

**Carnegie Mellon**  
**Information Networking Institute**

**Agent-Based Resolution of Remote Network Access Issues  
Using Semantic Domain Knowledge**

**TR 2004-19**

**A Project Submitted to the  
Information Networking Institute  
in Partial Fulfillment of the Requirements  
for the Degree**

**MASTER OF SCIENCE  
in  
INFORMATION NETWORKING**

**by  
John Austin Talanda Fath**

**Project Advisor: Dr. Katia Sycara, RI  
Project Reader: Dr. Joseph Giampapa, RI**

**Pittsburgh, Pennsylvania  
April 23, 2004**

**Carnegie Mellon University  
Information Networking Institute**

**AGENT BASED RESOLUTION OF REMOTE  
NETWORK ACCESS ISSUES USING  
SEMANTIC DOMAIN KNOWLEDGE**

**A Thesis Submitted to the  
Information Networking Institute  
in Partial Fulfillment of the Requirements  
for the Degree**

**MASTER OF SCIENCE  
in  
INFORMATION NETWORKING**

**by  
John Austin Talanda Fath**

**Advisor: Katia Sycara  
Reader: Joseph Giampapa**

**Pittsburgh, Pennsylvania**

## **Acknowledgements**

This work was completed under a grant from the Office of Naval Research, Interoperability of Future Information Systems through Context-and Model-based Adaptation (#N00014-02-1-0499). The views and conclusions in this thesis are those of the author and should not be interpreted as presenting the official policies or position, either expressed or implied, of the Office of Naval Research or the U.S. Government unless so designated by other authorized documents.

I would also like to extend special thanks to the people at the Intelligent Software Agents Lab for their technical guidance in this project and to my INI classmates who have helped make this thesis possible.

### **Katia Sycara**

- Thesis Advisor and IFIS principle investigator

### **Joe Giampapa**

- Thesis Reader and research and project manager

### **Dan Siewioreck**

- IFIS principle investigator

### **Aaron Steinfeld**

- Designed the User Interface, helped with the previous technical report and taxonomies

### **Naveen Srinivasan, Massimo Paolucci**

- Candid advice on Web Service, DAML-S, OWL, and Inference Engines

### **Rahul Singh**

- Programmed the Service Agents and helped Retsina communicator

### **Ratika Sanghi**

- Background research, Previous Technical Report

### **Sean Owens**

- Introduction to the Retsina Java API and advice on software agents

### **Robert Cosgrove, Mark Puskar**

- For their time and access to the SCS Computing Facilities

<b>1</b>	<b>INTRODUCTION .....</b>	<b>1</b>
<b>2</b>	<b>BACKGROUND.....</b>	<b>3</b>
2.1	IDENTIFYING THE PROBLEM.....	3
2.2	PROBLEM RESOLUTION MODEL.....	4
2.3	SOFTWARE AGENTS .....	6
2.4	REFLECTIVE PROGRAMMING IN MAUDE .....	9
2.4.1	<i>Functional Modules .....</i>	<i>10</i>
2.4.2	<i>System Modules.....</i>	<i>11</i>
2.4.3	<i>Full Maude.....</i>	<i>12</i>
<b>3</b>	<b>EMERGING TECHNOLOGIES .....</b>	<b>14</b>
3.1	SERVICE ORIENTED WORLD .....	15
3.1.1	<i>Web Services .....</i>	<i>15</i>
3.1.2	<i>Service Oriented Architectures .....</i>	<i>16</i>
3.2	KNOWLEDGE REPRESENTATION .....	17
3.2.1	<i>Resource Description Framework .....</i>	<i>18</i>
3.2.2	<i>Web Ontology Language .....</i>	<i>19</i>
3.2.3	<i>Inference.....</i>	<i>20</i>
<b>4</b>	<b>THISTLE MULTI-AGENT SYSTEM .....</b>	<b>21</b>
4.1	OVERVIEW .....	22
4.2	KNOWLEDGE MODELS .....	22
4.3	SECURITY MODEL .....	22
4.4	APPLICATION PROFILES .....	25
4.5	SECURITY POLICIES.....	27
4.6	THISTLE ARCHITECTURE.....	29
4.7	APPLICATION AGENTS .....	30
4.8	SERVICE AGENTS .....	31
4.9	POLICY AGENT .....	32
4.10	TASK AGENT .....	32
4.10.1	<i>Policy Verification in Maude .....</i>	<i>33</i>
4.11	THISTLE USER INTERFACE.....	35
4.11.1	<i>Email Scenario.....</i>	<i>35</i>
<b>5</b>	<b>DISCUSSION.....</b>	<b>42</b>
5.1	ACHIEVEMENTS.....	42
5.2	DIFFICULTIES.....	43
5.2.1	<i>Emergent Technologies and Architectural Changes.....</i>	<i>43</i>
5.2.2	<i>Developing a Reflective Logical Formalism .....</i>	<i>44</i>
5.3	FUTURE WORK .....	45
5.3.1	<i>More, Smarter Agents.....</i>	<i>45</i>
5.3.2	<i>Improved Architecture .....</i>	<i>46</i>
5.3.3	<i>Knowledge Representation.....</i>	<i>46</i>
5.3.4	<i>Extras .....</i>	<i>47</i>
<b>6</b>	<b>CONCLUSION .....</b>	<b>48</b>
<b>7</b>	<b>REFERENCES .....</b>	<b>49</b>

# List of Illustrations

## Tables

Table 1: OWL Lite key word Summary .....	20
Table 2: OWL Axioms in Descriptive Logic.....	21
Table 3: Simplified Security Parameters .....	23
Table 4: Service Security Rights and Encryption Mappings.....	26

## Figures

Figure 1: Interoperability Problem Resolution Model .....	4
Figure 2: Retsina Multiagent System and Individual Agent infrastructures. ....	7
Figure 3: Retsina Individual Agent Architecture .....	8
Figure 4: Basic Retsina Agent Types and Interactions .....	9
Figure 5: Denker's original Security Ontology. ....	24
Figure 6: Additions to the security ontology mapped to OWL-S ServiceParameter. ....	25
Figure 7: THISTLE Architecture .....	29
Figure 8: User Interface after initial network discovery .....	37
Figure 9: Mail Application started .....	37
Figure 10: User Interface with E-Mail Policy Violation .....	38
Figure 11: User Interface with Options for E-Mail Policy Violation .....	38
Figure 12: User Interface after VPN is started.....	39
Figure 13: VPN Agent waits for the user to connect the VPN .....	39
Figure 14: User Interface with VPN service composed .....	40
Figure 15: VPN Client has connected. ....	40
Figure 16: VPN Application blocked, new rights needed.....	41

## Code Samples

Code Sample 1: XML and RDF Examples.....	19
Code Sample 2: Application Profile in OWL .....	27
Code Sample 3: E-Mail Policy .....	28
Code Sample 4: RemoteAccess Full Maude Classes .....	33
Code Sample 5: Policy Verification in Full Maude .....	34
Code Sample 6: Find Options conditional rewrite rule .....	34

## **Abstract**

Managing remote network connections can be very intimidating for computer users who lack a technical background, especially when security is a concern. Users are confronted with a variety of applications and service configurations that change with domain, location, and with changing security policy demands. Different configurations lead to different levels of security and may or may not satisfy security policies even if the applications still work. Network administrators usually end up configuring computers manually or posting instructions and FAQs online to solve common problems. These documents are useful for expert users, but still require time to read and implement. However for novice users, these documents are often too technical or they are ignored because of the time spent trying to find them. The information is freely available but it is not being used. Remote services still fail leading to frustrated users and lost time for the administrators who then have to diagnose user settings and solve help requests for common problems. Ignoring security policies may also lead to damaging breaches to security as unencrypted traffic is passed through public networks.

In an effort to solve these remote network issues and improve the current state of service interoperability, this thesis presents a novel multi-agent system that uses domain knowledge to manage user applications. By capturing the security policies, application profiles, and connection processes as computer readable domain knowledge, the agent based system can present the current security states of the system to the user with easily understood icons, discover policy violations before they cause errors, and suggest solutions. While the software agents are still not able to replace the problem solving capabilities of the system administrator, the multi-agent system is able to enhance the user experience by understanding and quickly disseminating knowledge.

# 1 Introduction

The purpose of an information system is to provide people with the knowledge they desire while relieving them from the burden of collecting and extracting that knowledge from the available data. Current information systems like the Internet, company intranets, and the databases or file systems that store their content are very good at sharing information, but they often require too much technical or domain knowledge to be practical for ordinary users. Help documents, FAQs, APIs, and manuals may be readily available, but either the time required to search and read through the appropriate documents is prohibitive or a person may not have the technical expertise or domain knowledge to use the documents. The information is freely available but is not being used because it is too difficult to find in the system.

In the domain of remote network access, this lack of timely information exchange often frustrates network users and burdens network administrators with redundant help requests. The types of remote access applications and their configurations change from network to network and there are usually many different ways to connect between the local and remote networks. People who wish to access the networks remotely must understand these different modes of connection as well as all of the security and quality of services implications that accompany the network path they choose to use. Even correctly configured applications may then have to be adapted to network outages or undocumented changes. Information about a specific network policy, modem model, or application is usually available online, but choosing the best application and diagnosing any problems that occur requires knowledge of the complete network connection. People often do not have the time or technical understanding of the network to read all the necessary documents, so they either don't use services or they use the default settings until an error occurs and then send help requests to the network administrators. The network administrators then have to take time to respond to each help request. This often requires multiple message exchanges as the administrators try to understand the current state of the remote computer and network so they can offer suggestions for interacting

with the SCS network. Even though the problems are often very similar, it still takes time to understand what the situation is for each help request and then to help the person correctly configure their application. This is time the user spends without a working service and time the administrator has to use to solve a problem that should already be solved when the network information is posted.

Clearly the problem with the remote network access domain is that although information is available, the appropriate knowledge is not effectively available when and where it is needed. When users attempt to access the network remotely from different locations they often do not know the current state of the local network, their applications, or the possible paths to the remote network. When a problem occurs and technical help is called upon to diagnose a problem, that person often does not know the local state of the computer, application, network or user so time is lost negotiating a common vocabulary to understand these states.

The THISTLE multi-agent system solves these problems by using intelligent software agents with a formal knowledge representation to verify remote access connections. Security, Application, and Policy ontologies written in the Web Ontology Language (OWL) are used to define the concepts and relationships in the remote network access domain. Using these semantics, information agents embedded in network applications can correctly represent the state of the connections in a consistent and well defined language, thus eliminating the normal ambiguities in help requests. With the aide of a formal policy verifier written in Maude, an intelligent task agent then uses this state and the network policies to automatically diagnose network problems and offer possible options to the user. Remote access users are also presented with a graphical view of their current connections and properties.

By imbuing the software agents with the appropriate domain knowledge and giving them control of the network applications, THISTLE is able to effectively transform solitary user driven applications into a single virtual network of semantic services. The task agent is then able to view this network in terms of the domain ontology and intelligently



compose connections that satisfy high level properties about the entire connection instead of the individual applications. These intelligent services alleviate the user from manually monitoring and diagnosing the underlying remote applications. The automated nature of the agents also allows network administrators to effectively update users on new policies or network services by publishing policies that all THISTLE systems can read and enforce without any added effort on the users part, thus effectively transferring the appropriate knowledge from the administrators to the remote users.

The rest of this thesis will discuss the previous work that lead to the development of the THISTLE prototype, review the emerging technologies that made it possible, detail the THISTLE Architecture, and describe future research for the project.

## **2 Background**

### **2.1 Identifying the Problem**

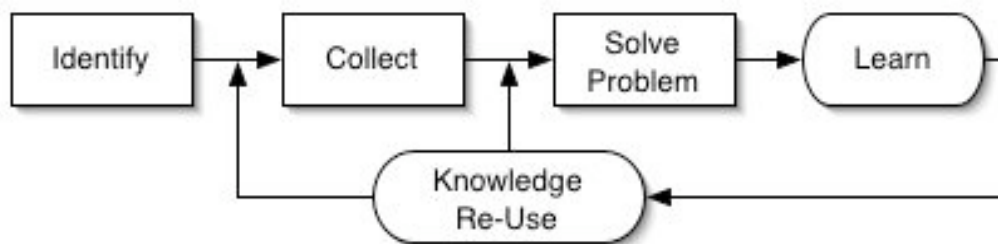
While there are continuous efforts to improve the effectiveness of university help desks[1], most have taken network errors for granted and focused on the management of 24/7 help desks for thousands of users [2]. Some of the larger public universities like the University of Pittsburgh have turned to knowledge management solutions [3] in an attempt to improve knowledge reuse. Corporations such as Serviceware [4], Primus [5], and Parature [6] all integrate common help desk databases and provide sophisticated self-help web portals for users. Decision trees, collaboration tools, document classification, and even a Cognitive Processor are used to help manage knowledge created by help desk administrators, but all of these tools only manage knowledge for distribution between human users. Even the best of portals take time to search and require users to set up their own applications. Nothing is done to facilitate the automatic prevention, identification, or resolution of networking errors.

To discover exactly what could be done to help automate help desks from the client perspective, in 2003, Sanghi and Steinfeld [7] conducted an extensive survey of remote access trouble ticket data from the School of Computer Science (SCS) Computing Facilities to determine the specific problems in this domain. Two and a half years of

tickets were codified and then categorized for statistical comparison. The study found that the majority (47%) of requests were simple phone number queries for modem users and single configuration changes due to shifting network policies. These are problems where users either didn't understand or were not aware of configuration values. Security problems were also frequent, especially with regards to VPN usage, as over half of end user problems were related to obtaining necessary user rights. Problems with third party networks had the highest mean time until resolution. Administrators and non-technical users often have very different mental models of remote access so extra communications were required to determine exactly what people were asking and how their computers were configured. It was also noted that administrators made very little use of the "Root Cause" and "Solutions" fields in the database.

## 2.2 Problem Resolution Model

Based on these observations, it was determined that the majority of user errors could be solved locally with the aide of an automated problem resolution system.



**Figure 1: Interoperability Problem Resolution Model**

*Identify Problems:* Remote access users needed a better way to understand the state of their remote connections and identify when errors have occurred. Programs usually provide logs or alert messages for errors, but many errors go undetected. A person may not want information they send to their corporate network to be transmitted in plain text across public domains, but sending unencrypted messages rarely causes errors. Also the system needs to discover configuration or connection errors before they cause applications to violate policies.

*Collect State:* Once problems have been identified, it is important to collect the current state of the system in a formal language. If users don't understand the terminology used in error messages then they can't effectively communicate what their specific problems are. Even users who do understand the domain usually do not want to take the time to search separate logs and run third party diagnostics to determine the current state of their connections, they just want their applications to run. The system needs a precise model of remote connections that allows applications, users, and administrators to share a common mental model of the system and eliminate any of the ambiguities that administrators usually have to deal with when diagnosing a problem.

*Solve Problem:* With all the information collected, solving known problems then comes down to effectively modeling all the requirements and effects of the applications used in remote connections and matching them to the relevant policies. Problems result from errors in required components or violations of network policies. Problems can be corrected by finding new connection options with the same services that meet the requirements of the policies. When there are multiple solutions, users can be informed of the consequences of those choices and then use their own personal preferences to decide which connections to use.

*Learn:* As problems are identified, diagnosed, and solved the system is creating new knowledge. If all the steps that lead up to a correct solution are saved in a format that is flexible enough to compare to future errors, then by saving this state the system can learn. Learning is inherently dependent on the generality and accuracy of the knowledge used to model the states of the system.

*Re-Use Knowledge:* As was noted in the case study, network administrators currently do not reuse knowledge. The "Root Cause" and "Solutions" are communicated to other administrators as needed, but there is no efficient way to inform non-technical users of ways to resolve common problems. To bridge this gap, once solutions are found, they need to be learned by the system and automatically distributed to other users.

## 2.3 Software Agents

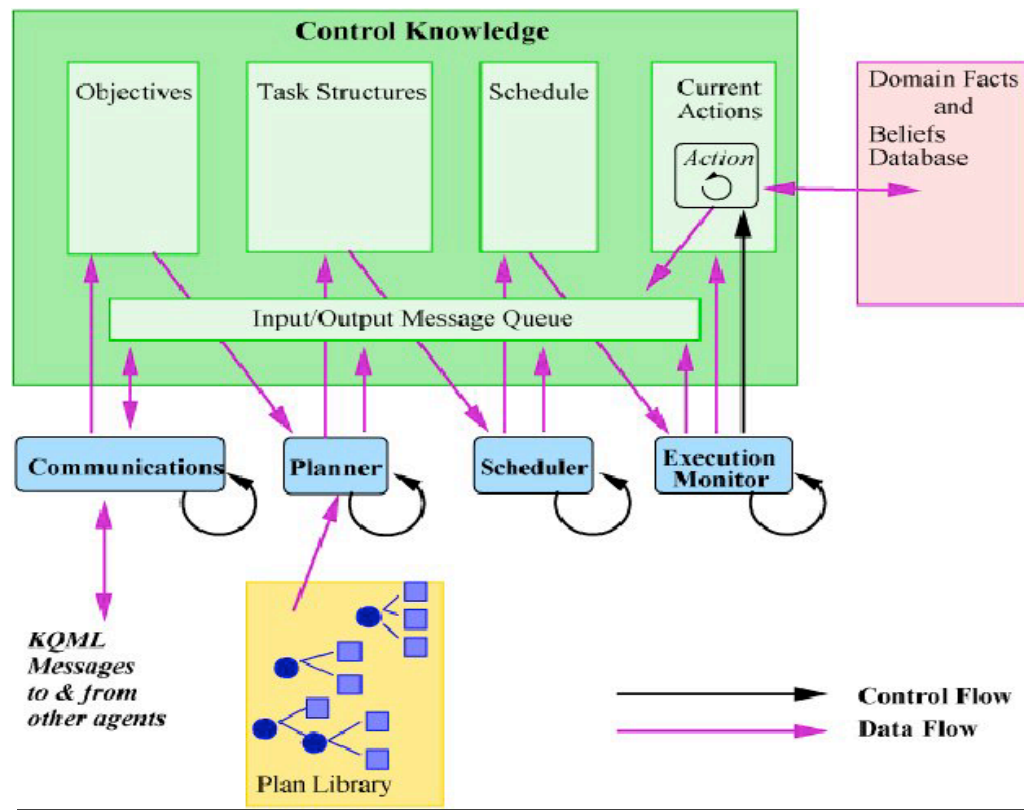
While network administrators and technical end users can solve remote network access problems easily, the problem resolution model outlined above is ideally suited for software agents. Agents can react to events at the speed of processors and can perform tedious policy verification tasks without complaints. As previous work at CMU [8,9] and elsewhere [10] has shown, systems of software agents are very effective at collecting domain information and aiding humans in making decisions. These automated systems could learn to eliminate errors at the source and thus avoid the entire help desk scenario by delivering domain knowledge directly to the application. Human intervention would only be necessary to solve initial problems or resolve hardware failures.

But what exactly is a software agent? The distinction between a control system and a software agent is sometimes blurred, but it is generally accepted that a software agent is “an autonomous, (preferably) intelligent, collaborative, adaptive computational entity”. Very sophisticated programs do often seem intelligent, but in fact have very rigidly defined controls. For instance a spell checker may seem to understand language, but it can only compare tokenized strings to a dictionary. Agents are situated in a software environment and not only react to it, but have the ability to change it based on their goals [12]. Agents communicate with other agents to share knowledge and collaborate on tasks [11] which require diverse specialties, something applications never do. They are also usually characterized by their ability to make choices based on their perceptions, which requires the ability to infer and execute needed actions in pursuit of their goals.

MAS INFRASTRUCTURE	INDIVIDUAL AGENT INFRASTRUCTURE
<b>MAS INTEROPERATION</b> Translation Services    Interoperation Services	<b>INTEROPERATION</b> Interoperation Modules
<b>CAPABILITY TO AGENT MAPPING</b> Middle Agents	<b>CAPABILITY TO AGENT MAPPING</b> Middle Agents Components
<b>NAME TO LOCATION MAPPING</b> ANS	<b>NAME TO LOCATION MAPPING</b> ANS Component
<b>SECURITY</b> Certificate Authority    Cryptographic Services	<b>SECURITY</b> Security Module    private/public Keys
<b>PERFORMANCE SERVICES</b> MAS Monitoring    Reputation Services	<b>PERFORMANCE SERVICES</b> Performance Services Modules
<b>MULTIAGENT MANAGEMENT SERVICES</b> Logging,    Activity Visualization, Launching	<b>MANAGEMENT SERVICES</b> Logging and Visualization Components
<b>ACL INFRASTRUCTURE</b> Public Ontology    Protocols Servers	<b>ACL INFRASTRUCTURE</b> ACL Parser    Private Ontology    Protocol Engine
<b>COMMUNICATION INFRASTRUCTURE</b> Discovery    Message Transfer	<b>COMMUNICATION MODULES</b> Discovery Component    Message Transfer Module
<b>OPERATING ENVIRONMENT</b> Machines, OS, Network    Multicast    Transport Layer: TCP/IP, Wireless, Infrared, SSL	

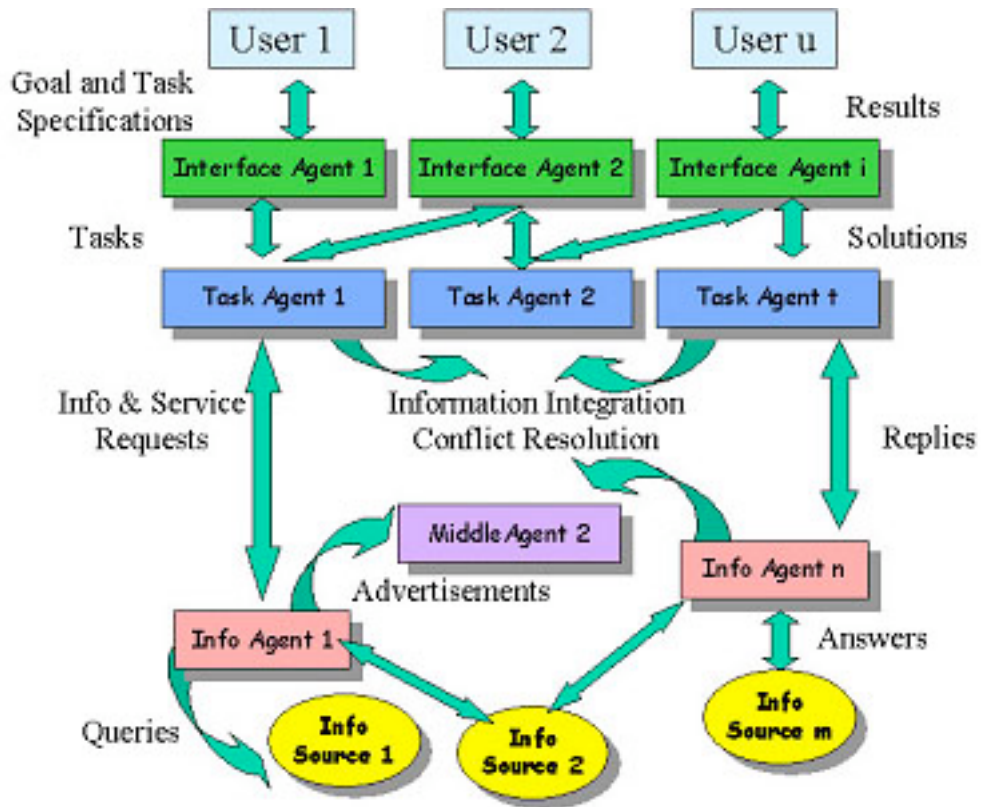
**Figure 2: Retsina Multiagent System and Individual Agent infrastructures.**

Building on many years of experience with multiagent systems (MAS), the Retsina [14] architecture was developed at CMU as flexible infrastructure to provide the components common to most agent systems. The MAS infrastructure provides the communication, security, translation, and discovery mechanisms which agents need to collaborate effectively. The management services also provide convenient resources for testing and deploying MAS. The individual agents utilize four generic modules for interfacing with the MAS infrastructure and carrying out their own tasks. Each agent has a communication module, a task planner, scheduler, and an execution monitor. The types of tasks, objectives, and beliefs an agent may have give it its unique domain specific abilities, but collaboration, planning, and execution are common to all agents.



**Figure 3: Retsina Individual Agent Architecture**

Agents interacting in MAS can generally be categorized into four different types – information, task, middle, and interface agents. Information agents provide intelligent access to information in the system. They are responsible for creating a specific instance of knowledge other agents need to use but either can't or don't want to find. Task Agents aid users in a specific service which requires problem solving, adaptation, and an integration of the domain knowledge found by the information agents. Task agents need to collaborate with other agents to illicit information or aid with their plans. Middle agents act as matchmakers and use their knowledge to redirect requests from task and information agents requesting services to those offering services.



**Figure 4: Basic Retsina Agent Types and Interactions**

There have been many studies of autonomous agent architectures, agent description languages, and multi-agent systems. There have also been many attempts to use agent based systems in controls, robotics, e-Commerce, and scientific studies and there have even been some commercial implementations. However, as a technology, software agents are still evolving. Many agent implementations are focused on simple reactive behaviors or rule based expert systems with limited ability to reason over a domain or adapt to changes outside of the rule base. The current THISTLE research efforts hope to improve these efforts by developing a practical application that incorporates new trends in knowledge representation and formal methods.

## 2.4 Reflective Programming in Maude

Maude[15] is a fully reflective programming language and development environment that utilizes rewriting logic and its equational logic sublanguage to specify formal executable environments [16, 17, 18]. The reflective nature of rewriting logic allows Maude

programs to specify not only the data structures and algorithms common to C or Java, but to create the actual algebra that defines the programmer’s specific application. Programs are able to define their own syntax, operations, and data types to model the behavior of concurrent systems. While a full theoretical explanation of rewriting logic and the implementation of Maude are beyond the scope of this thesis, the rest of this section will give an overview of the Maude concepts needed to understand policy verification in THISTLE.

#### 2.4.1 *Functional Modules*

Maude is a declarative language based on a sound logical system. Indeed it was designed as a metalanguage to define formal systems. Every program is built from logic theories that are expressed as programming modules. Computation is equivalent to logical deduction based on the rules defined by the logic in the programs. Functional modules form the foundation of that logic. They create the data types and operations used in the equational theories. Data types are specified in terms of sorts and subsorts. The keyword `sort` is used to define any type in the system and `subsorts` are used to define more specific types within a sort. The following example creates `Positive` and `Negative` sorts as specific subsorts of the `Rational` sort.

```
subsorts Positive Negative < Integer .
```

Operations define the syntax used to create sorts. Maude allows both prefix and mixfix operators to be defined using the `op` or `ops` commands. An operation consists of the `op` keyword followed by the operator symbols, a colon, then a list of sorts for the arguments, a right arrow, the sorts for the results, and then any operator attributes. Underscores are used to specify where mixfix arguments are placed.

```
op OpName : Sort0 ... Sortk -> Sort [OperatorAttributes] .
--- Examples
op + : Integer Integer -> Integer .
op _+_ : Integer Integer -> Integer .
```

The possible operator attributes include associative, commutative, identity, precedence levels, and constructor. Constructors are operations that take no arguments but have



syntax and produce a sort. Identity operators do not affect the sort if included (like adding 0 or a null set). Operators can also be overloaded using different sort types.

To give these operational definitions meaning, equations are used. Equations define the rules for determining equivalence and serve to simplify the operation. When operators are used in expressions, Maude will evaluate the equations defined for each operator to determine what expressions evaluate to. Equations must be confluent and deterministic (Church-Rosser [18]) so they can always be reduced to a single sort. Variables are simple placeholders for sort types used to help define equations. The syntax and examples for both are given in the module below [17].

```
eq Term-1 = Term-2 [StatementAttributes] .
var name : sort .

--- Example functional module

fmod CARD-DECK is
  sorts Number Suit Card .
  ops A 2 3 4 5 6 7 8 9 10 J Q K : -> Number [ctor] .
  ops Clubs Diamonds Hearts Spades : -> Suit [ctor] .
  op _of_ : Number Suit -> Card [ctor] .
  op CardNum : Card -> Number .
  op CardSuit : Card -> Suit .
  var N : Number . var S : Suit .
  eq CardNum( N of S ) = N .
  eq CardSuit( N of S ) = S .
endfm
```

Equations can be further refined using conditions. Module defined Boolean operations, membership axioms, patterns, and equations using the built-in `Bool` module can all be used to determine when an equation applies.

#### 2.4.2 System Modules

The system modules build on this equational logic by adding rewrite rules which transition a system from one state to another. This forms a full 4-tuple rewrite theory  $\mathfrak{R} = (\Sigma, E \cup A, \phi, R)$  [16] where  $\Sigma$  is the signature of the type definitions,  $E$  is the set of equations,  $A$  is the set of attributes,  $R$  are the rewrite rules, and  $\phi$  is the frozen set of arguments to  $\Sigma$ . While equations specified simplifications in the system, rewrite theories in general define one-way transitions between states. Function modules can be thought of

as the basic data types of a system, defining structures and their operations, and rewrite rules are all the *possible* methods to apply to those structures. However, unlike functional programming languages, rewrite rules already have a defined set of predicates that must exist before they can be called. Each rewrite rule consists of the rewrite operator `rl`, a label enclosed in square braces followed by a colon, a required term followed by an `=>`, and the resulting term with any statement attributes following in square braces. The required term is like a set of arguments that must be present, but in Full Maude, those arguments can include conditions on attributes as well.

```
rl [Label] : Term-1 => Term-2 [StatementAttributes] .
```

Both terms must be of the same kind [16]. Just as with equations, rewrite laws may have conditions that can be any Maude expression that evaluates to a `Bool` sort, membership axiom, or pattern to search. The conditional equations can even contain additional rewrite laws. These conditions are much like the formal preconditions defined in languages such as Z [19] that are used to define formal semantics for functional programs. The difference is that the conditions in Maude are part of its logical foundation and specifically defined in the module logic, not a layer of abstraction forced onto the functional code. The tradeoff is that programmers have to define a formal system and this can be far more challenging than using functional code to solve tasks. More concrete examples of rewrite laws will follow in the Full Maude description.

### 2.4.3 Full Maude

Full Maude is an object-oriented extension of the Core Maude modules written in Maude using an interactive loop mode. Full Maude defines a generic syntax for objects, classes, messages, and configurations to aid in the programming of event based systems. A class defines the structure of an object, just as in Java, and objects are specific instances of a class. A class can also be thought of as a high level sort, defining the possible object signatures. A class consists of a class identifier (`Cid`), which is a sort, and a list of attributes that are sorts, including class or object identifiers. Classes also support multiple inheritance with subclasses that inherit all the attributes of parent classes.

```
class C | attribute1: Sort1, ... , attributen: Sortn .
```

```

< Oid_Name : C | attribute1: variable1, ... , attributen:
variablen >
msg syntax : Oid Sort1 Sortn -> Msg .

```

Each object has an object identifier (Oid), a class identifier, and a list of the instances of its attributes. Objects serve as variables in rewrite rules but actual instances are also returned as Maude programs are run. Messages are assumed to be “sent” to objects to convey information, but are mostly left to the design of the programmer. They are analogous to operators but at the object level. Each message type has a name and a list of arguments that starts with a destination Oid and results in a Msg. The Msg and Configuration sorts are used as placeholders for groups of msg and Oid sorts. Configurations are used to express the sort needed for the results of rewrite laws and equations that have many objects and messages that are not directly part of the rule. It is like a general state of the overall system. The following is an example object module that specifies a 3x3 sliding puzzle game from the Maude Primer [17].

8	3	2
5		6
4	1	7

Mixed Up

1	2	3
4		5
6	7	8

Solved

```

(omod TILE-PUZZLE is
  sorts Value Coord .
  ops One Two Three Four Five Six Seven Eight : -> Value .
  op empty : -> Value [ctor] .
  ops 0 1 2 : -> Coord [ctor] .
  op s_ p_ : Coord -> Coord .
  op `(_`,`_`) : Coord Coord -> Oid [ctor] .
  eq s 0 = 1 .
  eq s 1 = 2 .
  eq p 1 = 0 .
  eq p 2 = 1 .
  class Tile | val : Value .
  msg move : Oid Oid -> Msg .
  vars R1 R2 C1 C2 : Coord . var V : Value .
  crl [l] : move((R1, C1), (R1, C2))
  < (R1, C1) | val : V > < (R1, C2) | val : empty >
    => < (R1, C2) | val : V > < (R1, C1) | val : empty >
  if C2 == p C1 .
  crl [r] : move((R1, C1), (R1, C2))
  < (R1, C1) | val : V > < (R1, C2) | val : empty >
    => < (R1, C2) | val : V > < (R1, C1) | val : empty >
  if C2 == s C1 .
  crl [u] : move((R1, C1), (R2, C1))

```

```

    < (R1, C1) | val : V > < (R2, C1) | val : empty >
    => < (R2, C1) | val : V > < (R1, C1) | val : empty >
    if R2 == p R1 .
    crl [d] : move((R1, C1), (R2, C1))
    < (R1, C1) | val : V > < (R2, C1) | val : empty >
    => < (R2, C1) | val : V > < (R1, C1) | val : empty >
    if R2 == s R1 .
endom)

```

Rewrite rules in Full Maude transition the system from a configuration of objects and messages to a new configuration of objects and messages. As messages are sent from object to object, new messages are fired and objects may be created, destroyed, or updated. As was mentioned before, rewrite rules define all the possible transitions for a concurrent system. Just as functional methods they do not in themselves have any specific ordering other than the preconditions specified in the rule and possible conditional arguments. Concurrent systems are tested by loading the initial object oriented modules into Maude and then running rewrite and search commands to change the state or to view all the possible states a system can reach with a bounded number of rewrites. Maude provides very fine grained control over the different types of searches and strategies that are employed when choosing which available rewrite rules to execute, which is necessary when verifying properties of the system, but simple modules can be run using the basic `rewrite` or `frewrite` commands with a number of rewrites to execute. Maude then returns the resulting configuration of objects and messages after the rewrite rules have been executed.

### 3 Emerging Technologies

This is an exciting time for research in software agents. The current research in mobile computing, pervasive information systems [21], the semantic web[20], and the ever increasing complexity and demand of large scale commercial distributed software systems [22] has created a great interest in technologies related to software agents. With millions of lines of code tied up in legacy projects and project lifecycles decreasing, companies are looking for better ways to develop and integrate their systems [23]. Although projects dealing with high-level artificial intelligence are still met with much deserved apprehension, the software industry is beginning to realize the benefits of adaptable, autonomous, and preferably intelligent software components with formal

properties. The current web services and service oriented architecture movement has grown out of this need for autonomy on the internet and has increased the attention of industry on defining standards for interoperability and information exchange. These standards are starting to merge with research efforts typically left to DARPA and universities. The following section will detail the emerging technologies that are changing the way services interoperate and show how research in software agents and web services are beginning to merge.

### **3.1 Service Oriented World**

Computers and the internet have become an integral part of the daily lives of most businesses and the growing digital generation [25]. As information becomes more accessible more people come to rely on it and the demand to improve accessibility increases. While originally the internet was designed to provide human readable content, the idea of clicking through different static web pages has become mundane and inefficient. Online businesses can't wait for orders to be processed manually or for calls to be sent to business partners. Internet users don't want to click through multiple pages to find information. The same can also be said for desktop applications and operating systems. As the amount of information sources and the complexity of networks increases, users are becoming inundated with the task of handling all their separate applications and migrating the information to PDA's, cell phones, and laptops. Applications that were suppose to eliminate paper and increase efficiency seem to only be producing more paper and less free time. This demand for better more integrated services has been at the heart of AI and software agent research since its inception, but now the internet and software industry are beginning to lay the framework for these composable services to become common place.

#### *3.1.1 Web Services*

Although many .com companies in the late 1990's promised online services that would change the way people use the internet, web service technologies are only now maturing into real usable standards. Web services leveraging the internet and XML [26] are set to

provide the basic independent components needed to create composable services. XML has become *the* industry standard for data interoperability. Unlike DCOM or CORBA, which use binary interchange formats, XML data is text based with a single standard all parsers adhere to. While this requires every service to translate data from binary to text and back, it also does make the data interoperable. Data from any service with a published schema can easily be parsed by another service. All the current web service standards build on XML to provide service discovery, data transparency, and even security. Data transparency is achieved using the Simple Object Access Protocol (SOAP) [28]. SOAP is a W3C standard for exchanging XML messages. The protocol defines a simple XML envelope with a header and a body. The header contains domain specific metadata about the message and the body contains the actual data being transmitted between SOAP nodes. To discover what XML should be in the SOAP messages, the Web Service Discovery Language (WSDL) [29] provides an XML language for describing the abstract functionality and concrete details of a service method. A WSDL description provides the exact data types and names of all the methods available in a web service and the sequence of message exchange. A WSDL can be created automatically from the interface definition of a web service, and the SOAP messages can then be generated from the WSDL. Universal Description, Discovery, and Integration (UDDI) [30] servers then allow web services to advertise their WSDL definitions. More complicated business processes may also be defined using WSCI[31] or BPEL4WS[32] which again combine the XML definitions of service interfaces to ensure interoperable calls between components.

### *3.1.2 Service Oriented Architectures*

Service Oriented Architectures are more than just systems using web services. Ideally an architecture is independent of the underlying components and instead abstracts the types of components and connection properties that are needed to design a successful system. A service oriented architecture is a recognition at the architectural level that services are composed of disparate components that must be able to interact seamlessly with each other. While objects inside the components may rely on specific interfaces with known data structures and methods, each component operates as a black box and only provides

abstract interfaces for other components to communicate with. Functionality in the system is a matter of composing services from the available components. The component connections must be designed to support both location and data transparency so they are not tied to a specific component or platform, but can interoperate with any new components that may be added later. Achieving this flexibility is difficult. Interoperability requires added layers of communication and abstraction and an infrastructure to make these changes transparent to developers. More computation is needed as information is translated between component and communication layers instead of direct native function calls. But as the transitions from assembly to functional to object oriented and even interpreted or platform independent languages have shown, programmers are more than willing to trade performance for reduced complexity and added functionality.

### **3.2 Knowledge Representation**

For services that are willing to make the performance tradeoff, XML serves as a very flexible interchange language. However, XML and the current web service standards only provide syntactic interoperability [34,35,36]. A SOAP message is known to contain XML data, and, based on a WSDL, a service can determine what XML data type an element contains, but XML and XML-Schema provide no way of determining what the data represents. A web service must know a priori that the message it receives contains a certain XML element from a previously arranged standard so it can be parsed and turned into the correct internal data structure for the application. The UDDI servers also can't offer any way to search based on the capabilities of a service because WSDL only defines data types. The best UDDI can do is check to make sure what types you wish to pass match up exactly with what a service offers. So after all that effort, these XML interfaces only really provide a standard web interface with data transparency and support for static compositions of services.

Much of the difficulty with designing more intelligent software agents is also due to the difficulty in expressing knowledge in syntax software algorithms can interpret. To this

end, there has been much ongoing research in the area of formal knowledge representation languages. These languages provide rules for expressing information in a structured manner that allows software to make implicit and mathematically verifiable inferences on the explicit knowledge of a given domain. Just as a person inherently knows that an employee is a person with certain properties, software agents need some way to understand how person data in XML is related to instances of employee data. This research builds upon the formal mathematical rules of First Order Logic (FOL), Model Theory (MT), and their derivatives to create languages that have provably searchable expressions for knowledge representations. While many of these languages have been proposed, the computational requirements of reasoning over domains using these languages have been shown to be NP complete and therefore not practical for software reasoning. Further research is being done to make these languages more tractable by removing some of their expressiveness to limit the types of relationships allowable and therefore improve performance.

### *3.2.1 Resource Description Framework*

In order to make truly dynamic service compositions, web services need some way to represent the knowledge contained in the XML data so services can adjust to differences in structure, and matchmaking servers can search for services based on capability instead of interfaces. The semantic web languages are an attempt to do exactly that. The Resource Description Framework (RDF) [37, 38] gives XML a standard concept of classes and properties. Classes are not just XML elements but domain terms and may have subclasses. Classes and subclasses create a domain taxonomy of possible types. Properties are relations between classes. Each property has a domain and range of possible classes, and like classes, properties may have subproperties. Together classes and properties define all the metadata in RDF. Resources are then described using triples – statements containing a resource, a property it has, and the value of that property. These statements are unambiguous because all classes and properties use universal resource identifiers (URI) – their names are linked to the ontology they are defined in.



<pre> &lt;?XML version="1.0"?&gt; &lt;element attribute="stuff"&gt;   &lt;childElement&gt;     &lt;value type="xsd:string"&gt;       foo     &lt;/value&gt;   &lt;/childElement&gt; &lt;/element&gt; </pre>	<pre> &lt;rdf:Property rdf:ID="registeredTo"&gt;   &lt;rdfs:domain rdf:resource="#MotorVehicle"/&gt;   &lt;rdfs:range rdf:resource="#Person"/&gt; &lt;/rdf:Property&gt;  &lt;rdf:Property rdf:ID="rearSeatLegRoom"&gt;   &lt;rdfs:domain rdf:resource="#PassengerVehicle"/&gt;   &lt;rdfs:range rdf:resource="xsd:integer"/&gt; &lt;/rdf:Property&gt;  &lt;rdfs:Class rdf:ID="Person"/&gt; </pre>
---	---

**Code Sample 1: XML and RDF Examples**

### 3.2.2 *Web Ontology Language*

RDF is fairly powerful. It gives XML resources specific meaning and properties so they can be more easily searched and compared by software. Class and property hierarchies provide a basis for semantic matching with exact, general, and more specific matches. What RDF does not provide is any kind of logic to provide further reasoning. If resource Bob has the parentOf relation to Bobby, it might be convenient for the system to know that x parentOf y implies y childOf x, without having to specifically write the second statement. A knowledge representation is useful precisely because it allows the program to understand relationships and infer properties that are not directly specified in the data. Using RDF instead of plain XML allows programs to infer that different structures of XML elements are actually related based on the class and property hierarchies. The Web Ontology Language (OWL) [20] is an RDF standard which adds logical properties to the base RDF taxonomies.



**Table 1: OWL Lite key word Summary**

<b>RDF Schema Features:</b> <ul style="list-style-type: none"> <li>• <i>Class (Thing, Nothing)</i></li> <li>• <i>rdfs:subClassOf</i></li> <li>• <i>rdf:Property</i></li> <li>• <i>rdfs:subPropertyOf</i></li> <li>• <i>rdfs:domain</i></li> <li>• <i>rdfs:range</i></li> <li>• <i>Individual</i></li> </ul>	<b>(In)Equality:</b> <ul style="list-style-type: none"> <li>• <i>equivalentClass</i></li> <li>• <i>equivalentProperty</i></li> <li>• <i>sameAs</i></li> <li>• <i>differentFrom</i></li> <li>• <i>AllDifferent</i></li> <li>• <i>distinctMembers</i></li> </ul>	<b>Property Characteristics:</b> <ul style="list-style-type: none"> <li>• <i>ObjectProperty</i></li> <li>• <i>DatatypeProperty</i></li> <li>• <i>inverseOf</i></li> <li>• <i>TransitiveProperty</i></li> <li>• <i>SymmetricProperty</i></li> <li>• <i>FunctionalProperty</i></li> <li>• <i>InverseFunctionalProperty</i></li> </ul>
<b>Property Restrictions:</b> <ul style="list-style-type: none"> <li>• <i>Restriction</i></li> <li>• <i>onProperty</i></li> <li>• <i>allValuesFrom</i></li> <li>• <i>someValuesFrom</i></li> </ul>	<b>Restricted Cardinality:</b> <ul style="list-style-type: none"> <li>• <i>minCardinality</i> (0 or 1)</li> <li>• <i>maxCardinality</i> (0 or 1)</li> <li>• <i>cardinality</i> (0 or 1)</li> </ul>	<b>Header Information:</b> <ul style="list-style-type: none"> <li>• <i>Ontology</i></li> <li>• <i>imports</i></li> </ul>
<b>Class Intersection:</b> <ul style="list-style-type: none"> <li>• <i>intersectionOf</i></li> </ul>	<b>Versioning:</b> <ul style="list-style-type: none"> <li>• <i>versionInfo</i></li> <li>• <i>priorVersion</i></li> <li>• <i>backwardCompatibleWith</i></li> <li>• <i>incompatibleWith</i></li> <li>• <i>DeprecatedClass</i></li> <li>• <i>DeprecatedProperty</i></li> </ul>	<b>Annotation Properties:</b> <ul style="list-style-type: none"> <li>• <i>rdfs:label</i></li> <li>• <i>rdfs:comment</i></li> <li>• <i>rdfs:seeAlso</i></li> <li>• <i>rdfs:isDefinedBy</i></li> <li>• <i>AnnotationProperty</i></li> <li>• <i>OntologyProperty</i></li> </ul>
<b>Datatypes</b> <ul style="list-style-type: none"> <li>• <i>xsd datatypes</i></li> </ul>		

### 3.2.3 Inference

The logics of OWL are based on description logic [39]. Description logic is a knowledge representation formalism closely related to propositional modal logics. The system is based on concepts and roles. Concepts make up a set of objects and roles are binary relations between the objects. Concept and role axioms create the terminology box (TBox) which is separate from the asserted instances in the Assertion Box (ABox). The inference system is used to create new assertions from the current ABox and the relationships defined in the TBox. These assertions are found using an optimized tableaux calculus[40]. There has been extensive research by Ian Horrocks [40] and others to ensure that the axioms contained in OWL are sound and tractable. This has lead to the three separate OWL specifications – OWL Lite, DL, and Full. The key words for OWL Lite are given in Table 1. The mathematical syntax for most of the axioms are given in the table below.

OWL Axioms		
Axiom	DL Syntax	Example
subClassOf	$C_1 \sqsubseteq C_2$	Human $\sqsubseteq$ Animal $\sqcap$ Biped
equivalentClass	$C_1 \equiv C_2$	Man $\equiv$ Human $\sqcap$ Male
disjointWith	$C_1 \sqsubseteq \neg C_2$	Male $\sqsubseteq \neg$ Female
sameIndividualAs	$\{x_1\} \equiv \{x_2\}$	{President_Bush} $\equiv$ {G_W_Bush}
differentFrom	$\{x_1\} \sqsubseteq \neg\{x_2\}$	{john} $\sqsubseteq \neg$ {peter}
subPropertyOf	$P_1 \sqsubseteq P_2$	hasDaughter $\sqsubseteq$ hasChild
equivalentProperty	$P_1 \equiv P_2$	cost $\equiv$ price
inverseOf	$P_1 \equiv P_2^-$	hasChild $\equiv$ hasParent <sup>-</sup>
transitiveProperty	$P^+ \sqsubseteq P$	ancestor <sup>+</sup> $\sqsubseteq$ ancestor
functionalProperty	$\top \sqsubseteq \leq 1 P$	$\top \sqsubseteq \leq 1$ hasMother
inverseFunctionalProperty	$\top \sqsubseteq \leq 1 P^-$	$\top \sqsubseteq \leq 1$ hasSSN <sup>-</sup>

  $\mathcal{I}$  **satisfies**  $C_1 \sqsubseteq C_2$  iff  $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$ ; satisfies  $P_1 \sqsubseteq P_2$  iff  $P_1^{\mathcal{I}} \subseteq P_2^{\mathcal{I}}$   
  $\mathcal{I}$  satisfies ontology  $\mathcal{O}$  (is a **model** of  $\mathcal{O}$ ) iff satisfies every axiom in  $\mathcal{O}$

Logical Foundations for the Semantic Web – p. 18/37

**Table 2: OWL Axioms in Descriptive Logic**

## 4 THISTLE Multi-Agent System

With all these great emerging technologies one might be tempted to ask why aren't there more semantic web applications? Where are the composable services to improve applications? While part of the problem is that the inference engines are still immature, another problem is that there have not been many practical applications of these emerging technologies. While the developers are waiting on the inference engines and knowledge representations, the industrial backers are still waiting on the proven applications. After all, a sound or complete logic is only a mathematical construct. Semantic applications will help demonstrate exactly how semantics can be applied and help focus the current efforts on practical endeavors. The THISTLE project at CMU is a prototype system that was implemented to help test the feasibility of these emerging technologies and give direction to further interoperability research in practical semantic applications.

## 4.1 Overview

THISTLE is a novel multi-agent system designed to solve remote network domain issues by facilitating the Problem Resolution Model set forth in [41]. THISTLE identifies network errors, collects local state, and solves policy violations using a formal knowledge representation of the current state and network policies. A concise graphical interface keeps users informed of the current encryption levels and user rights for each monitored application and displays meaningful alert messages whenever errors occur or decisions need to be made. The multi-agent system design is based on the Retsina multi-agent architecture and the individual software agents are compatible with other Retsina agents.







## 4.2 Knowledge Models

Software agents are “*autonomous, (preferably) intelligent, collaborative, adaptive computational entities*” that have the ability to affect their environment. The THISTLE agents reside on a user’s computer, so their environment is the operating system and applications that create remote access connections. The software agents affect this environment by running applications and system tools as a user would. However, in order to use these tools intelligently to aide users in solving networking problems, the agents need a way to represent the knowledge a person uses to choose remote network access connections. For the THISTLE domain, this requires a model of security terms, their application to remote network access protocols, and a formal way to describe what behaviors are acceptable. The next sections will detail the development of the Security Model, Application Profiles, and Network Policies.

## 4.3 Security Model

The primary goal of the THISTLE system was to make users aware of the security properties of their connections. Though many users have some notion that connections should be secured, most do not have the technical expertise to determine exactly what levels of protection they have and when security is in place. Just as web browsers add lock icons for SSL connections, the THISTLE system needed some graphical way to

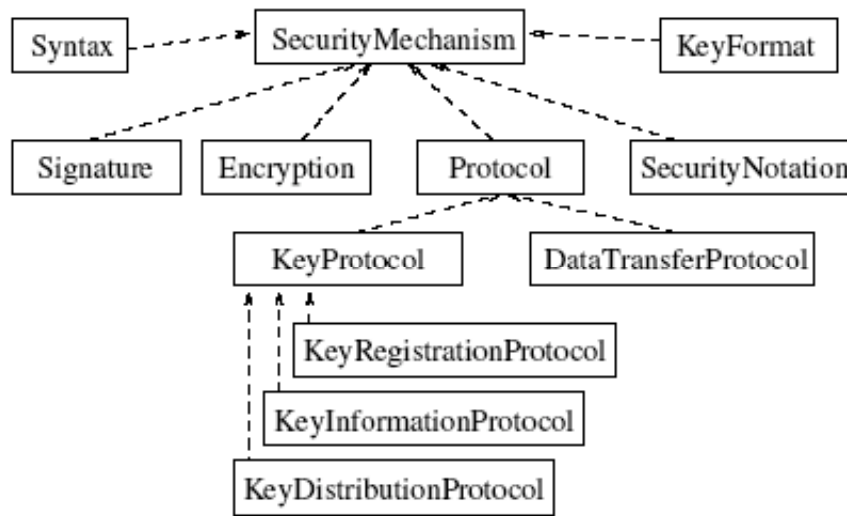
express current security properties of connections that would give users some sense of how they had violated security policies. After reviewing the results of the help desk study, it was decided that the most intuitive approach would be to provide three coarse levels of granularity for user rights and encryption, as shown in Table 3. Aaron Steinfeld then designed the accompanying icons. The user rights allow a person to visualize whether they are currently connected as a public user, a trusted user, or a full member of the network realm. The encryption icon then informs them if their data is being fully encrypted, only partially encrypted, or if there is no encryption at all. Instead of worrying users with the details associated with different types of encryption or authentication schemes, the icons assume that the administrators creating the security ontology have already done the worrying and mapped the protocols to the appropriate security levels. Applications that provide compromised encryption schemes or flawed authentication should be given their appropriate security levels in the ontology. Technical users may be interested in seeing more details, but the prototype system is geared toward non-technical users so more detailed semantics will be left for future work.

<b>Paramter State</b>	<b>Icon</b>	<b>For the task in question, the user...</b>
<b>Rights</b>		
<i>Public</i>		is recognized as being a member of the general public.
<i>Trusted</i>		has provided some information indicating they are to be trusted (e.g., a password).
<i>Realm</i>		is a member of the "home" network realm (e.g., issued an IP number by the VPN).
<b>Encryption</b>		
<i>None</i>		is using unencrypted communication.
<i>Partial</i>		is passing some, but not all, data in an encrypted manner (e.g., just the password).
<i>Full</i>		is using encryption for all data with the task server on the home network.

**Table 3: Simplified Security Parameters**

The THISTLE agents needed a formal way to determine what security level should be applied to all the remote network access services. The model needed to separate the security concepts from the instances and be flexible enough to support many different

services. Since this was exactly the goal of the semantic web for web services, OWL was the perfect fit. There were (and still are) many industrial groups attempting to define XML web service security standards[42,43] and researchers have also adapted those to DAML-S. The THISTLE Security ontology is an extension of Denker's previous credential and security ontologies [44]. The security ontology was updated from DAML syntax to OWL syntax and new Authentication and Encryption classes were added as subclasses of SecurityMechanism. The SecurityMechanism was then made a subclass of the OWL-S [45] ServiceParameter so that any SecurityMechanisms could be used in an OWL-S Profile. OWL-S Profiles were used in order to be interoperable with other web service technologies and the OWL-S Matchmaker project underway at CMU[35].

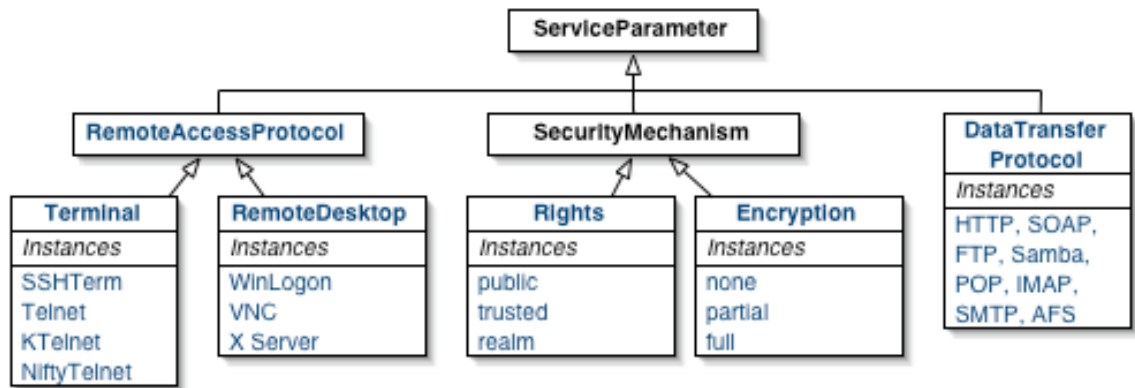


**Figure 5: Denker's original Security Ontology.**

--> denotes Daml subClassOf relations

The original security ontology defined an open interface for describing security credentials in DAML-S. A security mechanism could have properties related to the type of syntax, security notation, protocol, encryption, signature, or authentication it had. Instances were created for many of the current standards and provisions were made to simplify the addition of new standards that will inevitably become available. A separate credential ontology created links to XML standards and basic forms of authentication. For the THISTLE Security ontology, the Data Transfer Protocol was made a Service

Parameter so more general transfer protocols could be used outside of the security domain. A RemoteAccessProtocol class was also added to define remote access services as searchable OWL-S parameters. These changes are highlighted in Figure 6.



**Figure 6: Additions to the security ontology mapped to OWL-S ServiceParameter**  
Blue color indicates new terms.

#### 4.4 Application Profiles

In the THISTLE knowledge model, applications provide services and services are defined by their ServiceParameters. In the Security ontology, all of the security mechanisms used in remote access connections were given their respective levels of user rights and encryption based on the previous technical report and Aaron Steinfeld’s classifications. The simplified security levels were sometimes difficult to decide in cases like VPN encryption where traffic is encrypted over the public networks, but not encrypted inside the SCS domain.

Tool #	Tool	Rights	Encryption
1	SCS		
1.1	mail client	public	none
1.2	news client	public	none
1.3	web browser	public	none
1.4	ISP access tool	public	none
1.5	Chat client	public	none
1.6	VPN	realm	partial
1.7	SSH	realm	full
1.8	SSH tunnel	realm	full

1.9	kerberized telnet	trusted	partial
1.10	SCP	trusted	full
1.11	samba	trusted	none
1.12	Windows neighborhood	trusted	none
1.13	kerberos	trusted	partial
1.14	authentication	trusted	partial
2	Other Domains (including Andrew)		
2.1	SSL-mail	trusted	full
2.2	SSL-web	trusted	full
2.3	telnet	trusted	none
2.4	SFTP	trusted	Full
2.5	FTP	trusted	None

**Table 4: Service Security Rights and Encryption Mappings**

The Application Profiles ontology defines the service class hierarchy and relates all of these security mechanisms to the service classes. Each service is given an OWL-S serviceParameterName, authentication and encryption properties based on the simplified security ontology, and possible port and server properties used by the PortScan agent to test connections. Every application is given a ServiceProfile as defined in the OWL-S specification. The profile consists of the name and the service parameters of service defined by the profile. Thistle applications are given the security policy, which is a subclass of securityMechanism, and will be described in the next section.

```

<profile:Profile rdf:ID="MailProf">
  <profile:serviceName rdf:parseType="Resource">
    <rdf:value rdf:datatype="&xsd:string">E-Mail</rdf:value>
  </profile:serviceName>
  <profile:serviceParameter rdf:parseType="Literal">
    <Policy>
      <wsp:Policy>
        <wsp:All>
          <wsp:ExactlyOne>    <!-- add categories -->
            <profile:serviceCapability rdf:resource="#IMAP"/>
            <profile:serviceCapability rdf:resource="#POP"/>
            <profile:serviceCapability rdf:resource="#SMTP"/>
          </wsp:ExactlyOne>
        </wsp:All>
      </wsp:Policy>
    </Policy>
  </profile:serviceParameter>
</profile:Profile>

```



```

    <wsp:ExactlyOne>
      <profile:serviceCapability rdf:resource="#SSL"/>
      <profile:serviceCapability rdf:resource="#Login"/>
    </wsp:ExactlyOne>
  </wsp:All>
</wsp:Policy>
</Policy>
</profile:serviceParameter>
</profile:Profile>

```

**Code Sample 2: Application Profile in OWL**

## 4.5 Security Policies

While OWL-S provides a framework for defining service parameters, it does not have any current support for policies that use those parameters. For the actual policy syntax, THISTLE uses another web service working draft, the WS-Policy specification [46]. Although WS-Policy is only an XML standard, it provides a convenient syntax that is domain independent and has industrial support. WS-Policy was designed with some of the other WS-Security standards in mind, so it is a natural fit for the THISTLE Security Ontology.

The WS-Policy specification is fairly simple. Each Policy consists of a **<ws:policy>** element in which policy instances may define Assertions **<ws:assertions>**. Each assertion has a specific usage attribute and contains as its value some domain specific XML elements that can be used to determine if the assertion is valid. Assertions can be grouped together using **<wsp:All>**, **<wsp:ExactlyOne>**, or **<wsp:Any>**, creating logical sets of assertions.

```

<profile:Profile rdf:ID="E-MailPolicy"> <rdf:Property rdf:ID="registeredTo">
  <profile:serviceName>
    <xsd:String>
      <rdf:value>Send-E-Mail</rdf:value>
    </xsd:String>
  </profile:serviceName>
  <profile:serviceParameter rdf:parseType="Literal">

```

```

<Policy>
  <Service>Send-E-Mail</Service>
  <Condition>
    <state entity="Host" property="HostDomain" value="NonCMU"/>
  </Condition>
  <wsp:Policy>
    <wsp:Assertion wsp:Usage="Required">
      <AuthenticationLevel rdf:resource="&security;#Realm"/>
    </wsp:Assertion>
  </wsp:Policy>
  <Message>
    Rights too low for Sending Mail. Select Option or only read mail.
  </Message>
</Policy>
</profile:serviceParameter>
</profile:Profile>

```

### Code Sample 3: E-Mail Policy

THISTLE policies are OWL-S service parameters with a WS-Policy element embedded in a custom **<Policy>** element. The policy element provides the logical preconditions for the **<wsp:Policy>**. Policies are applied to a specific service type and have optional conditions. Conditions consist of a single statement triple specifying the resource, property, and value which enables the policy. In the example above, the Send-E-Mail service (which is the super class of SMTP) requires Realm authentication when the local host is not on the CMU network.

## 4.6 THISTLE Architecture

THISTLE divides the remote network access application into different types of agents with different roles and varying levels of sophistication.

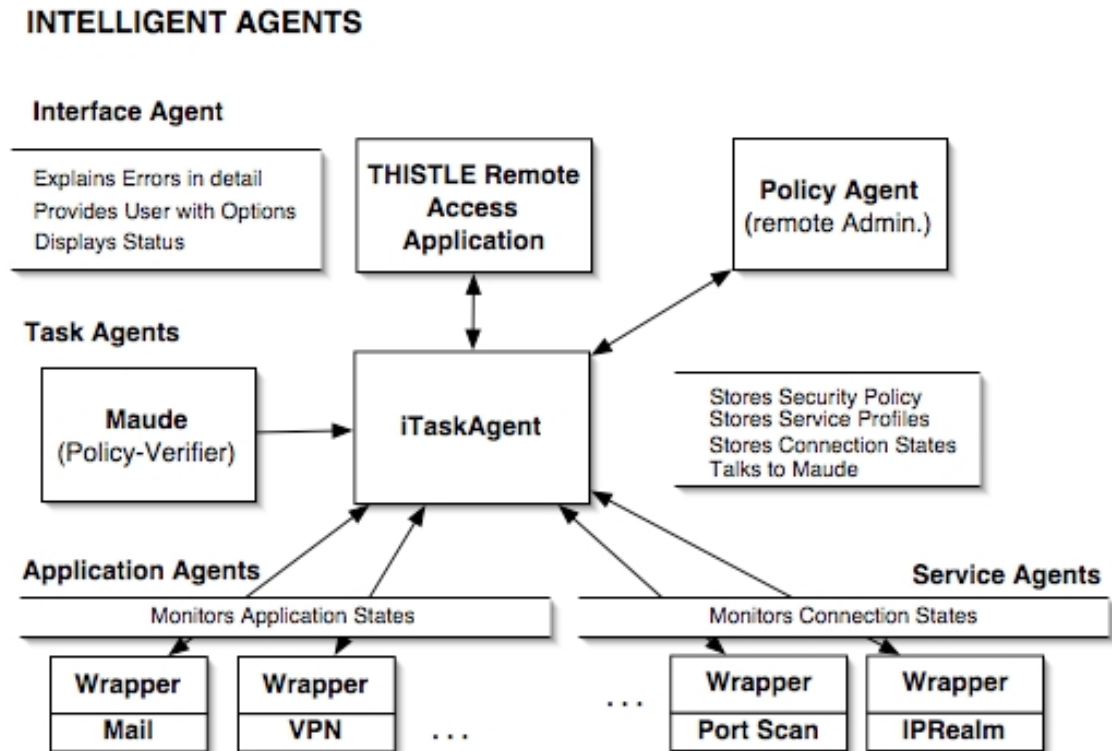


Figure 7: THISTLE Architecture

Information about the current state of the system is gathered from the Application, Service, and Policy Agents. The Application Agents are currently simple Information Agents[14] that monitor the state of an application and expose the services it can provide. In a similar manner, Service Agents wrap services that the operating system or networking tools provide. The local Policy Agent stores and parses all the network policies and is responsible for communicating with any external policy sources. All user interaction with the system is coordinated through the Interface Agent. Application planning is done in the Task Agent. It collects all of the state information from the Application Agents, checks the system state using the Service Agents, and then uses the policies from the Policy Agent to verify that there are no policy violations. When errors

occur, the Task Agent determines all the possible correct network options and sends the choices to the Interface Agent.

Each agent is a separate executable application with its own Retsina Communicator [48] and knowledge model. All the information exchanged by the agents is modeled using OWL ontologies and transmitted in KQML [49]. The next section will go into detail about the purpose of each agent, their knowledge representations, and what design choices were made in their development.

## **4.7 Application Agents**

Application Agents serve as the virtual body of the THISTLE system, allowing agents to affect the remote network access connections. Every application that a person ordinarily uses to connect to a remote network (email, VPN, iPass, Web Browser, etc) is exposed as an Application Agent. The Task Agent sends Application Agents commands and they interpret and execute them, starting up applications and configuring settings. While the applications are running, the Application Agents monitor their status and update the task agent with the current state. In order to coordinate their activities, both the Application and Task agents have to share a common knowledge representation or have some interpreter between the two layers. The Task Agent needs to know not only what services are available, but also how those services can be composed into remote network connections and how each service will affect the user rights and encryption of the connections. To provide this flexibility, the Application Agent provides THISTLE with an OWL-S service profile for the application and the methods to call to execute those services. The profiles are also stored on the Policy Agent reflecting the idea that vendors might provide default profiles while administrators may need to set their own custom profiles for their domains.

Using the OWL-S service profile gives THISTLE an understanding of not only what services an application provides, but also how those services affect the security of remote connections. OWL-S provides a standard framework for defining properties of services in

a way that distinguishes them from other services that may offer similar functions. OWL-S builds on the simple type definitions given in WSDL to provide semantically meaningful inputs, outputs, preconditions, and effects (IOPE) for the services. The standard framework allows generic semantic service matchmaker applications to be written, such as the DAML-S Matchmaker at CMU [35]. The service vendors then fill in their own properties based on service specific ontologies. In the case of THISTLE, the service properties are based on the modified Security Ontology of Denker, using the newly defined Encryption and Authentication rights as derived properties of the services.

Ideally Application Agents would incorporate the application code as their own services, or call well defined API's exposed by proprietary applications to make access to each application's internal events and methods as seamless as possible. It would also be possible to create a standard logging interface that output directly to Application Agents using well defined terminology (perhaps an apache logging appender with OWL messages). While this can be done with some effort on open source projects, most commercial applications are designed for human users and provide only error logs or visual alert messages.

Due to time constraints, THISTLE currently has only two Application Agents, a VPN Agent and a simple Application Agent stub that calls executables. The VPN Agent is a full-featured wrapper for Cisco's VPNClient application. After starting the VPNClient, the agent parses the VPN log file and checks for errors, warnings, new connections, or the heartbeat messages to determine the state of the connection. The agent still requires manual user interaction to connect, sign on, and disconnect from the network, but all the states are determined automatically. The Mail Agent currently just uses the stub to start up the Mail application.

## **4.8 Service Agents**

The Service Agents provide the Task Agent with access to the current state of the local machine. These are custom functions that the Task Agent can use to test and diagnose the states returned by the Application Agent. For instance, when the VPN fails to establish a

secure tunnel to the gateway, there may be a local or server side firewall blocking connections. By checking the networking settings and using a port scan of the server, the Task Agent can isolate possible errors that the application would have no way to determine.

THISTLE currently has two Service Agents that were adapted from code initially written by Rahul Singh. The PortScan Agent has a list of known ports and protocols and is used to verify the connection to any remote server that a service needs (IMAP port, Kerberos Authentication Server, VPN Gateway). The IPRealm Agent is used to determine the type of the current network domain. Different policies apply to public, trusted, or local domains and these IP ranges need to be set by the network administrator. As the project progresses, more networking services will be added to provide knowledge about packets and routes through the network.

## **4.9 Policy Agent**

The local Policy Agent acts as a link to the knowledge created by the network administrators. In the absence of any network connection, it allows the Task Agent to retrieve the current application profiles, network policies, and security ontology. The Policy Agent also contains convenience methods for parsing and retrieving information contained in the ontologies. This helped to keep the OWL syntax and Inference mechanisms confined to a single object.

## **4.10 Task Agent**

The Task Agent is responsible for monitoring and verifying the remote access connections in THISTLE. Task Agents task the information agents and maintain a high level view the system in terms of the Security, Application, and Policy ontologies. The application agents monitor their given applications and update the Task Agent when services are started, running, or have errors. The service agents provide further updates on operating system and network level actions for determining the current state of the local system. Based on these updates, the Task Agent creates an internal model of the current running connections and uses the Security and Application ontologies to

determine the user rights and encryption levels each of the composed services should have. Connections are checked for policy violations using the Maude Agent whenever any states change in the system.

#### 4.10.1 Policy Verification in Maude

All connections are verified using a RemoteAccess program written in Full Maude to search for any violations. The Maude module defines connections as a list of running processes which have composed properties. Each process contains an application and a service. Applications have a list of available services and their own properties. The application names and service types are taken directly from the Service Ontology.

```

--- CLASSES ---
class PROPERTY      | name : Qid, value : Qid .
class SERVICE       | name : Qid, type : Qid, properties : OidList .
class APPLICATION   | name : Qid, services : OidList,
                    | properties : OidList .
class PROCESS       | application : ApplicationID, service : ServiceID .
class CONNECTION    | properties : OidList, processes : OidList .
class OWL-CLASS     | subClasses : OidList .

--- POLICY SPECIFICATION ---
class CONDITION     | instance : Qid, predicate : String,
                    | property : PropertyID .
class POLICY        | usage : String, servicetype : Qid, condition : Qid,
                    | requirements : OidList .

```

#### Code Sample 4: RemoteAccess Full Maude Classes

Policy objects are composed of a condition, requirements list, OWL service type, and a usage. This is basically a direct translation of the policy element used in the Policy Ontology. To determine if a condition has been violated, a checkPolicy message is sent with the connection identifier and the current Configuration of the system. A search command is then issued to see if the conditional rewrite rule fires. If the connection has a process with the service type or subclass specified in the policy, and is in the domain specified by the condition, then the conditional checks will verify that the connection has achieved any required properties of the policy. If not, then APolicyViolation() message is returned.

```

crl [policy_checker] : checkPolicy( CID )

```

```

< Host      : APPLICATION | name : hostName,
                           services : hostServices,
                           properties : hostProps >
< conProp   : PROPERTY    | name : conpname, value : conpvalue >
< reqProp   : PROPERTY    | name : reqpname, value : reqpval >
< Cond      : CONDITION   | instance : Host, predicate : any:String,
                           property : conProp >
< Pol       : POLICY      | usage : pusage, servicetype : OwlClass,
                           condition : Cond,
                           requirements : reqProp reqs >
< OwlClass  : OWL-CLASS   | subClasses : subClss >
< ServID    : SERVICE     | name : servName, type : ServType ,
                           properties : servProps >
< AppID     : APPLICATION | name : appName, services : ServID
                           appServices, properties : appProps >
< PID       : PROCESS     | application : AppID, service : ServID >
< CID       : CONNECTION  | properties : connProps,
                           processes : PID connservices >
=> APolicyViolation( CID, Pol, pusage, reqProp )
if ServType in subClss /\ conProp in hostProps
                           /\ not ( reqProp in connProps )
.

```

### Code Sample 5: Policy Verification in Full Maude

When the Task Agent needs to verify a connection, all the internal knowledge about the current state of the system is translated into the Maude syntax defined in the RemoteAccess module. The Maude Agent then invokes the Maude environment from inside java and then passes each connection with a checkPolicy() message to Maude. If a policy violation is found then the Task Agent sends Maude a new configuration which includes all the possible applications, the violated policy, and a FindOptions() message.

```

crl [options] : FindOptions( servProps , CID )
  < AppID : APPLICATION | name : appName, services : ServID
                           appServices, properties : appProps >
  < ServID : SERVICE    | name : servName,
                           properties : reqProp props >
  < CID : CONNECTION   | properties : connProps,
                           processes : connservices >
=> < CID : CONNECTION | properties : reqProp props connProps,
                           processes : < "Process: " + string( appName ) + ":"
                                       + string( servName ) >
                           connservices >
  < < "Process: " + string( appName ) + ":" + string( servName ) >
      : PROCESS | application : AppID, service : ServID >
  < ServID : SERVICE    | properties : props >
  < AppID : APPLICATION | name : appName, services : appServices,
                           properties : appProps >
  Option( AppID, ServID, CID )
if Composable in props /\ reqProp in servProps .

```

### Code Sample 6: Find Options conditional rewrite rule



The options rewrite law will create Option messages containing the application, service, and the connection the service needs to be composed with to satisfy the given policy violation. These Options are then translated back into the Task Agent, which sends the options to the Interface agent for the user to choose.

## **4.11 THISTLE User Interface**

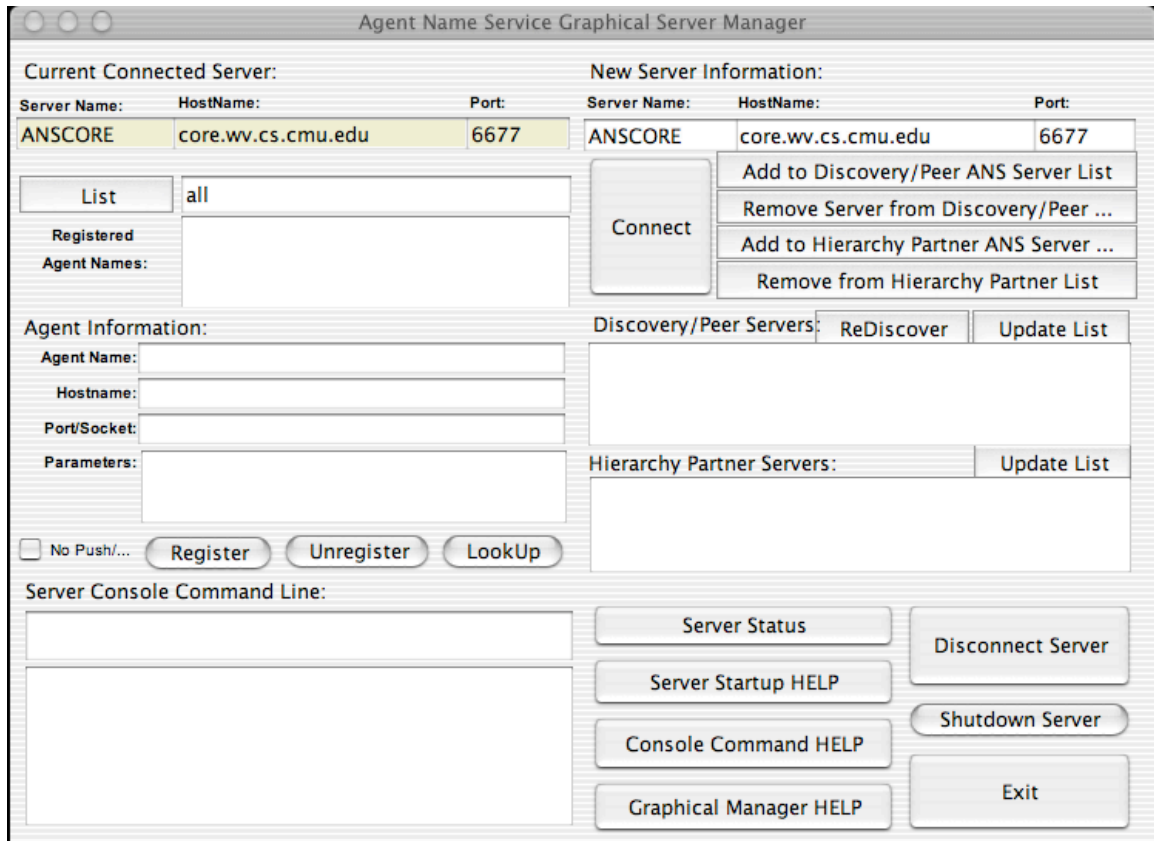
The Interface Agent is responsible for all the interactions the remote network user has with the THISTLE system. The user interface was designed by Aaron Steinfeld to provide non-technical users with a simple and appealing graphical representation of their current connection properties while being as minimally intrusive as possible. Each monitored application is given a single collapsible security panel in the application window. The security panel then contains an alert pane for messages, an option pane for displaying service options, and a status pane with the current connection status. When messages arrive from the Task Agent, the appropriate panes slide down and display the new information.

The key to the Thistle interface is the security abstraction provided by the user rights and encryption icons. The Security Ontology defines three instances of the Authentication class – Realm, Trusted, Public, and three instances of the encryption class – Full, Partial, None. The Service Ontology then defines every protocol in terms of these instances. Each remote access service utilizes some security or protocol and so inherits their rights and encryption levels. When connections are composed with VPN, SSH, or a similar service, the connection inherits the higher rights and encryption levels granted by the tunnels.

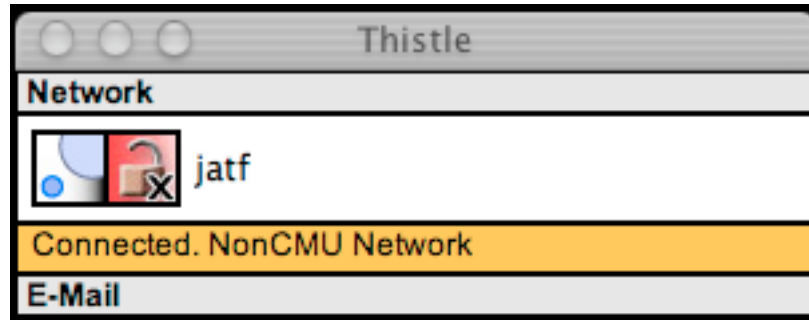
### *4.11.1 Email Scenario*

To demonstrate the capabilities of THISTLE, this section will step through a typical policy verification scenario in which a user starts the system and attempts to send email through the SCS from outside the network.

1. The Agent Name Server (ANS) is initialized. The ANS can be local or on another server.

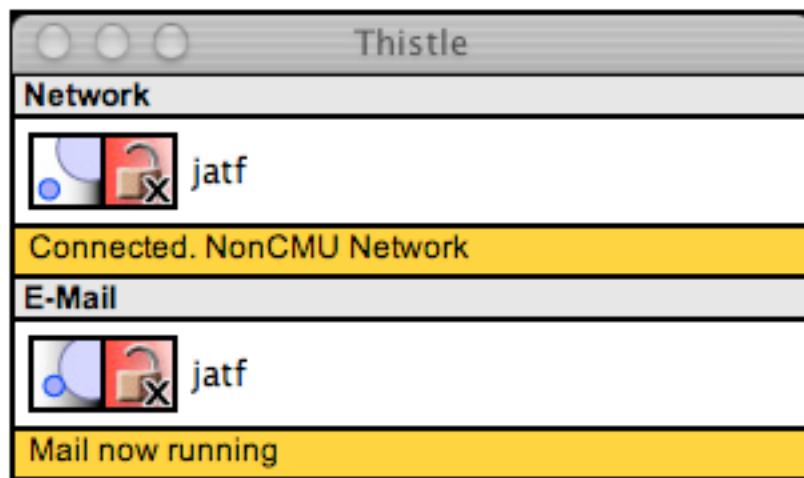


2. THISTLE system is initialized.
  - a. The Policy Agent, IPRealm Agent, PortScan Agent, and User Interface Agents are started.
  - b. Each agent registers with the ANS and waits for messages.
3. Task Agent is started.
  - a. The Maude Agent is initialized loading Full Maude and the RemoteAccess program.
  - b. The ontologies are requested from the Policy Agent.
  - c. The IPRealm Agent is told to monitor the Realm at 5 second intervals.
4. Task Agent receives the current Realm and sends an update to the User Interface.



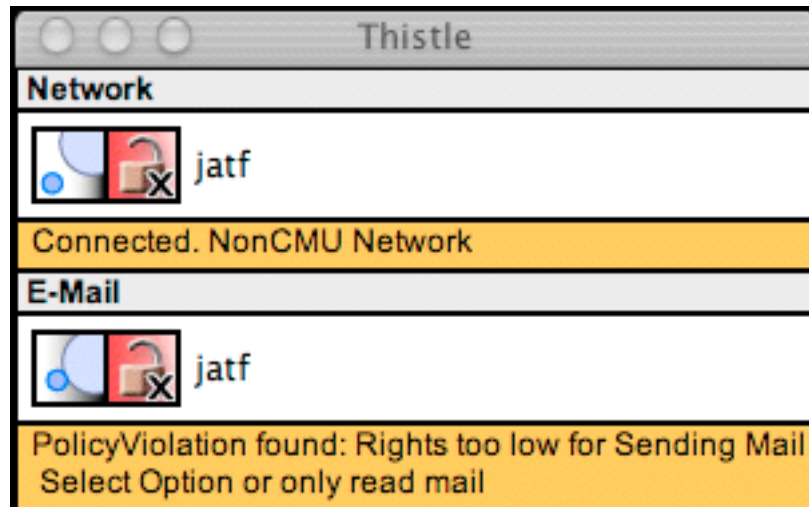
**Figure 8: User Interface after initial network discovery**

5. User starts the agentified Mail application.
  - a. Actual Mail application is launched.
  - b. Task Agent informs the port scan agent to begin scanning the ports listed for all the services Mail offers. SMTP, IMAP, POP



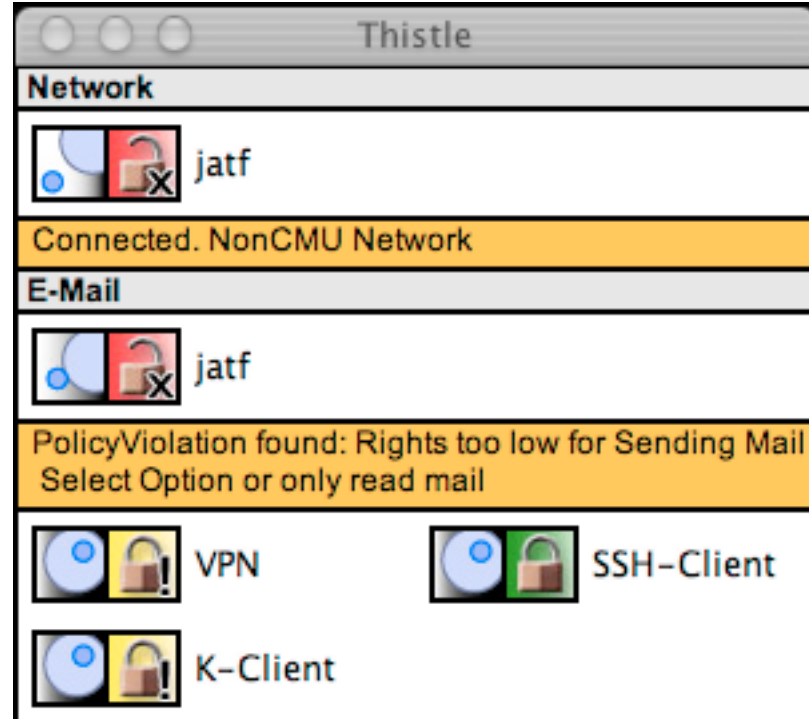
**Figure 9: Mail Application started**

6. Policy Violation Found. The SCS domain allows users to check their email remotely without realm rights, but to send mail a user must achieve realm status. The standard Mail login only achieves trusted user rights.



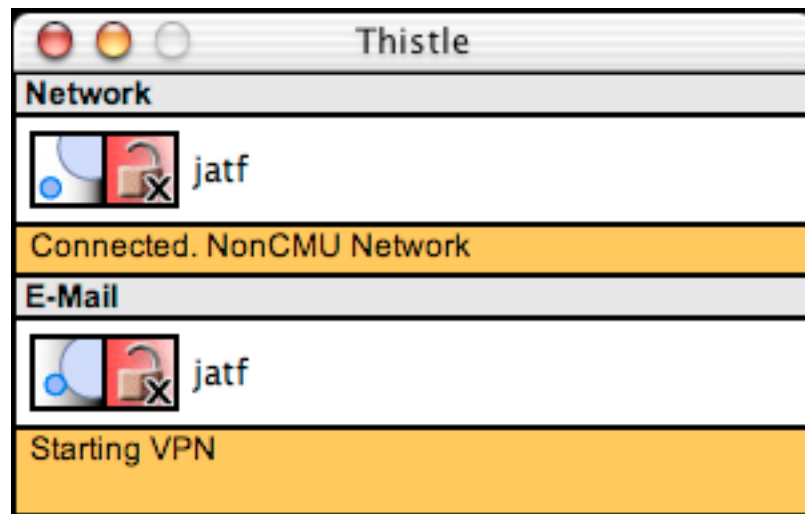
**Figure 10: User Interface with E-Mail Policy Violation**

7. Service options and security models are given. To gain realm rights the user can start up VPN, open an SSH-Client, or use Kerberos with the K-Client.



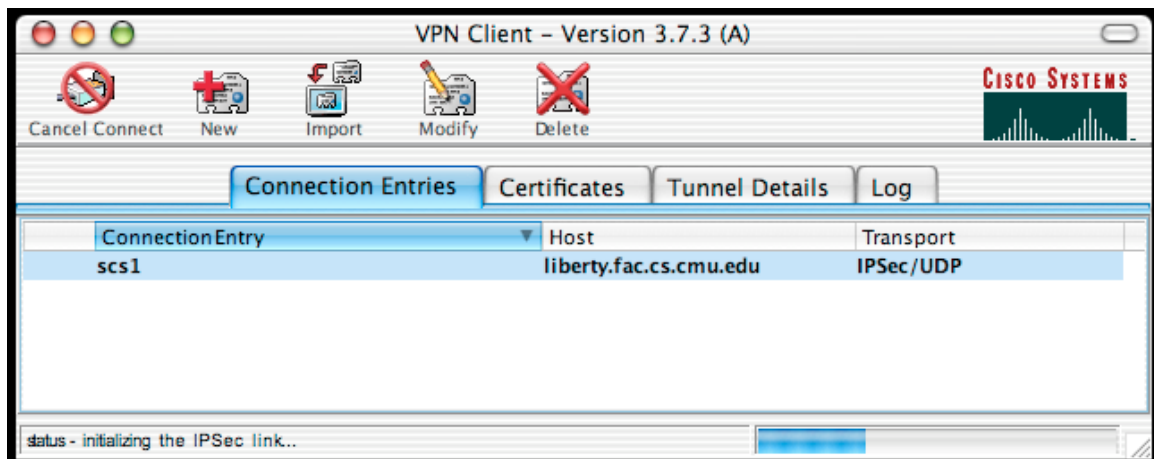
**Figure 11: User Interface with Options for E-Mail Policy Violation**

8. The user elects to start up the Cisco VPN client. Clicking on the VPN icons launches the VPN Agent and removes the options.



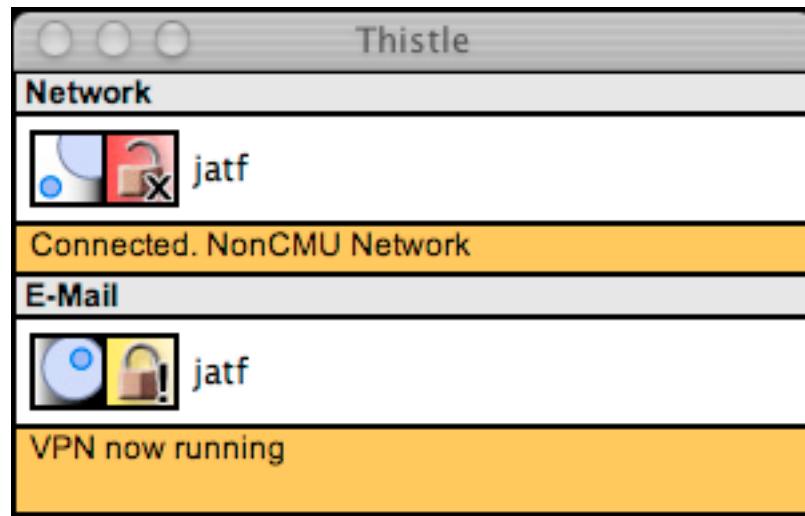
**Figure 12: User Interface after VPN is started**

9. The VPN starts up and the VPN Agent begins monitoring the VPN logs to determine if there are any errors. The VPN rights and encryption aren't applied to the email connection until VPN actually connects.

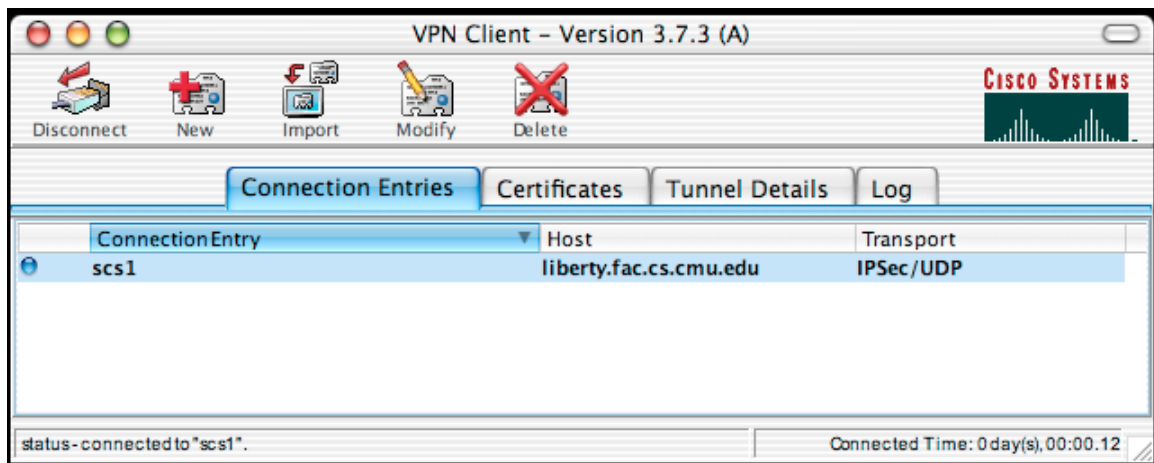


**Figure 13: VPN Agent waits for the user to connect the VPN**

10. User connects through the VPN. A secure tunnel is established and the VPN Agent informs the Task Agent of the running ssl-tunnel. The Interface Agent is updated to reflect this change.

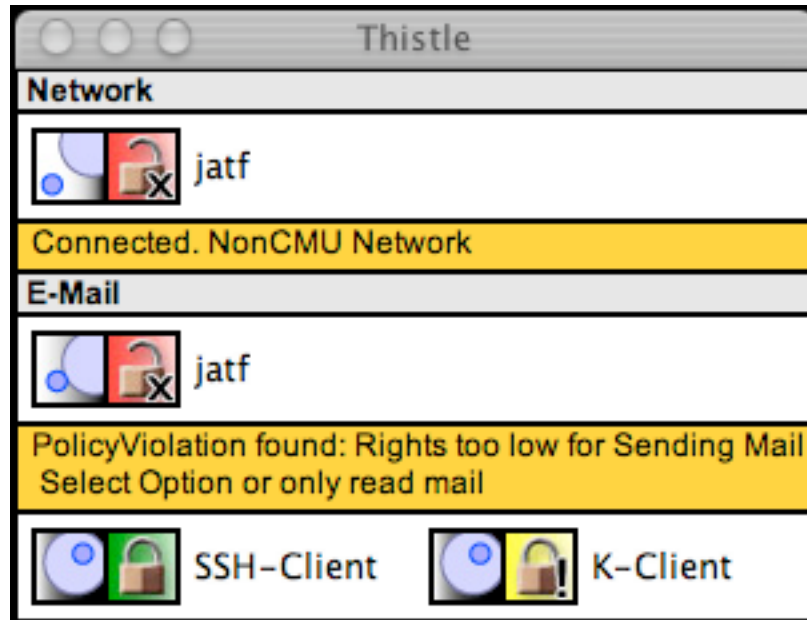


**Figure 14: User Interface with VPN service composed**



**Figure 15: VPN Client has connected.**

11. An error occurs at the VPN gateway (port is blocked by an intervening firewall). VPN rights and encryption are removed, and the new options are given.



**Figure 16: VPN Application blocked, new rights needed**

## 5 Discussion

The THISTLE system was designed with three main objectives. The first objective was to make users more aware of the security properties of their remote access connections. To this end, the user Interface Agent was designed around a simplified security model with easily understood icons. The second objective was to improve help desk systems by automating the problem resolution model and proactively solving errors without the intervention of the help desk. THISTLE solves this objective by using a multiagent system with agentified applications to monitor connection states and verify network policies. THISTLE's third objective was to develop a practical application to demonstrate the benefits of using a formal knowledge representation language. All the domain knowledge in THISTLE is modeled using OWL and policies are verified using Maude. This section will evaluate the current system's ability to meet these objectives by discussing its achievements and pointing out its weaknesses.

### 5.1 Achievements

While the THISTLE system is still a prototype, it has already achieved many of project objectives. The user interface is very intuitive and, although THISTLE has not been rigorously tested with non-technical users, the simplified security model does seem to adequately and succinctly define the properties of remote network access connections. The rights and encryption icons are easy to recognize and they are color coded with additional tool tips to aide new users. The first three steps of the automated Problem Resolution Model have been implemented. Application Agents and Service Agents are able to autonomously monitor the state of remote access connections and identify when errors occur. The Task Agent has demonstrated the ability to identify network policy violations based on connection, allowing the system to determine if errors should occur without waiting for services to fail. The Task Agent has also demonstrated the ability to solve network issues by using its knowledge base to offer options with the same services and improved security properties. Many new technologies have been tested and integrated into the THISTLE system. Three OWL ontologies were created extending existing research ontologies and incorporating recommended industry standards such as



WS-Policy. All the agents make use of OWL for their knowledge representations and use the open source Jena system to parse and make inferences on the ontologies. The Task Agent employs a policy verification program in Full Maude to test the abilities of a formal reflective programming language and experience was gained in modeling systems using a logical formalism. The autonomous agents are able to each perform their own tasks and simplify the tasks of others by sharing domain knowledge. The Task Agent was able to dynamically compose simple virtual services by using the Application Agents to run applications. Communication between agents has been achieved using both the Retsina infrastructure with KQML and web service communications using SOAP messages sent with the open source Axis package.

## **5.2 Difficulties**

### *5.2.1 Emergent Technologies and Architectural Changes*

This project had to deal with many difficulties due to the nature of the emerging technologies being integrated and its research focus. Changing standards, beta releases of research tools, the steep learning curves for the technologies, and changes in the system specifications all slowed the development of the THISTLE system. Developing ontologies seemed easy enough, but the effectiveness of the ontologies was dependent on the abilities of the logical layer, the inference engines, which used the ontologies to produce the resulting agent behavior. After researching JTP [50], RACER [51], and custom inference engines, HP Lab's Jena package [52] was chosen to parse and provide simple inferences. Though easy to configure and integrate, the original Jena 2.0 system proved to be very slow with only rudimentary access to OWL axioms and incomplete inferences. After parsing and retrieving class and property definitions, much of the inferencing had to be moved away from Jena. The system was then designed to leverage the ongoing research with the DAML-S and now OWL-S matchmaker, using the virtual machines to find service options and Maude to verify that a chosen option was secure. All the communications were converted from Retsina KQML messages to SOAP calls and the agents were given web service interfaces. However, as the original project deadline approached, the matchmaker was still under development and had not yet been separated from the UDDI server. Integrating the matchmaker would have required a UDDI server,

full web server with Axis, Agent Name Server, Maude, and all the agents to be running in the background of a user's remote system to aide with connections. Integration with the matchmaker was left for future work, and the system was reduced to the running agents and Maude with a local or remote Agent Name Server. This simplified the system, but put more emphasis on the Maude application that now had to handle diagnosis, verification, and composition of services. This created multiple internal views of ontologies, with Jena having its own internal ontology structure, Maude needing to support reasoning to perform the policy verifications, and the java Task Agent having another view of the knowledge to translate between the two systems. Adding features required changes to all three models and the increased complexity introduced both inefficiencies and errors into the system.

### *5.2.2 Developing a Reflective Logical Formalism*

While the logical formalism of the remote access domain in Maude is now one of THISTLE's best features, the reflective programming language was the most challenging technology to learn and integrate. The Maude system is designed from a formal methods perspective, which is very unique. Maude provides a logical foundation and metalanguage for defining syntax instead of keywords and familiar operators. Maude programs create a domain algebra of sorts and equations instead of objects with an API. Maude uses rewrite rules with local logical transitions to create concurrent systems instead of interwoven loops and method calls. As with all new perspectives, this required some effort to get comfortable with. This effort was further increased by the limited debugging capabilities of the Maude runtime environment. Maude is a semi-compiled language and Full Maude is written entirely in Maude. When tracing is enabled in the system Full Maude runs through tens of thousands of rewrite rules to parse and run a search theory. Without knowing what is important and being familiar with how to filter the output, the trace is useless. When traces are disabled, Maude simply returns. A failed search will return nothing but the number of rewrite rules executed, and improper searches can cause Maude to fail with no errors given. Modules have to be tested line by line to find parse errors which get increasingly difficult to find as new syntax is created. Maude has extensive documentation (260 pages) on its logical foundations and another

173-page primer written for new users, but many examples from these documents need to be modified to run correctly (causing much confusion) and they do not focus on the actual errors that can occur when attempting to learn the system. The system is fine for experienced users, but required painstaking trial and error testing to learn.

Since the start of the THISTLE prototype in August, many of these technologies have made significant improvement. The W3C has improved the OWL status to an official recommendation. Maude 2.1 has been released and Mobile Maude is developing direct socket communication with external programs. The DAML-S matchmaker has been greatly improved and the W3C and Oasis groups have made significant progress towards standardizing the other web services technologies, including security, UDDI, and process specifications. Also more commercial semantic applications are also making their way to market, further validating the usefulness of semantics.

## **5.3 Future Work**

### *5.3.1 More, Smarter Agents*

The THISTLE MAS is still a prototype that requires more work and new features before the system can be deployed across SCS. More Application Agents with better access to application methods are needed to proactively change settings not available in configuration or log files. For instance, a Mail Agent running an open source mail application could set preferences while the application runs and read incoming mail to check for relay errors and bounced messages. It would also be notified of new states by application events instead of having to constantly poll a log file as the VPN agent does. The basic system also still needs to learn and reuse knowledge derived from the solution of policy violations. The current system is event driven and has an understanding of policies and current connections, but does not have a process model for service operation. While it is helpful to know that services have failed and find solutions, it would be better to learn where errors occur in the connection process and match these errors to different solutions. This would allow the Application Agents to model the operation of each application's services and map errors at any step to diagnostic and recovery methods. If a

new error occurred at any step in the process, or the application did not follow the given process model, administrators could be automatically informed of the error and given the process with the current system wide states at each step and the diagnostics performed. The Task Agent would carry out the same process verification using the high level view of the system and reporting in terms of ontology classes. When the administrators diagnose the error, their solution would be added to the process models and then distributed to all the agents in the system. The process models and current ontologies would allow the agents to be more proactive in solving issues before they arise and serve to quickly learn all the errors common to the system. This work could be done in parallel with the efforts of the OWL-S group or utilize the BPELWS or WSCI proposed industry standards.

### *5.3.2 Improved Architecture*

To facilitate the addition of the process model and multiagent learning, the current architecture needs to be streamlined by either embracing Maude or a new java based semantic reasoner and eliminating the different ontology representations. All the thinking should be done using one internal model and one inference engine and this should probably also be linked with the planning and scheduling components of the system. If Maude is going to be used then the manual java translation of ontologies and connection states should be changed to an XSLT script and the java Task Agent should only be used for communication and scheduling. The RemoteAccess program in Maude would also have to be upgraded to provide nearly full OWL compatibility and add new modules for the process specification being used. This would require a new algebra and use of more of Maude's advanced features. If it were decided that Maude is too difficult to use, then a pure java agent would need to have its own internal ontology representation and inference mechanism. A new OWL compatible knowledge base and process planner should be added to the general agent classes.

### *5.3.3 Knowledge Representation*

The current agents make use of OWL ontologies and a custom Maude application to understand remote access connections and communicate knowledge. However, due to the

limited capabilities of the inference mechanisms in Jena and the custom Maude code, knowledge that should be in the internal semantic knowledge base is still being processed in java. The Task Agent has its own data structures for modeling connections that rely on the semantics, but also extend them. To gain the full advantage of having a semantic knowledge base, the agents need to only “think” in OWL. All the internal state should be represented as instances of OWL classes. The difficulty with this approach is that the logics of OWL are limited. The OWL axioms are useful for describing classes, but can’t replace programmatic rules that are needed to execute based on the instance knowledge. While one could develop a policy violation class in OWL, inferring that a policy is in violation and then that an action must be taken, requires a very adept interpreter for the inference engine. However, once an application asks for a specific class name to perform processing on its properties, the application is again dependent on the ontology. Changing the name of a class will cause errors and adding new properties or classes will not provide the agent with any new features until it is coded to use them. This dependency is fine for static domains such as the THISTLE system, where class structures are constant and operations on the instances are also set, but research should be done to determine how these dependencies could be avoided.

#### 5.3.4 *Extras*

There are a few other features that were not mentioned in the project goals that would be useful for future systems. Agent communications could be optimized by having different protocols for communicating locally versus over the network. Security in the agents has also not been addressed. The Task Agent has control of many applications so its actions need to be monitored and malicious users must be prevented from altering ontologies or sending faulty requests to agents and administrators. Machine learning techniques such as Bayesian networks might be explored to model user preferences and execute choices without waiting for user intervention.

## **6 Conclusion**

Thistle has demonstrated that a multiagent system utilizing formal knowledge representations of the remote access domain is able to greatly improve the effectiveness of the help desk system. By automating the problem resolution model with software agents Thistle is able to proactively solve networking errors on the local computers. A simplified security model and well-designed graphical user interface make users aware of the present security state of their connections. The prototype system has also tested the viability of emerging technologies in the areas of knowledge representation and service oriented architectures, and shown how future work could apply these results to deploy Thistle to SCS remote access users. The Thistle system was a successful prototype, but much work still needs to be done to make semantic applications commercially viable.

## 7 References

1. ACM Special Interest Group on University and College Computing Services  
<http://www.acm.org/siguccs/>
2. Graham, J. & Hart, B. (2000). Knowledgebase integration with a 24-hour help desk. In *Proceedings of the 28th Annual ACM SIGUCCS Conference on User Services, October 2000* (pp. 92-95).
3. Gormly, Brown. Rapid help desk revitalization. In *User Services Conference Proceedings of the 31st annual ACM SIGUCCS conference on User services, 159-162* 2003, San Antonio, TX, USA.
4. Serviceware Knowledge Management for Help Desks  
<http://www.serviceware.com/solutions/main.asp>
5. Primus Knowledge Management for Help Desks  
<http://www.primus.com/solutions/knowledgeSolutions/helpDesk/>
6. Parature Knowledge Management for Help Desks  
<http://www.parature.com/helpdesk.aspx>
7. A. M. Steinfeld, R. Sanghi, J. A. Giampapa, D. Siewiorek, and K. Sycara. An examination of remote access help desk cases. Technical Report CMU-CS-03-190, Computer Science Department, Carnegie Mellon University, September 2003.
8. Terri L. Lenox, Terry R. Payne, Susan Hahn, Michael Lewis and Katia Sycara. "Agent-based aiding for individual and team planning tasks." In *Proceedings of IEA 2000/HFES 2000 Congress*, 2000
9. Terry R. Payne, Rahul Singh and Katia Sycara. Calendar Agents on the Semantic Web, *IEEE Intelligent Systems*, Vol. 17(3), pp84-86, May/June 2002. Copyright 2002, IEEE Computer Society.
10. Yunwen Ye. Programming with an Intelligent Agent. *IEEE Intelligent Systems*, pp. 43-47, May/June 2003.
11. Sycara, Katia P., Multiagent Systems. *AI Magazine*, 79-92, Summer 1998
12. Honavar, Vasant. Intelligent Agents and Multi Agent Systems. In *IEEE CEC 99*, Washington, D.C., July 1999.
13. Stone, Peter, Veloso, Manuela. Multiagent Systems: A Survey from a Machine Learning Perspective. In *Autonomous Robotics* volume 8, number 3. July, 2000.
14. Sycara, K., Paolucci, M., van Velsen, M. and Giampapa, J., The RETSINA MAS

Infrastructure, in the special joint issue of Autonomous Agents and MAS, Volume 7, Nos. 1 and 2, July, 2003.

15. The Maude system. <http://maude.cs.uiuc.edu/>.

16. Manuel Clavel, Francisco Dur'an, Steven Eker, Patrick, Lincoln, Narciso Mart'1-Oliet, Jos'e Meseguer, Carolyn Talcott. Maude 2.0 Manual Version 1.0. June 2003

17. Combs, Theodore. Maude 2.0 Primer v1.0. August 2003

18. Manuel Clavel, Francisco Duran, Steven Eker, Patrick Lincoln, Narciso Marti-Oliet, Jose Meseguer, Jose F. Quesada. Maude: Specification and Programming in Rewriting Logic. Theoretical Computer Science, 2001

18. F. Duran and J. Meseguer, A Church-Rosser Checker Tool for Maude Equational Specifications, Technical Report, Universidad de Malaga and SRI International, July 2000

19. Jim Woodcock, Jim Davies. Using Z: Specification, Refinement, and Proof. Prentice Hall, July 1996.

20. The OWL Web-Ontology (WebOnt) Working Group  
<http://www.w3.org/2001/sw/WebOnt/>

21. David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste. Project Aura: Towards Distraction-Free Pervasive Computing. IEEE Pervasive Computing, special issue on "Integrated Pervasive Computing Environments", Volume 21, Number 2, April-June, 2002. pp. 22-31.

22. Kishore Channabasavaiah, Kerrie Holley, Edward M. Tuggle, Jr.. Migrating to a service-oriented architecture. Technical Report, IBM developerWorks, December 16, 2003.  
<http://www-106.ibm.com/developerworks/webservices/library/ws-migratesoa/>

23. Mark Colan. Service-Oriented Architecture expands the vision of Web services. Technical Report. IBM developerWorks, April 21, 2004.  
<http://www-106.ibm.com/developerworks/library/ws-soaintro.html>

24. Service Oriented Architectures. <http://www.service-architecture.com/>

25. A NATION ONLINE: How Americans Are Expanding Their Use of the Internet Washington, D.C. February, 2002 <http://www.ntia.doc.gov/ntiahome/dn/>

26. The eXtensible Markup Language. <http://www.w3.org/XML/>

27. Erik Wilde. XML Technologies Dissected. *IEEE Internet Computing*, SEPTEMBER



• OCTOBER 2003, pp. 74-78

28. XML Protocol Working Group. <http://www.w3.org/2000/xp/Group/>

29. Web Service Description Language. <http://www.w3.org/TR/wsdl20/>

30. Universal Description, Discovery, and Integration Specification.  
<http://www.uddi.org/specification.html>

31. Web Service Choreography Interface 1.0. <http://www.w3.org/TR/wsci/>

32. Business Process Execution Language for Web Services 1.0. <http://www-106.ibm.com/developerworks/library/ws-bpel/>

34. José M. Vidal, Paul Buhler, Christian Stahl. Multiagent Systems with Workflows., *IEEE Internet Computing*, JANUARY • FEBRUARY 2004, pp. 76-82

35. Katia Sycara, Massimo Paolucci, Anupriya Ankolekar and Naveen Srinivasan, "Automated Discovery, Interaction and Composition of Semantic Web services," *Journal of Web Semantics*, Volume 1, Issue 1, September 2003, pp. 27-46.

36. Charles Petrie, Christoph Bussler. Service Agents and Virtual Enterprises: A Survey. *IEEE Internet Computing*, JULY • AUGUST 2003, pp. 68-78

37. The Resource Description Framework <http://www.w3.org/TR/rdf-primer/>

38. The RDF Vocabulary Description Language 1.0 : RDF-Schema  
<http://www.w3.org/TR/rdf-schema/>

39. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, Peter Patel-Schneider. The Description Logic Handbook Theory, Implementation and Applications. Cambridge University Press, January 2003. <http://dl.kr.org/>

40. Ian Horrocks. Reasoning with Expressive Description Logics: Theory and Practice. Presentation, University of Manchester, Manchester, UK

41. Daniel Siewiorek, Katia Sycara, Joseph Giampapa, Ritika Sanghi, Aaron Steinfeld. Interoperability of Future Information Systems briefing to the MURI Project on Adaptive System Interoperability, April 1, 2003

42. Web Services Security (WS-Security) specification. OASIS Standard 1.0.  
[http://www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=wss](http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss)

43. XML Signature Working Group, <http://www.w3.org/Signature/>

44. Grit Denker, Lalana Kagal, Tim Finin, Massimo Paolucci, Naveen Srinivasan and

Katia Sycara, "Security For DAML Web Services: Annotation and Matchmaking" In *Proceedings of the Second International Semantic Web Conference (ISWC 2003)*, Sandial Island, FL, USA, October 2003, pp 335-350.

45. The Ontology Web Language for Services (OWL-S)  
<http://www.daml.org/services/owl-s/1.0/>

46. Web Service Policy specification. May 28, 2003  
<http://www-106.ibm.com/developerworks/library/ws-polfram/>

47. Uszok, A., Bradshaw, J. M., Hayes, P., Jeffers, R., Johnson, M., Kulkarni, S., Breedy, M. R., Lott, J., & Bunch, L. (2003). DAML reality check: A case study of KAoS domain and policy services. Submitted to the International Semantic Web Conference (ISWC 03). Sanibel Island, Florida.

48. Shehory, O. and Sycara, K. "The Retsina Communicator". In *Proceedings of Autonomous Agents and Multi-Agent Systems*, 2000.

49. Finin, T., Labrou, Y. and Mayfield, J., "KQML as an agent communication language" in *Software Agents*, Jeff Bradshaw (Ed.), MIT Press, Cambridge, (1997).

50. Fikes, Richard, Jessica Jenkins, and Gleb Frank. "JTP: A System Architecture and Component Library for Hybrid Reasoning." *Proceedings of the Seventh World Multiconference on Systemics, Cybernetics, and Informatics*. Orlando, Florida, USA. July 27 - 30, 2003. <http://www.ksl.stanford.edu/software/JTP/>

51. Volker Haarslev, Ralf Möller. Description of the RACER System and its Applications. *Proceedings International Workshop on Description Logics (DL-2001)*, Stanford, USA, 1.-3. August 2001

52. HP Labs Semantic Web Research. Jena Semantic Web Framework 2.0.  
<http://jena.sourceforge.net/documentation.html>