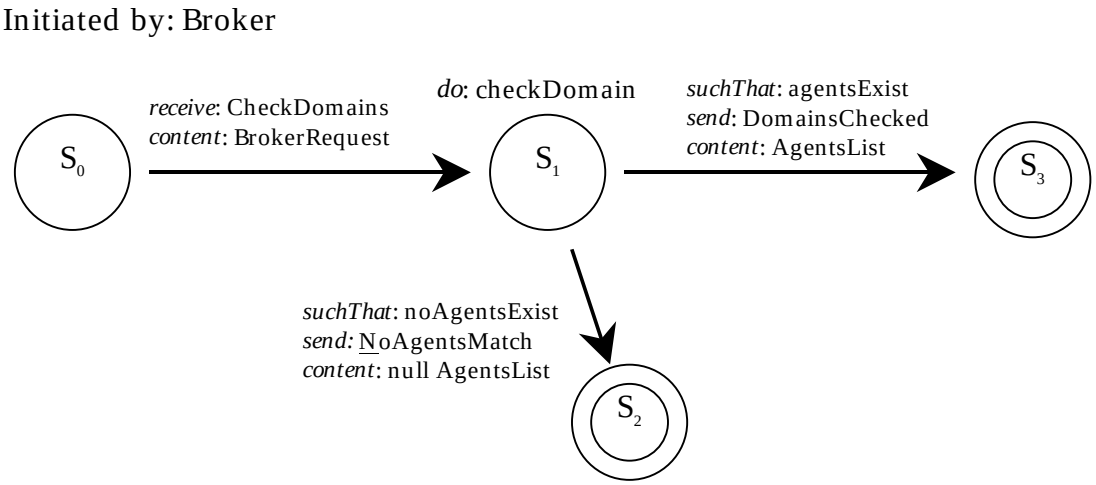

Figure 27 Ontology to Broker Conversation

The Ontology Agent must also be able to communicate with the Registration Agent to receive new agent's domain information. This conversation (Figure 28) is initiated by a *NewAgent* message that contains the new agent's full name and domain. The new agent's information is taken from the message content, the global list of agents is updated and an acknowledgement is sent to the Registration Agent. The acknowledgement is simply a message entitled *OntologyAdded.* Once the message is sent the conversation is terminated.

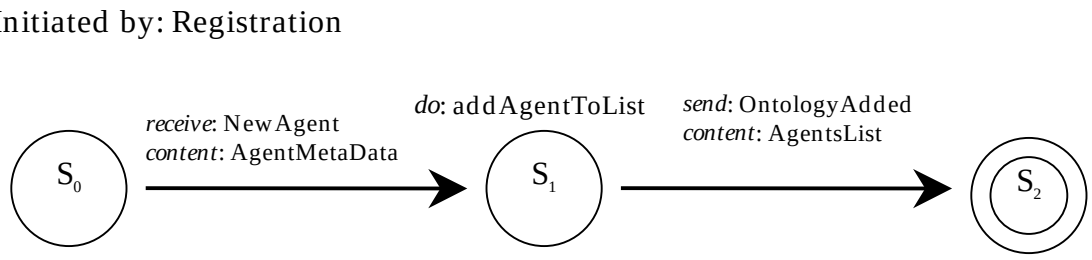

Figure 28 Ontology to Registration Conversation

4.5.7.2 *Data Structures* The Ontology Agent requires the same classes as the Broker Agent and does not introduce any new requirements. It uses the AgentList (see Section 4.4.5.2) to maintain the agents, their domains, and the Request class (see Section 4.5.4.2) to check the X and Y values against the domain. The AgentList class (see Section 4.5.5.2) is also used to store the list of agents passed to it by the Broker Agent.

4.5.8 Data Analysis Agent

4.5.8.1 *Conversations* The Data Analysis Agent must interact with the Task Agent and the resource interface. As mentioned in Section 4.4.6, the Data Analysis Agent encapsulates the resource interface and data mining algorithm instance for efficiency. Because of

this, there are no conversations between the resource interface, data mining algorithm, and the Data Analysis Agent. Again, this would add unnecessary overhead and delay an already lengthy data mining process. The Data Analysis Agent awaits a *BeginMining* message from the Task Agent (Figure 29). The message contains the original request as well. Once this is received, it starts the data mining algorithm associated with it. When it is completed, it sends a *MiningCompleted* reply with the results as the content.

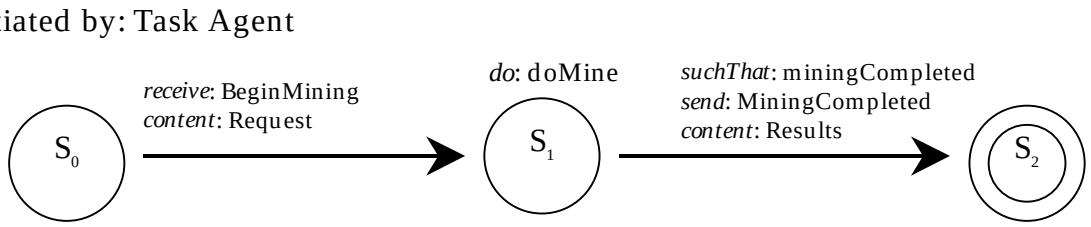

Figure 29 Data Analysis to Task Agent

4.5.8.2 *Data Structures* The Data Analysis Agent does not introduce any new data structures. It does use the Request class (see 4.5.4.2) to hold the request passed to it by the Task Agent and the RulesList (see 4.5.4.2) to hold all rules it has found through the data mining process.

4.5.9 Unification Agent

4.5.9.1 *Conversations* The Unification Agent is utilized only at the very end of the entire process. It is contacted only after all Data Analysis Agents have completed their respective mining operations. The Task Agent will send a Unify message with all results in the content to the Unification Agent (Figure 30). Once it receives the results, it performs the

unification and returns the unified rules as the content of a *UnifyCompleted* message. This is the only conversation in which the Unification Agent engages.

Initiated by: Task Agent

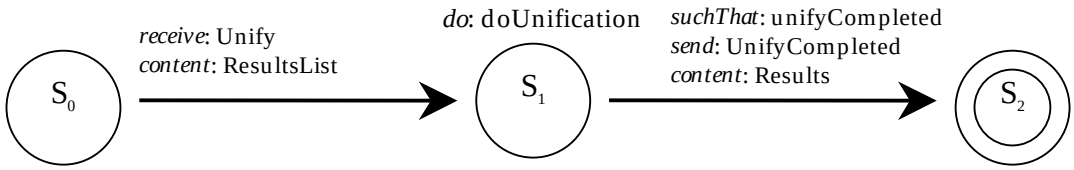

Figure 30 Unification to Task Conversation

4.5.9.2 *Data Structures* The Unification Agent must be able to accept the combined RuleLists (see 4.5.4.2) of all Data Analysis Agents as well as pass back one unified list of rules. It uses the AllRules (see 4.5.5.2) class to store the combined RuleLists from the Task Agent, then uses the RulesList to pass back the unified rules.

4.6 Detailed Agent Design

The final step in the methodology is to implement the algorithms that will allow each agent to perform the functions specified in the previous steps. Each specific type of agent is a subclass of the abstract Agent class with additional methods and structures identified as needed. The specification of the methods is outlined for each agent type and any algorithms used are specified.

Each agent will be discussed and the methods to be implemented shown and discussed. The data structures required for passing messages have already been discussed for each agent in section 4.5. They will not be included again here as it would be redundant. Similarly, the conversations are not included as they are specified in each section in 4.5. The performatives

expected and the method calls are shown for each conversation. The details of a conversation in general are discussed in the next section and are applicable to all the agent conversations

4.6.1 *Common Detailed Design Requirements* All agents must include implementations of the abstract methods specified in the JAFMAS provided abstract Agent class. The following methods must be implemented in each agent:

```
public abstract void startConversation(Object ob);

public abstract void addSubjects();
```

The addSubjects method adds multicast subjects or groups to which agents in the multi-agent system can subscribe. In this research the use of this was avoided for the reasons specified in Section 4.4.4.1. Each agent will subscribe to a group known as *ThesisNetwork*. Additionally, the *startConversation* method starts the conversation thread in the Task Agent depending upon the message received. This is unique for each agent dependent on the conversations defined in Section 4.5.

4.6.2 Agent Conversation Detailed Design Each conversation is a subclass of the JAFMAS provided abstract Conversation class. The only abstract method in this class is the *initializeRules* method. A rule defines a state transition in the conversation diagrams shown throughout Section 4.4. Each rule is based on the abstract *ConvRule* class and provides several methods that may be extended. The conversation rule methods used in this research are the *suchThat*, *setRecvdMessage*, *doBefore*, *doAfter*, *findRecvdMsgmatch*, and *setTransmitMessage*. Because there is a common rule for each transition, how to construct the detailed conversation design from each diagram in Section 4.4 is shown here once.

The first method is the *suchThat* method, this provides an initial transition test. It is specified on the transition and must be true for the rule to fire. If it is not specified or if it is true, the *doBefore* method is executed if it exists. This method typically calls any unique methods internal to an agent. If the rule defines a new conversation with another agent, it is created and started here. In the diagrams this new conversation is shown in the *start* field. After this is completed the *findRecvdMessage* method executes and checks for a message and extracts any contents. It makes a call to *setRecvdMessage* that checks to see if the message contains the appropriate performative. If not, the *findRecvdMessage* method fails and the rule fails. The value expected by the *setRecvdMessage* is shown in the receive field of a transition. Any content to be extracted by a *findRecvdMessage* method is shown in the content field. Finally, any messages to be sent back is done so with a *setTransmitMessage* method. This is reflected in the diagram by a transition with the only a send value specified.

In this way each conversation can be created and the rules that define the conversation specified.

4.6.3 *User Agent* In order to fulfill the tasks specified in Section 4.4.2, the User Agent must have several methods. The agent must take a request from the external application and convert it to a *RequestClass* object so it can be passed to the Task Agent. This is

accomplished with a setRequest method that simply takes in the values for confidence, support, and any X and Y values and instantiates a variable of type RequestClass, assigning the values passed in to it.

In the agents constructor it performs the standard start-up commands, then awaits a Task Agent to enter the system. It makes a request to the Broker Agent for the name of the agent that can fulfill the task of Task Agent and waits until an agent can be found. Because the User Agent relies on a Task Agents presence, when starting the system, the Task Agent should be created prior to the User Agent. This is not mandatory, but should be accomplished

The required method *startConversation* begins when a Task Agent has been found. It waits notification a request has been made, then calls the *setRequest* method. Once the request is set, it initiates the User to Task Conversation specified in Figure 19 and discussed in Section 4.5.4.1. When the results are returned to the User Agent, it sets them to a local variable through the *setResultsList* method. This ensures the values are stored in the case the external application is not ready to accept the values or they are corrupted in transmission.

4.6.4 *Task Agent* The role the Task Agent fills is discussed in Section 4.4.3. It must a initialize itself in the constructor, then await a message from the User Agent. The *startConversation* method awaits a new message, then begins the Task to User Conversation shown in Figure 20 and discussed in Section 4.5.5.1.

Upon receipt of a request, it starts the Task to Broker Conversation. The result of this conversation is a list of agents. If none exist, it is null and when the Task to Broker Conversation ends, the Task to User Conversation rule checking for a null agent list fires and tells the User Agent that no useful data sources exists and that conversation terminates. If there are useful sources, the Task to User Conversation Rule to begin the Task to Data Analysis Conversations

fires. This passes the request to the Data Analysis Agents and awaits results. When results are received the Task to Data Analysis Conversation ends and the Task to User Conversation Rule to begin a Task to Unification Conversation fires. The Unification Agent unifies the results and passes them back. Once received the results are sent to the User agent and the Task to User Conversation Terminates.

Because the Task to User Conversation calls all other conversations, the data structures being passed do not need to be set in the Task Agent itself. Instead the Task to User Conversation class must implement the methods that set, store, and retrieve the values. These are implemented with traditional *get* and *set* methods such as *setResults*.

4.6.5 *Broker Agent* The Broker Agent must check to see if agents are in the system and meet some specific criteria. To do this it must have several unique methods. When an agent is added to the system, the Broker to Registration Conversation must add the agent information to the list of agents. This is done through the *addAgent* method. This method simply performs an *addElement* to the *agentList*.

The startConversation method must accept two possible messages to start either of its two conversations. The first is an *addAgentInfo* message that starts the Broker to Registration Conversation (Figure 24). The second is a *FindAgent* request and should start the Broker to Requesting Agent Conversation (Figure 25).

If an agent requests other agents with certain capabilities, the Broker to Agent Conversation begins, receives the request, and must call methods that check the list for agents fulfilling the request. This is done with the *findAgents* method. This method takes the type of agent required, and searches the list of all agents for agents that can provide the task. Simply

looping through the array of agents, if a matching agent is found, it is added to an array of matching agents.

If, based on the conversation rules, it is determined the Ontology Agent must be contacted, the Broker to Agent Conversation must find the Ontology Agent. This is implemented as *findOntology* and simply calls *findAgents* with Ontology as the type. It returns the name of the Ontology Agent if one exists. The Broker does not require any other unique methods.

4.6.6 Ontology Agent The Ontology Agent tasks are discussed in Section 4.4.5 and the conversations and data structures are shown in Section 4.5.7. The Ontology Agent, like the Broker, has two possible functions. First it must add an agent and its domain or check a list of agents domains against a particular request. The second conversation is the Ontology to Broker Conversation that takes a list of agents and checks the domains of those agents against the original PESKI request. In order to do this, it must provide a checkDomains method that accepts an agentList and X and Y values. It will simply loop through the agentList passed to it from the Broker, find each agent in its agentList, then see if X or Y are in that agents domain. If so, it is added to the returnAgentList and passed back. The startConversation must be defined to accept either message.

The first will be a message with a performative of *addAgentInfo*. This will start the Ontology to Registration Conversation shown in Figure 28. The only method needed in support of this is to add an agent and domain to the list of system agents and their domains. This is accomplished with the *addAgent* method that simply performs an *addElement* to the *agentList* array.

4.6.7 Registration Agent Because of the Registration Agents role in the system (see Section 4.4.1) it must be created first. Upon creation, it must ensure a Broker Agent is

created as well. This can be accomplished in the constructor and a *findBroker* method must be implemented. This method should wait until a message is received on the *ThesisNetwork* stating the Broker has entered the system. Once found, the Registration Agent can await new agents to enter.

The *startConversation* method should await only one type of message, the *FindAgents* message. Once it is received, it begins the Registration to Agent Conversation shown in Figure 16. This conversation launches the Registration to Broker Conversation shown in Figure 17 This conversation simply passes the agent information and does not require any new methods.

The only other method required is the *findOntology* method. If it is determined the Ontology Agent must be contacted, the conversation needs to know the name of the Ontology Agent from the Registration Agent. The Registration Agent can maintain this by checking each registering agent type for ontology. Once the Ontology Agent registers, the Registration Agent stores the name and returns it if *findOntology* is called.

4.6.8 Data Analysis Agent As discussed in section 4.4.6, the Data Analysis Agent is simply a container agent for the resource interface and data mining algorithm. The only unique method it must have is a *beginMining* method which calls the *doMine* method of the mining algorithm.

One unique aspect of the Data Analysis Agent is the fact it holds a resource interface instance and a mining algorithm instance. The constructor must take in these values, then assign them to local variables so it has visibility to them. Additionally, the data mining algorithm must have direct visibility to the resource interface and when it is assigned, the Data Analysis Agent must pass in a pointer to the resource interface. Once the initialization is done, *startConversation* waits for a *doMine* message. Currently, the *doMine* message is the only message the Data

Analysis Agent can expect to receive. After the data mining algorithm is done, the Data Analysis Agent passes the results back to the Task Agent.

4.6.8.1 *Resource Interface* Currently, the resource interface is encapsulated in the Data Analysis Agent, and as such it is covered as a subsection of the Data Analysis Agent. It does not have any explicit conversations as the Data Analysis Agent handles all communication for it. It is an abstract class that is manipulated through method calls from the data mining algorithm. This component will be one of the components required when the system is extended and a new data source is added. Because of this, special consideration must be given to making it as extensible as possible. The class should define abstract methods for any methods a new resource interface must have to ensure compatibility with the data mining algorithm. There are several requirements. First, it must be able to return the domain of the data source for which it is responsible. In order to ensure this, the abstract method *readDomain* must be implemented that reads the domain from the data source. Because the each data source may store its domain differently or not at all, the resource interface must know how to retrieve this. Each retrieval method may be implemented differently, hence the abstract method choice.

The next method it must have is a *nextTransaction* method. As the mining algorithm operated the mining algorithm over the data source, it will need to have access to each transaction. Again, this is data source dependent and is an abstract method. Each mining algorithm may need to make multiple passes through the data source and as such should be able to start with the first transaction. In order to ensure this interface is available an abstract *resetDB* method must be created. Finally, as a data mining algorithm runs, it should be able to query the data source to determine if there are any more transactions. To incorporate this, an abstract *moreTransactions* method is implemented.

The abstract methods and the fact the resource interface is implemented as an abstract class provides a great of flexibility. Each data source is accessed differently and if a Resource interface is instantiated, the implementations of the abstract methods allow data specific interface code to be incorporated, while ensuring the data mining algorithm can communicate with it.

4.6.8.2 Data Mining Algorithm The mining algorithm, like the Resource interface, is encapsulated by the Data Analysis Agent. It must be designed to be an extensible component as well, since it can be changed or modified for any associated Resource interface. It must be implemented as an abstract class for many of the same reasons as the resource interface. First, the Data Analysis Agent in which the algorithm is encapsulated must have a method that can be called to start the algorithm. This is implemented with the abstract doMine method. When called by the Data Analysis Agent, this method should start the algorithm operating over the data source.

Because the algorithm must make calls directly to the resource interface, it must be able to have direct visibility to it. It maintains this visibility by storing the resource interface as a local variable that is set through a *setResource* method. This is not an abstract method and is called by the Data Analysis Agent to let the mining algorithm know the resource interface it will be operating on.

The fact this is an abstract class allows any methods required in the implementation of a specific mining algorithm, to be added. It also ensures that the Data Analysis Agent can start the algorithm running, no matter what specific implementation is used.

4.6.9 *Unification Agent* The Unification Agent function is described in Section 4.4.7. As shown in Figure 30, the Ontology Agent receives the lists of all results from the Task Agent. Once it receives this list, it must perform the unification process. The *startConversation*

method is implemented to recognize that when a message is received, it launches the Ontology to Task Conversation, and extracts the resultsList.

The main method called *doUnify* must first check all results to see if there are any duplicates. If there are duplicates, it must unify all duplicate rules into one. This is done by weighting each rule by the number of transactions that contained the X and Y values. This means the higher the confidence and support, the more weight will be given to the support value found. A rule from a data source that has extremely high confidence and support will have more influence on the support and confidence level of the unified rule, as opposed to the same rule generated from another source.

Several methods can be created to support the above algorithm. First, a *checkForDuplicates* method will be implemented to check the results lists for all duplicate rules. Duplicates are passed to the unification method that will look at confidence and total transactions for each and provide a unified rule. Once the *doUnify* method is completed, it passes the structure back to the Task Agent and awaits another message.

4.7 Data Mining with Proposed Architecture

In order to understand how the process works, it is useful to trace through the communication paths a request would take. The process begins with the User Agent receiving notification from the application that a request needs processing. The User Agent picks up the application-formatted request and converts the data into a request of type Request Class so that the agent system can understand it. It then sends a message to the Task Agent. The Task Agent then asks the Broker Agent for all useful Data Analysis Agents. The Broker Agent receives the request, compiles a list of all Data Analysis Agents in the system, then checks to see if an Ontology Agent exists. If one exists, it sends the list of Data Analysis Agents, along with the

request for analysis. The Ontology Agent accepts the request and checks to ensure the Data Analysis domains against the request. It returns a list of useful agents to the Broker. The Broker then returns this list to the Task Agent. Once the Task Agent receives the list, it sends a request to each Data Analysis in the list to begin mining. Each Data Analysis Agent accepts the request and begins data mining their applicable data source. Once completed, the Data Analysis Agents sends the results back to the Task Agent. Once the Task Agent has all the results, it passes them to the Unification Agent. The Unification Agent processes the results, unifies them, and passes the results back to the Task Agent. The Task Agent then passes the unified results back to the User Agent. The User Agent then converts the results to a format the external application can recognize and notifies the application the results are available.

5 Implementation

5.1 Overview

One of the goals of this architecture was to provide an extensible framework that could more easily accept data sources of varying formats. This chapter outlines the features of the architecture that make it extensible. It also shows this extensibility through an actual implementation of a new data source. Section 5.2 will discuss the features of the framework that make DBMiner extensible. Section 5.3 will explain in more detail the agents that need to be created and all classes that require modification or addition to allow for a data source of varying format. An example of how DBMiner was extended is covered in detail in Section 5.3, showing the implementation of the extension.

5.2 Extensibility

In general, extensibility can be defined as the ease with which software can be modified to adapt to new requirements or changes in existing requirements. In the case of this research, the changing requirements are the number and formats of the data sources being mined. In order for this to be an extensible system, new data sources, or requirements, should be able to be added with relatively no changes or modifications to the existing system as a whole. One way this can be accomplished is to allow re-use of existing classes and data structures for the new data source. An analysis of what changes or extensions are required to accommodate new data sources, gives a better look at the ease or difficulty of the task of adapting to the new requirements. Subsection 5.2.1 looks at what components (agents) are affected when new requirements are added and the extensibility of those components.

5.2.1 *Effects of New Requirements* When a new requirement is added, it must be assessed for all areas it may impact. In the specific case of DBMiner, the most common new requirement, as mentioned, is a new data source. In order to add a new data source, we must look at how existing data sources are incorporated in the system, and attempt to incorporate the new source in the same manner, ensuring we fulfill all existing requirements for communication and interoperability. The ease with which the system allows this to happen can be effectively termed the extensibility of the system.

In the current architecture, a single Data Analysis Agent controls each separate data source. The Data Analysis Agent in-turn encapsulates and relies on an instance of a resource class and an instance of a mining algorithm. Thus, in order to bring a new data source into the system, a new Data Analysis Agent must be instantiated with the required components – a resource interface and mining algorithm. It must also register with the system to ensure the other agents are aware of its existence and the data source can be utilized to fulfill system goals. Looking at the system diagram in Figure 12, the only agents that could be affected by a new Data Analysis Agent is the Task Agent. The existing framework is designed to handle the addition a new requirement such as this without any changes.

First, we look at why no other agents are affected. In order for the new Data Analysis Agent to become a part of the system, the Task Agent, Broker Agent, and Ontology Agent must be aware it exists. The Task Agent is directly affected as it is responsible for tasking all useful Data Analysis Agents in fulfilling a data mining request. The Broker Agent is responsible for maintaining a list of all agents, and the Ontology Agent maintains a list of all agents with domains along with their respective domain. The Broker and Ontology Agent's are made aware of any new Data Analysis Agent upon its registration (discussed next). Once they are aware of the addition, the Task Agent is the only remaining agent affected. Every time a data mining

request is made, the Task Agent queries the Broker Agent for all useful agents. Once the new Data Analysis Agent is registered, it can be returned (and hence be visible) to the Task Agent, by the Broker, for communications. The Task Agent does not maintain a 'memory' of Data Analysis Agents that are in the system from request to request.

The registration of the new Data Analysis Agent has been made as easy as possible by placing all common method implementations in the abstract Agent class. All code required to register an agent is inherited by the abstract agent subclasses, which includes all Data Analysis Agents. Registration then becomes a one line entry, this.register(), in the constructor of the subclass. The Broker and Ontology Agents are notified of the presence through the registration process. By reducing the otherwise complex registration process to one line we add a great deal of extensibility.

5.3 Instantiating a New Data Analysis Agent

Because it is a subclass of the abstract Agent class, any Data Analysis Agent subclass includes all the logic and methods that allow it to communicate within the system. It also, by allowing any resource interface and mining algorithm to be encapsulated in it, gives the flexibility to be used by any format data source. This format independence comes from the fact the Data Analysis Agent acts only as an interface to the Task Agent to receive requests and pass back results. The resource and mining components contain any format specific code. Because of this, the only code required to create a new Data Analysis Agent and 'introduce' a new data source becomes a one line instantiation as shown below:

```
agent = new DataAnalysis(name, represents, subscribeTo, res, alg);
```

In the instantiation, 'name' is a string representing a unique name in the system. The 'represents' field is a list of all tasks the agent can perform and is set to "Data_Analysis". The 'subscribeTo' parameter allows the agent to subscribe to any "group" of agents within the system.

This research does not define any specific groups aside from the overall system. The ‘res’ and ‘alg’ are the chosen Resource interface and data mining algorithm. Currently, these values are all set through a graphical user interface but can be set manually as well.

This ease of instantiation provides extensibility to allow any data source of any format to exist in the system. Unfortunately, simply instantiating a Data Analysis Agent does allow a new data source to be mined, it only allows the system have visibility to the data source. Any format-specific code is included in the Resource interface and mining components that comprise the Data Analysis Agent. These aspects are discussed in the following two subsections.

5.3.1 *Resource Interface* The resource interface is the second facet in adding the new data source. This agent contains all code required to interface with the data source. It also must also include the standard interfaces to communicate with the Data Analysis Agent that encapsulates it, as well as the data mining algorithm that will operate over it.

In order to ensure all new resource interfaces can communicate in the system, an abstract resource interface class was created. Any new class must extend this abstract class and implement the abstract methods it includes. It is important to note that the abstract Resource class *is not* a subclass of the abstract Agent class. Because the Data Analysis Agent, who includes all required methods for communication, encapsulates it, the resource interface does not need to communicate with any other agents. It will receive all information it requires from either the Data Analysis Agent or the data mining algorithm that operates over it.

The abstract Resource class also includes several abstract methods that any subclasses must implement. This was done to ensure a common interface for the data mining algorithm class to operate across, independent of the data source format. The interface specifications for the required abstract methods are shown below:

```
public abstract boolean moreTransactions();
public abstract Vector nextTransaction();
protected abstract void readDomain(String db);
protected abstract void resetDB(String db);
```

These methods reflect the fact any new data source will most likely be transaction-oriented. Currently, while useful data may be extracted from other types of data sources, PESKI and BKB's require connections from transaction-oriented sources. If, at some time in the future, other classes of data sources needed to be added, this abstract class could be modified.

It is important to note that these abstract classes do not ensure or provide any optimizations in the data mining process. Any subclass can include any other methods or algorithms that would provide the optimizations needed or desired for particular data sources. Because optimizations, and often data mining algorithms, are format specific, these classes allow for any new methods or logic to be implemented, while retaining the ability to communicate with the system through the parent Data Analysis Agent.

The keys to extensibility in the Resource class are the flexibility for expansion and format specific optimizations, while enforcing the minimum system-specific method implementation possible.

5.3.2 *Mining Algorithm* The other key component, the mining algorithm, operates in much the same framework as the abstract Resource class. It is encapsulated by the Data Analysis Agent, and as such, has required interface methods any instantiation must contain to communicate in the system.

It is important to note here that it is the mining algorithm class that must have visibility to the resource interface it will be operating over. This is because the Resource class is purely query

oriented – it only answers queries made to it. It does not initiate any communication with any other agents or components of the parent Data Analysis Agent. The mining algorithm class can expect, at a minimum to have visibility to the resource methods discussed in subsection 5.3.1.

Again, the mining algorithm class was made an abstract class for many of the same reasons as the Resource class. It must have certain methods and interfaces to allow the standard Data Analysis Agent to communicate with it. It also must allow for the data mining algorithm and any optimizations to be included as well. Because the algorithms themselves can be coded a variety of ways, abstract methods that would force a particular processing of the data were not included. The abstract methods included are shown below:

```
public abstract ResultsList doMine(RequestClass req, Vector rules);
```

The abstract doMine class allows the Data Analysis Agent to begin mining in response to a request from the Task Agent. It expects a list of results in the form of ResultsList to be returned so that it may return them to the Task Agent, indicating completion.

5.4 Actual Implementation

In order to better show how the system can be extended, this section details how a new data source was actually added and the code that was required to make the addition. There are three key portions of the extension that must be accomplished. They are the new data source, the data mining algorithm, and the resource interface. Each of these is covered below and any extensions required are shown and discussed.

5.4.1 New Data Source The new data source will contain transaction-oriented data relating to…. It is a flat-file type database that includes X variables.

5.4.2 Creating a Resource Interface The creation of the resource interface is performed first. As mentioned before, the resource interface must be a subclass of the abstract Resource class. As such, it must contain implementations of the abstract methods in the Resource class. It also must contain the methods required to interface with the new data source type. All the required methods deal with the data format specific access of the new source. It must first be determined how access will be done. There are several options – existing API's, ODBC calls, proprietary interface, or other means. Because the data source we are adding is a proprietary format, we will code a new proprietary interface. This will involve using native Java code to read the flat-file. If an API or existing interface shell existed, this could be utilized by coding the queries to fulfill the request of each abstract method. Java does provide support for ODBC and JDBC calls.

The first issue is how to read the domain or extract it from the data source. In this case, the domain is included as the header of the file. Thus for the required method *readDomain*, we simply read the first line, break the line into tokens, or random variables, then assign each random variable to an element of the domain array.

This is accomplished by first attempting to open the data source for reading and, if successful, begin reading. If not successful, it catches the error and the domain remains null. The method uses *StringTokenizer,* a Java call that automatically breaks a string into as many tokens as exist, to individually add each domain variable to the domain list.

The next method is the *nextTransaction* abstract method. This method must pass back a transaction, presumably in some order, but not necessarily. The only requirement is that it will

read all transactions once and only once unless the data source is reset. In a flat-file such as the one we are using, this is done by simply keeping the file open, reading a line, breaking the line into tokens and passing them back to the calling method in the form of an array. The only small optimization here is to actually pre-read a block of transactions and maintain the list in memory. The *nextTransaction* code reads a block whenever there are less than 10 records in memory. The *readTransactionBlock* supports the *nextTransaction* in fulfilling the abstract method requirement. This also reflects the ability to add new methods for particular data source formats.

This *readTransationBlock* operates in much the same fashion as the *readDomain* method does, it reads a line from the database and parses it into a transaction. It is important to note the absence of a close() statement. The file is left open to ensure that after a line is read, the file record pointer points to the next line. If the file were closed and re-opened, a read line command would read the first line of the data source. This does not fulfill the requirement to read all transactions once and only once.

The *resetDB* method implementation is the easiest to fulfill. This is called to set the data source pointer back to the top. No matter where the *nextTransaction* pointer is, if *resetDB* is called, it should go back to the first transaction. In this case, this is accomplished by simply closing the data source and then reopening it.

The final 'mandatory' method is the *moreTransactions* method. This method simply returns a True or False to reflect if there are any more transactions in the data source (based on the current transaction pointer). This is made easier by the fact the resource class is 'looking ahead' at transactions. Thus if it attempts to read a block of 100 and can only read 50, it knows that after there are no more transactions in the database after the 50^{th}. If this occurs, the *readTransactionBlock* method sets the *noMoreTrans* variable to true, indicating the end of the transactions. The moreTransactions method only needs to see what element the *nextTransaction*

is currently at, along with the *noMoreTrans* variable. This ensures the data mining algorithm knows when the data source transactions are exhausted.

This is all the required abstract classes that must be implemented in order to be a subclass of the resource class. It is fairly obvious that one must have some knowledge of the format of the data source and how the records are laid out within the data source.

5.4.3 Data Mining Algorithm The last component is the data mining algorithm that will operate over the data source. In general, any data mining algorithm can be coded to operate across any format data source. Data mining and data mining algorithms are discussed in more depth in Chapter 2. There are algorithms that do operate more efficiently over particular data formats, i.e. relational. It is in the encoding of an algorithm that the dependencies to a data format are generally introduced.

In this case there are two factors we must consider. First is that the domain contains random variables and ranges within those variables. Second, the transactions are of the form: Clear,Green,None,Present,Rapid,Normal,Moderate,High,False,True,True

Where each value is the value of a random variable (RV) in the domain. In each transaction, all random variables are specified. The domain is specified as the first line of the file and is shown below:

Clarity,categorical;Color,categorical;Odor,categorical;Algae,categorical;Breathi
ng,categorical;Appetite,categorical;Stress Level,categorical;Ammonia 3-Level,cat
egorical;Bloody Fins,categorical;Ammonia Burns,categorical;Blackened Fins,categorical;

Each random variable is specified, then the type of the variable is shown. It is not as important to know the possible values for each RV but would be useful for the Ontology Agent to be able to check for this information. However, it is not readily obtainable.

This is the data format that PESKI can currently mine with DBMiner. The data mining algorithm to operate over this will be similar to the generalized data mining algorithm discussed in Chapter 2. There is one required method that must be implemented when extending the abstract MiningAlgorithm class. The *doMine* method must start the data mining algorithm operating over the data source. Because this will depend on how we encode the algorithm, we will wait until last to create this method.

6 Conclusions

6.1 Overview

There are two expected products from this research. First is this document detailing the thought process and design decisions that were made in designing and making an extensible multi-agent system that allows data mining and unification of association rules. It describes the methodology used to design the system in Chapter 3, applies that methodology to the problem in Chapter 4, and shows how to extend the system to include other data sources in Chapter 5.

The second product is the Java source code that implements the framework and provides reusable methods for extending the system. It includes all the agents implementation code and conversation classes for directed communications as well. There is future work that can be accomplished and is detailed in subsection 6.2

6.2 Future Work

Like most agent systems, there is the possibility for future expansion. There are several key areas that could be expanded or optimized. The first is the interface to PESKI. The interface was not implemented in the course of this work and needs to be coded. The next area is the Ontology Agent. Currently there is no semantic meaning applied to the domain variables the Ontology Agent maintains. If it is queried it performs a simple search of the domain for an agent to see if it matches. This does not take into account the various uses of any word or domain variable. For instance, smoke may be used in reference to smoking in a medical database, but also in a database about forest fires. In is unlikely someone looking for trend data is interested in data from both sources.

Another possible area of work also involves the control and selection of data sources. The introduction of a monitor agent would be very useful. Such an agent can observe the actions

and results of mining requests, as well as whether or not the user incorporates results into a BKB. By assessing this data, a monitor agent can return valuable information about each data source and the data mining algorithm that operates over it. Algorithms that consistently take excessive time and return unused results can be identified and changed or eliminated. If a new data source is added, the monitor agent may be able to provide recommendations as to what mining algorithm may be effective for the source based on its format.

Another extension can be the introduction of a 'trusted data source' identifier. Each user in PESKI has a profile that includes information about the users habits. Evaluating the patterns of data sources selected or by user specification, a data source can be more trusted than others and its results may carry more weight in unification.

One final area is the Unification Agent. The Unification Agent currently uses a naïve algorithm of assigning weights based on total number of records. This algorithm can be evaluated and changed based on inputs from the user inclusion of results. A monitor agent, or trusted sources. Unification is a critical area and other, possibly more valid or useful methods should be investigated.

Bibliography

[AGRA96] Agrawal, Rakesh, et. al. “Fast Discovery of Association Rules” *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press. Chapter 12, 307-328.

[ACHK93] Arens, Yigal, Chin Y. Chee, Chun-Nan Hsu, and Craig A. Knoblock *Retrieving and Integrating Data from Multiple Information Sources.* International Journal of Intelligent and Cooperative Information Systems. Vol. 2, No. 2. Pp. 127-158, 1993.

[AS94] Agrawal, Rakesh and Ramakrishnan Srikant “Fast Algorithms for Mining Association Rules.” September 1994 *Proceedings of the 20th Very Large Data Bases Conference*, Santiago, Chile 487 – 499.

[BAY96] Bayardo, R. et. al. “Semantic Integration of Information in Open and Dynamic Environments” MCC Technical Report, MCC-INSL-088-96, 1996.

[BRAD97] Bradshaw, Jeffrey “Software Agents” J. Bradshaw ed., AAAI/MIT Press, Menlo Park, CA 1997.

[CHAU97] Chauhan, Deepika “JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation”, ECECS Department, University of Cincinnati, 1997

[FU95] Usama M. Fayyad, Ramasamy Uthurusamy (Eds.): August 20-21, 1995 Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD-95), Montreal, Canada, AAAI Press.

[JAF97] JAFMAS, Comparison of JAFMAS to other Java-Based Agent tools, http://www.ececs.uc.edu/~abaker/JAFMAS/tab2.html, University of Cincinnati, 1997.

[JAT97] JATLite, JATLite Overview, http://java.stanford.edu/java_agent/html/, Stanford University, 1997.

[MHIP95] H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and Jennifer Widom. "Integrating and Accessing Heterogeneous Information Sources in TSIMMIS". In *Proceedings of the AAAI Symposium on Information Gathering*, pp. 61-64, Stanford, California, March 1995. .

[HNFD98] J. J. Han, R. T. Ng, Y. Fu, and S. Dao, `` Dealing with Semantic Heterogeneity by Generalization-Based Data Mining Techniques", M. P. Papazoglou and G. Schlageter (eds.), *Cooperative Information Systems: Current Trends & Directions* , Academic Press, 1998, pp. 207-231.

[KA97] Knoblock, Craig A. and José Luis Ambite “Agents for Information Gathering” J. Bradshaw ed., AAAI/MIT Press, Menlo Park, CA, 1997.

[KAH94] Knoblock, Craig A., Yigal Arens, and Chun-Nan Hsu "Cooperating Agents for Information Retrieval" *Proceedings of the Second International Conference on Cooperative Information System*s, Toronto, Ontario, Canada, University of Toronto Press, 1994

[NWA96] Nwana, Hyacinth S. "Software Agents: An Overview" *Knowledge Engineering Review*, Vol. 11, No 3, pp.1-40, Sept 1996.

[SHOH97] Shoham, Y. "An Overview of Agent-oriented Programming" J.M Bradshaw (ed), *Software Agents*, AAAI Press, 1997.

[SING98] SING98, Narinder "Unifying Heterogeneous Information Models" May 1998 *Communications of the ACM*, Vol. 41, No. 5. 37-44.

[SRIK97] Srikant, Ramakrishnan, Quoc Vu, and Rakesh Agrawal "Mining Association Rules with Item Constraints" August 1997 *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, Newport Beach, California. 67 – 73.

[SA96] Srikant, Ramakrishnan, and Rakesh Agrawal "Mining Quantitative Association Rules in Large Relational Tables" June 1996 *Proceedings of the ACM SIGMOD Conference on Management of Data*, Montreal, Canada.

[SA95] Srikant, Ramakrishnan, and Rakesh Agrawal " Mining Generalized Association Rules" 1995 *Proceedings of the 21st Very Large Database Conference*, Zurich, Switzerland. 407-419.

[STEIN96] Stein, Daniel J. III. " Utilizing Data and Knowledge Mining For Probabilistic Knowledge Bases" MS Thesis AFIT/GCS/ENG/96D-25. Graduate School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, December, 1996.

[WT95] Woelk, Darrell, and Christine Tomlinson. "InfoSleuth: Networked Exploitation of Information using Semantic Agents" COMPCON, March, 1995.

[WJ95] Wooldridge, M. and Jennings, N. "Intelligent Agents: Theory and Practice", *Knowledge Engineering Review* Volume 10 No 2, June 1995, Cambridge University Press.