SYSTEMS INTEGRATION IN THE FORESTRY DOMAIN

by

RAJESH K. KOMMINENI

(Under the direction of Dr. Walter D. Potter)

## ABSTRACT

The change in the user requirements over the years, has underscored the need for the integration of multiple information sources into a single viable source. In Information technology, legacy systems and data are those that have been inherited from languages, platforms and techniques earlier than the current technology. Integration of such heterogeneous applications is a difficult task. This thesis proposes a framework called Open Intelligent Information System for Forestry Domain (OIISFD) for integrating such applications in the forestry domain. OIISFD contains three components, namely, the inference component, the knowledge component and the legacy component. The inference component contains application description knowledge ("what does the application do"), domain specific knowledge and a reasoning module for handling user queries. The knowledge component contains application specific knowledge ("how to run the application"), input/output criteria and special vocabulary associated with the application. The legacy component contains legacy applications with their wrappers. The entire framework is developed on COM/DCOM platform, with XML (Schema) being used for knowledge representation and for communication mechanisms between the different components.

INDEX WORDS: Forestry Information Systems, Systems Integration, Intelligent Information systems, XML Schemas and Embeddable Prolog.

# SYSTEM INTEGRATION IN THE FORESTRY DOMAIN

by

Rajesh K. Kommineni

B.E., MNREC, India, 1998

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2002

SYSTEM INTEGRATION IN FORESTRY DOMAIN

by

RAJESH K. KOMMINENI

B.E., MNREC, India, 1998

Approved:

Major Professor: Dr. Walter D. Potter

Committee: Dr. Suchi Bhandarkar

Dr. Eileen Kraemer

Electronic Version Approved:

Gordhan L. Patel

Dean of the Graduate School

The University of Georgia

May 2002

## DEDICATION

I dedicate this thesis to my parents.

## ACKNOWLEDGEMENTS

I would like to express my sincerest gratitude to Dr. Walter D. Potter, my major professor, for his constant support and encouragement and invaluable advice throughout this challenging thesis project. I also wish to thank my other committee members, Dr. Suchendra Bhandarkar and Dr. Eileen Kramer, for their support and advice. My special thanks to Priyanka P. Brahmachary for putting up with me throughout this challenging period.

## TABLE OF CONTENTS

Page

# CHAPTER 1

# INTRODUCTION

The Internet has revolutionized the entire concept of information. In the past, information was to a large extent "stationary" in the sense that its reusability was limited. Excellent progress in interconnection afforded by the internet, web and distributed computing infrastructures has led to an easy access to a large number of independently created and managed information resources of broad variety. This in turn has resulted in dramatic changes in the design of information systems. Current information systems are designed such that they can be scalable and reusable. The trend is towards development of small, individual, non-monolithic units that can be embedded in the web of information technology. In the past, applications were developed without much foresight of their reusability and scalability. These applications can be referred to as legacy applications as they are rigid and not amenable to present day changes. In information technology, legacy applications and data are those that have been inherited from languages, platforms, and techniques earlier than current technology. Integration of these legacy applications into current systems is a very difficult task.

Over the years computer user's requirements have changed. He/she no longer wants to gather information from multiple sources but prefers a single source. For example, one of the requirements for NED-1 was to develop a computer program that would combine all the previously independently produced growth and yield models

developed by scientists within the station (Twery, 1997). This change in the requirements led to a need for integration of multiple information resources into a single resource.

There are various problems associated with the integration of multiple information sources into a single source. In the forestry domain, briefly, the problems are: heterogeneity of applications, absence of programmatic interfaces into applications, rigid input and output formats, lack of reusability, scalability of proposed architectures, expensive knowledge acquisition process and maintainability and little or no use of domain expert's knowledge in problem solving. These problems hinder the integration of multiple applications.

This thesis proposes an architecture for overcoming the above problems. We propose to use a knowledge representation language that is easy to author and to interpret (by domain experts themselves). This strategy will lessen the expensive process of knowledge acquisition and maintainability of the system. We also propose to use a common markup language to describe the output and input file formats of the applications. This helps in easing the interoperability process between various applications. The framework for this architecture encodes the knowledge of applications in such a way that it can be reusable and contain domain expert's knowledge that is very useful in decision-making process.

To test various proposed ideas in the thesis, a framework has been developed with two interacting applications. The knowledge of these applications is encoded in a markup language and has been used for decision-making. Interfaces have been developed to translate rigid input/output file formats into a common language and thereby reduce the difficulty in conversion of the data between applications. The entire framework is

developed on a platform that is guaranteed to handle the various issues involved with heterogeneous nature of the applications. This implementation includes a sophisticated intelligent module that is extensible and aids in decision-making.

The process of integrating an application into the framework is as follows:

- Describe the various knowledge types ("what" does the application do and if necessary, "how" to run the application, special vocabulary associated with the application).
- Describe the input and output formats of the application in a markup language.
- Develop a wrapper that provides an interface into the application in the format proposed by the framework.
- Develop the graphical user interface screens that might help the user.

Once the above steps are fulfilled for an application, this application can be integrated into the framework.

Even though this framework has been implemented only in the forestry domain, the proposed architecture can be applied to any domain with relative ease.

### 1.1 Systems integration in forestry domain

Decision Support Systems (DSSs) are interactive computer-based systems intended to help decision makers utilize data and models to identify and solve problems and make decisions. The "system must aid a decision maker in solving unprogrammed, unstructured (or "semi-structured") problems...the system must possess an interactive query facility, with a query language that ...is ...easy to learn and use (Bonczek,

Holsapple & Whinston, 1981; p. 19)". They are designed to assist managers in their decision processes in semi-structured (or unstructured) tasks. They support, rather than replace, managerial judgment; and their objective is to improve the effectiveness of the decisions, not the efficiency with which decisions are being made.

Several Decision Support Systems have been developed over the years for providing useful scientific data for the management of forest eco-systems. A typical decision support system for forest eco-system management contains a user interface, database, geographical information system (GIS), a knowledge base, simulation and optimization models, help/hypertext management, data visualization and decision methods (Rauscher 1999). There have been more than thirty Forest Ecosystem Management (FEM)-DSSs developed for use in the United States with different FEM-DSSs supporting different parts of the eco-system management process (Mowrer et al. 1997). There is not a single FEM-DSS that is competent enough to deal with issues concerning both ecological and management interaction in forest eco-system management (Liu 1998). An ideal FEM system therefore requires all available DSSs (Potter et al. 1998). This can be accomplished by integration and enhancement of existing systems rather than developing a totally new system.

The two main issues that concern the integration of forestry applications are:

- **Heterogeneity of the application**.

  Most FEM-DSSs were developed independently of one another with little or no coordination between most of them, leading to a heterogeneous development environment (Liu 1998). Most of them were developed to run on one machine and if

they had to be integrated into a distributed system, their heterogeneous platforms played an important role.

- **Legacy Systems**.

  Most of the FEM systems are legacy systems, written in languages like FORTRAN and have a command line interface. These legacy systems are not designed for a distributed technology and they are not amenable for reuse and extension. They are the stereotypic, monolithic, closed systems that are expensive to maintain (Liu 1998). Their integration into current FEM-DSS is not a trivial issue and requires much effort.

(Liu, 1998) has identified several features for more effective decision support for forest ecosystem management:

- Various existing decision support system modules have to be integrated so that they can interoperate.
- This interoperable framework should offer an interface standard that promotes communication between component modules. It should also be possible to integrate newly developed modules and refine current ones over time, if necessary.
- This framework should provide language interoperability, platform independence, module communication mechanisms, object reusability and interfacing standards for dealing with legacy systems.

This thesis takes into consideration the various features of Liu's model for effective decision support for forest ecosystem management.

The thesis mainly focuses on two things:

- The architecture for integration system and communication mechanisms between different components of the architecture.
- Representation of Knowledge and Data of legacy systems and development of wrappers (interface standards) for the legacy systems with prime importance to their reusability.

It uses the Forest Vegetation Simulator (FVS) as a testing ground for the proposed ideas.

This thesis is organized as follows:

The second chapter examines the integration system called the Intelligent Information System (IIS) (Somasekar, 1999), which has been proposed for the FEM system. It identifies the inherent disadvantages of the architecture of IIS. This chapter describes the guidelines or features that an individual module in FEM should possess, using an open systems approach. It examines the architectures of current generation information systems. This chapter briefly describes two architectures for information systems:

- Mediator based information architecture
- Information brokering architecture

This chapter also describes the knowledge engineering principles behind the current generation knowledge based systems.

This thesis uses the Extensible Markup Language (XML) as a means for communication between the different modules of OIISFD. The third chapter contains

background of XML and its role in the proposed architecture. It also analyzes the features in XML that can be used to represent knowledge.

The fourth chapter contains the proposed architecture for FEM, called Open Intelligent Information System for Forestry Domain (OIISFD). This chapter explains how platform independence and language neutrality can be achieved using the Distributed Component Object Model (DCOM) as a platform for OIISFD. The three main components of OIISFD are:

- Knowledge Component
- Legacy Component
- Inference Component

This chapter also contains implementation details of the above components.

The fifth and sixth chapters contain results and conclusions respectively. The fifth chapter describes various queries that are handled by the proposed architecture and screen shots of the implementation.

# CHAPTER 2

# BACK GROUND AND RELATED WORK

Much research has been done on the architectures for information systems (Sheth, 1998). These same architectures can also be applied for the development of FEM-DSSs. NED-1 (Twery et al., 1997), is a goal driven, full service ecosystem management system developed for the eastern US. It provides baseline analyses of conditions needed for multiple resource management areas that are composed of a set of forest stands. The architecture is essentially a client-server architecture. It is a standalone system and hence, does not have distributed processing capability. More over, it does not address the issues related to legacy applications. It is not platform or language independent and it is difficult to add more forestry applications to it (Liu, 1998).

In order to circumvent the above difficulties, Somasekar (1999) proposed the Intelligent Information System (IIS) model. This system contains three modules: legacy applications and wrappers, client interface module and an Intelligent Information Module (IIM). The IIM contains production rules, which are triggered according to the user's request. It then runs the appropriate application with the help of wrappers and returns the results to the user.

The setbacks of this model are as follows:

Intelligence (in the form of production rules) of the integrated system is centralized. Ownership of these production rules is a problem. This results in a difficulty to maintain

the system. Addition or deletion of any of the production rules lead to a temporary dysfunction of the system. The wrappers built for the legacy applications are custom wrappers that have to be changed when integrated into another system. Integrating a new system into the above system is very difficult. The knowledge and effort invested in developing these custom wrappers is not reusable. This custom wrapper needs to be modified if the current legacy system wants to interact with another system. Hence, it is essential that we have n-n wrappers for interaction between n applications. Also, there are no standards used for communication between the different modules of the IIS model.

We get a few pointers from the disadvantages of the above model. Primarily, a legacy system should be made "open". The framework should be developed keeping in mind its future extensibility. Standards should be used when ever possible. Reusability should be given prime importance.

### 2.1 Open Systems Approach

An open system (SEI, 1986) can be defined as a system, with characteristics that comply with specified, publicly maintained, readily available standards and that therefore can be connected to other systems that comply with these same standards.

An open system (SEI, 1990) can also be defined as

A collection of interacting software, hardware, and human components

- Designed to satisfy stated needs
- With interface specifications of its components that are
    - Fully defined
    - Available to the public

- Maintained according to group consensus

- In which the implementations of the components conform to the interface specifications.

An open systems approach for systems integration can be described in the following steps

- Adopt standard interfaces
- Acquire components
- Integrate components
- Use & support the integrated systems

This approach when compared to the traditional approach differs in the first two steps. In a traditional approach, instead of adopting standard interfaces, unique interfaces are defined. Also, instead of acquiring components, unique components are developed. The differences between the two approaches are outlined in figure 2.1

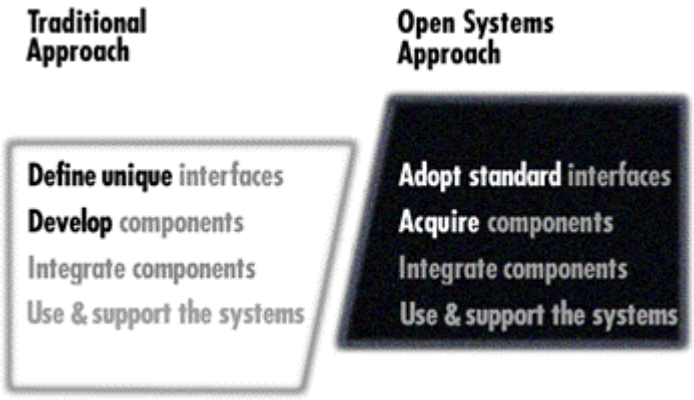


Figure 2.1 Differences between Traditional and Open Systems approach (SEI, 1990)

## 2.2 Current trends in information systems architecture

Sheth et al (Sheth, 1998) have described the changing focus towards developing information system architectures. According to them, the evolution of information systems, in the context of interoperability, has occurred in three generations. Generation I covers the period up to roughly 1985, Generation II covers the decade through 1995, and Generation III covers a period yet to be bound from 1996. The various features evolved during the different generations are as follows:

1. Software and information system architecture

    - Generation I

      Terminal access, point-to-point; mainframes and minicomputers with remote access, client-server

    - Generation II

      Client-server;

    - Generation III

      Network, distributed and mobile

2. Key human roles in supporting interoperability

    - Generation I

      Data (base) administrators or experienced users, software developer written access programs

    - Generation II

      Software developers to generate wrappers and mediators (with some toolkits) involving data level issues

    - Generation III

      Domain experts for developing ontologies and for generating information correlations

An ontology can be defined as a specific vocabulary and relationships used to describe certain aspects of reality, and a set of explicit assumptions regarding the intended meaning of the vocabulary of words (Gruber 1991; Guarino 1998). Ontologies are used to map domain specific information to a set of rules and axioms.

3. Dominant interoperability architecture

- Generation I

  Multidatabases or federated databases

  During this generation, there was more emphasis on achieving system interoperability, in particular, by addressing the heterogeneity due to differences in DataBase Management Systems (DBMSs). Correspondingly, the emphasis was on data management and data (as opposed to information or knowledge). More information on the work that took place during this generation can be found in the context of multidatabases (Litwin et al. 1982) and federated database systems (Heimbigner and McLeod 1985; Sheth and Larson 1990).

- Generation II

  Federated information systems, mediator architectures

  According to Sheth et al (Sheth, 1998), the key trends and achievements of this generation are as follows:
  - Technology for dealing with heterogeneity of systems, data, and representational levels;
  - Support for a broader variety of data - not just structured databases, but also text, semi-structured, and unstructured (including image and video) data

  Some of the systems in this generation adapted the Federated DataBase Systems (FDBS) (Heimbigner and McLeod 1985; Sheth and Larson 1990) architecture to federated information systems architecture. However, the mediator architectures (Wiederhold 1992) were clearly the dominant ones, involving wrappers for encapsulating heterogeneous information sources to provide a more uniform interface to the rest of the world.
- Generation III

  Mediator architectures, information brokering architectures

  These architectures are described in detail in the later sections of this chapter.

As mentioned above the dominant third generation information architectures are based on mediators and information brokers.

### 2.2.1 Mediator-based Information Architecture

In mediator-based information architectures (Wiederhold, 1992), the information creators and providers are decoupled from information users. Here, information creators and providers are heterogeneous information sources. The main difference between creators and providers is the way they own information. Information creators, when requested, obtain information from a remote source that is not known locally. They contain support for remote communication. Information providers are static information sources like databases. The mediator-based information architecture is primarily composed of three distinct layers that form a pyramid. The information providers and creators form the base of the pyramid (Fig 2.2). The mediators fall in the center of the pyramid while the information users form the apex as the application layer. Mediators bridge the gap between the information creators/providers and the information users.

This whole architecture makes extensive use of wrappers. A wrapper encapsulates a single data source to make it usable in a more convenient fashion than the original unwrapped source. The user requests information from the mediator who queries the wrapper to retrieve the information. The wrappers convert the mediator's query to the source level queries. The wrapper translates the language for the mediator and makes it possible for the mediator to deal with only one language.

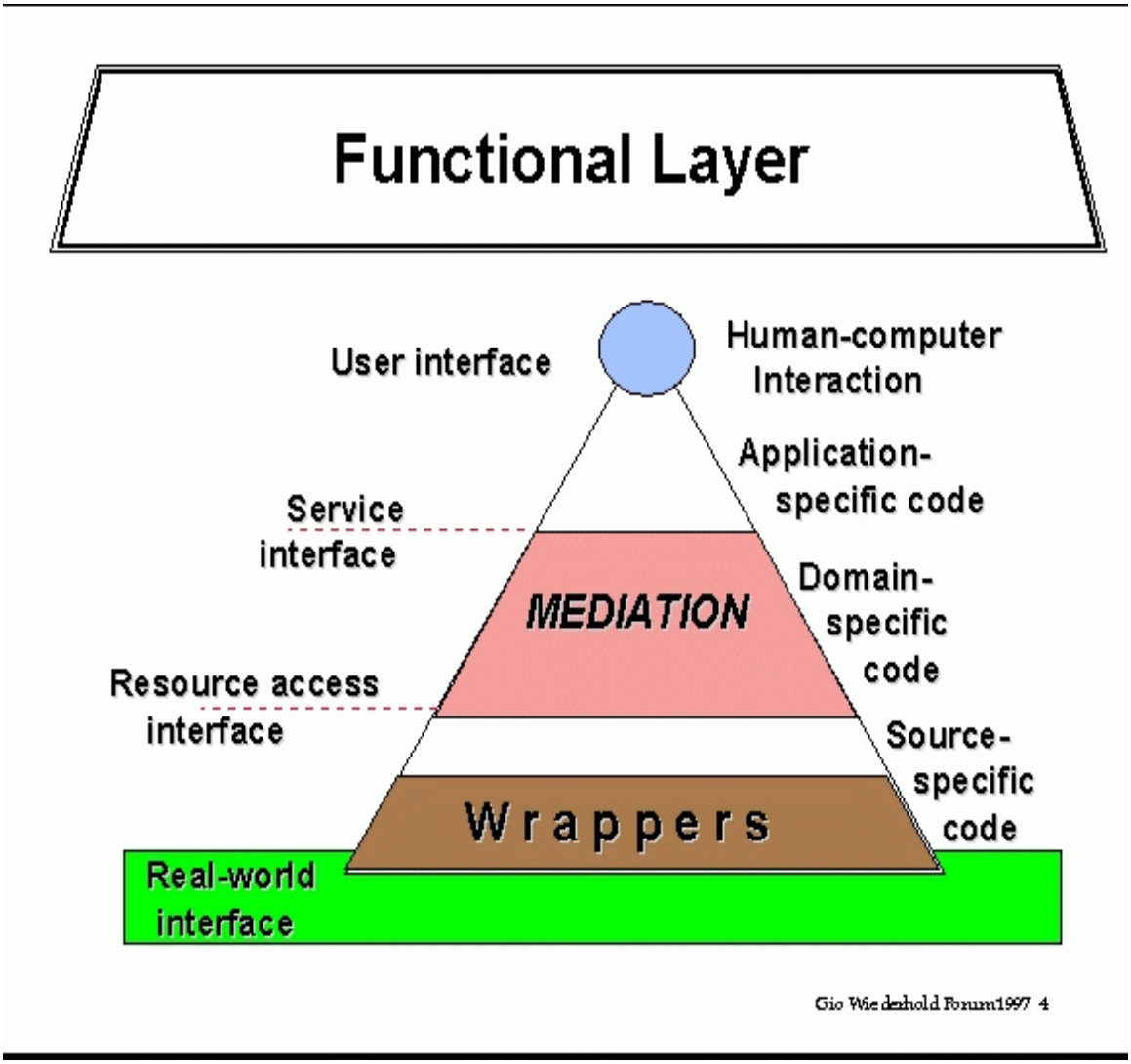


Figure 2.2 Functional Layer of Mediator based information system (Wiederhold, 1997)

### 2.2.2 Information brokering Architecture

This architecture is heavily based on the use of agent technology (Martin, 1997). An information brokering system provides coordinated access to a heterogeneous collection of structured and semi-structured information sources (Sheth, 1998). In this architecture, a broker agent accepts the requests from the user. The broker agent then with the help of a facilitator agent forwards the requests to the source agent(s). After retrieving the information from the source agent(s), the broker agent integrates the requested information and gives the result to the user. The source agent(s) that is in charge of the application (information source) possesses a high degree of independence and autonomy. It can be created, administered and enhanced independently. Figure 2.3 depicts ideal information brokering systems architecture. In this figure BQ (Broker Query) and BR (Broker Response) refer to items expressed in the broker schema whereas SQ (Source Query) and SR (Source Response) refer to items expressed in the source schema. RDB abbreviates Relation Data Base and KB abbreviates Knowledge base. Schema, here, is essentially the same as a relational data base schema. From developer's point of view, equivalently, it may be thought of as a collection of Prolog Predicates.

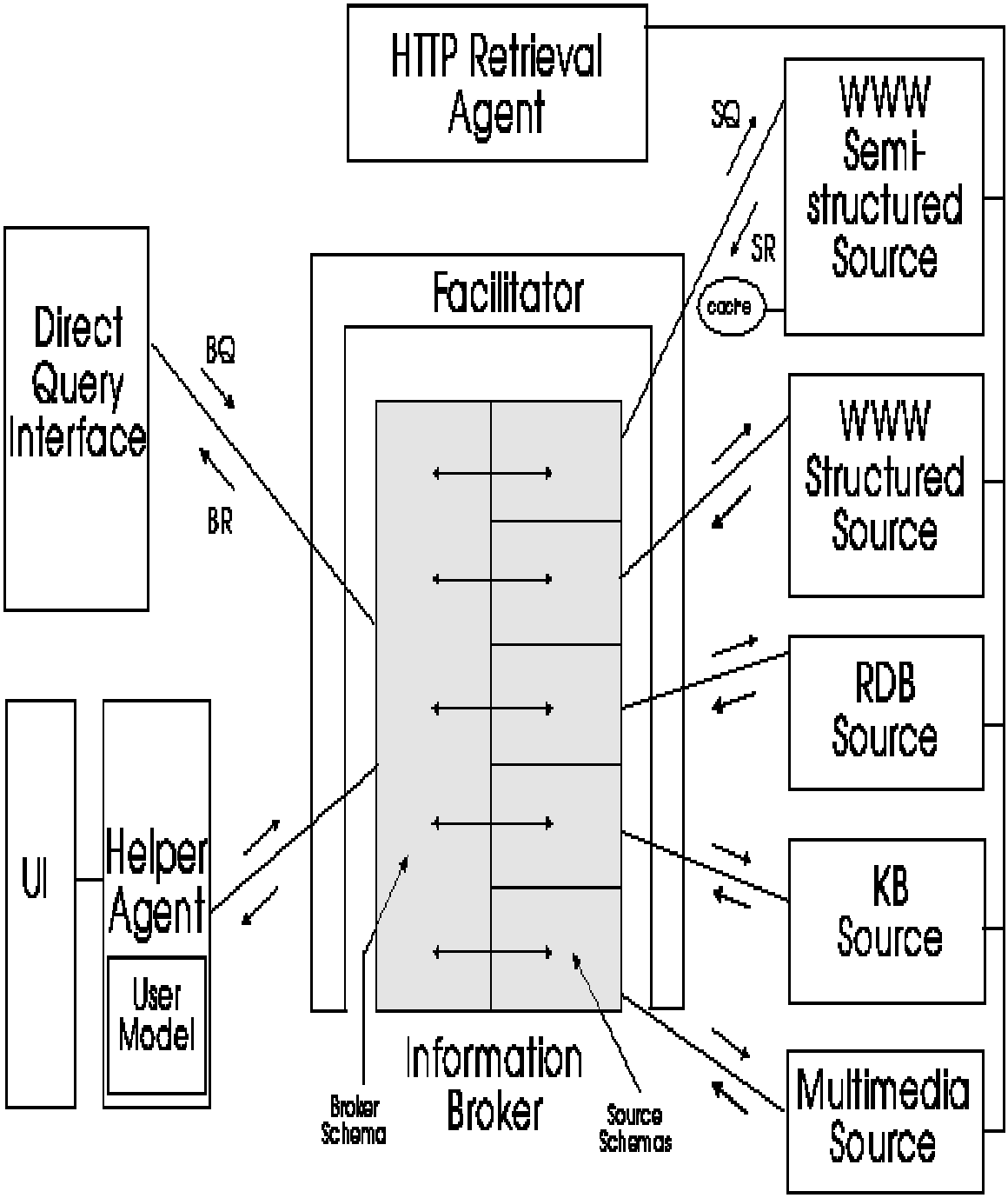


Figure 2.3: Information brokering system architecture

## 2.3 Current trends in Knowledge Base Systems

A Knowledge Base System (KBS) is a system that represents an application containing a significant amount of real knowledge, and has been designed, implemented and possibly maintained with due regard for the structure of the data, information and knowledge (Durkin, 1994).

In the early 1980s, the development of KBS was seen as a process of *transferring* human knowledge to an implemented knowledge base (Studer etal.1999). The knowledge in these "first generation" KBSs is in the form of associational rule-of-thumb the mapped from observable features to conclusions. Typically, knowledge was implemented in some type of production rules that were executed by an associated rule interpreter. These systems solved problems by chaining rules together, either forward (from premises to conclusions) or back ward (from goal to initial conditions). Thus, early KBSs typically had a simple control structure and uniform representation of knowledge, in the form for associational production rules. The knowledge was typically represented at a single level of abstraction and implicitly combined knowledge about "how" to perform a task, "what" is in the domain, and "why" things work. While it was initially thought that this would make systems fairly easy to develop, in fact it leads to several problems related to knowledge acquisition, explanation, brittleness and maintainability. Because of these problems, this transfer approach was only feasible for the development of small prototypical systems. This approach failed to produce large, reliable and maintainable knowledge base (David etal. 1993).

The above problems resulted in a paradigm shift from the *transfer* approach to the *modeling* approach (Studer etal. 1999). This paradigm shift was inspired by Newel's

knowledge level notion (Newell, 1982). The second generation KBSs can be distinguished from the earlier KBSs in the following ways:

- The knowledge is modeled independently from its implementation
- Different knowledge formalisms are used for representing various knowledge types.
- The control knowledge ("how" to perform a task) is explicit.

The control knowledge mentioned above is the rationale behind the use of *Problem - Solving methods* (PSMs) (Fensal etal., 1996).

This thesis uses two types of knowledge (and their corresponding PSMs) for decision-making. The following sections provide their necessary background.

### 2.3.1 Associational Knowledge and Production Systems

Production systems are composed of three components (Peter, 1998). These are:

- The rule set
- A working storage area which contains the current state of the system
- An inference engine that knows how to apply the rules

The rules are of the form:

left hand side (LHS) ==> right hand side (RHS).

The LHS is a collection of conditions that must be matched in working storage for the rule to be executed. The RHS contains the actions to be taken if the LHS conditions are met.

The execution cycle is:

1. Select a rule whose left hand side conditions match the current state as stored in the working storage.

2. Execute the right hand side of that rule, thus somehow changing the current state.
3. Repeat until there are no rules that apply.

Production systems can be implemented in any programming language such as C, C++, Java or Prolog. Prolog is usually preferred over other programming languages (Peter, 1998). The expressiveness of Prolog is due to three major features of the language: rule-based programming, built-in pattern matching, and backtracking execution. The rule-based programming allows the program code to be written in a form that is more declarative than procedural. This is made possible by the built-in pattern matching and backtracking that automatically provide for the flow of control in the program. These features together make it possible to elegantly implement many types of production systems.

Goal driven reasoning or backward chaining is an inference technique that uses IF THEN rules to repetitively break a goal into smaller sub-goals that are easier to prove. This is an efficient way to solve problems that can be modeled as "structured selection" problems. That is, the aim of the system is to pick the best choice from many enumerated possibilities. For example, an identification problem falls in this category. Diagnostic systems also fit this model, since the aim of the system is to pick the correct diagnosis. The features of prolog make it an ideal language for goal driven reasoning system.

On the other hand, Data driven reasoning or forward chaining is an inference technique that uses IF THEN rules to deduce a problem solution from initial data. Production systems are usually called forward chaining systems (Elaine etal, 1991). A forward chaining reasoning can be implemented using Prolog. The features of Prolog make it an ideal language to implement rule-based forward chaining inference engine.

### 2.3.1 Ontologies and Frame-logic

An ontology can be defined as a specific vocabulary and relationships used to describe certain aspects of reality, and a set of explicit assumptions regarding the intended meaning of the vocabulary of words (Gruber 1991; Guarino 1998). Ontologies are used to map domain specific information to a set of rules and axioms. One way to represent ontology is to use frames (Karp et al. 1995).

A frame is a collection of attributes (usually called slots) and associated values (and possibly constraints on values) that describe some entity in the world (Rich et al. 1991). An own slot of a frame has associated with it a set of own facets, and each own facet of a slot of a frame has associated with it a set of entities called facet values. Frame systems are built out of collections of frames that each other by relationships. An example relation is *a-kind-of*. This relationship is used for inheritance.

# CHAPTER 3

# XML

This thesis uses XML for two purposes: for communication mechanisms between different components and as a knowledge representation language. The following sections describe the features of XML and the rationale behind using XML as a knowledge representation language.

## 3.1 XML background

eXtensible Markup Language (XML) (XML, 1998), a subset of Standard Generalized Markup Language (SGML), is designed to be the universal data exchange language, with a special status as the next generation language. XML is different from other previous data exchange formats in its two major characteristics: extensibility and self-description. XML is extensible in that any author can extend the tags used in an XML document to meet his/her own needs for the document. XML is self-describing in that the tags in an XML document express the structure of the document and, in most cases, the intention of the author as well.

What makes XML even more attractive for many other purposes, and to a certain degree more self-describing, is the existence of Document Type Definition (DTDs) and XML Schemas (XML Schemas, 2000). XML Schemas (and DTDs) state the structure of an XML document and specify the allowed element types and the characteristics of each

element type, and also its allowed attributes. Although a XML Schema is optional for an XML document, its existence is valuable for many practical purposes.

Both XML Schemas and DTDs can be used to represent the knowledge. This thesis uses XML Schemas to represent the knowledge. The following section explains the rationale behind using XML schemas (over DTDs) and its features that are useful for knowledge representation.

### 3.2 XML Schemas and its Advantages over DTDs

Even though XML schemas and DTDs both are used to do the validation of the document, Schemas are preferred over DTDs in this thesis for the following reasons:

- DTDs are Difficult to Write and Understand

  DTDs use a syntax other than XML, namely Extended Backus Naur Form (EBNF), and many people find it difficult to read and use. The proposed XML schemas, however, actually use XML to describe the languages they define, removing the difficulty of learning EBNF before learning to read and write them.

- Programmatic Processing Of Metadata Is Difficult

  The use of EBNF also makes the automatic processing of metadata in DTDs difficult. It is not possible to inquire into the DTD from a program using the Document Object Model (DOM). The DOM makes no provision for gaining access to vocabulary's metadata written in EBNF. Hence, metadata information associated with the document won't be available for processing. Since schemas are written using XML syntax, this problem of programmatic processing of metadata can be overcome.

- DTDs Are Not Extensible And Do Not Provide Support For Namespaces

  All the rules in a vocabulary must exist in the DTD. One can't borrow from other sources without creating external entities. Creating and maintaining subsets of markup declarations isn't as flexible as simply referring to an existing definition. As all rules in a vocabulary must exist in the DTD, one cannot mix namespaces.

  While one can use a namespace to introduce an element type into a document, he/she cannot use a namespace to refer to an element declaration in a DTD. If a namespace is used, all elements from the namespace must be declared in the DTD.

- DTDs Do Not Support Data Types

  DTDs offer few data types other than text. This is a serious shortcoming when using XML in a knowledge-based application.

- DTDs Do Not Support Inheritance

  With DTDs there is no way of expressing inheritance. So if we have an entity called plot, there is no way one can say that plot entity is a subclass of forest entity.

Due to the above reasons XML schemas are used in the proposed architecture.

To understand how well XML is suited for knowledge representation purposes, an examination of the semantic expressiveness is useful here.

### 3.3 XML as a Knowledge Representation Language

*Semantic Expressiveness*

The expressiveness of XML can be examined by studying XML Schemas (XML Schemas, 2001). XML Schemas include four types of declarations: *element declaration, attribute declaration, entity declaration and notation declaration*. An element describes a minimal section of XML texts, and the combination of all the elements described in XML Schemas form the contents of XML. Each XML document has exactly one root element, which indicates what kind of document it is and encompasses all other elements in the document. Each element has an optional attribute list, which further describes the element. The basic components of an attribute are attribute name, attribute type, data type and default value.

The following features are present in XML Schemas that can be used for knowledge representation:

**Enumeration**

For any XML element or attribute, it is necessary to be able to enumerate the possible values for it. In knowledge representation languages, it is necessary sometimes to enumerate all instances of a class or all possible values for a slot. *XML Schemas fully supports enumeration and has a mechanism for specifying a default value for an element or attribute.* The fact that SouthEastern element may contain elements of type GeorgiaState or FloridaState can be represented using XML Schemas as follows:

```
<ElementType name=”SouthEastern”>
        <complexType>

            <element name=”State”>
                <complexType>
                    <choice>
```

```
                    <element name="GeorgiaState" type="Georgia" default/>
                    <element name="FloridaState" type="Florida"/>
                    </choice>
                </complexType>
            </element>
            </complexType>
        </ElementType>
```

Here, by using key word "choice", all possible elements can be enumerated. "default" keyword is used to make one of the elements to be default.

**Optional**

An XML Schema has the capability to indicate whether some attribute or element is optional. This is necessary for knowledge representation purposes. Sometimes additional properties of an object might help for inference purposes with out making the classification of the object difficult. Such additional properties can be made optional. For example, we need to indicate whether some aspect of a class definition is optional. *XML Schemas also have various other symbols to express "one or more occurrences" and "zero or one occurrence" etc*.

SpeciesName and SpeciesCode are two elements that are usually present in data files of Forestry Applications. SpeciesCode element is necessary for calculating different properties of a tree while SpeciesName element is present for informational purposes. This fact that SpeciesCode element should always present and SpeciesName element presence is optional can be represented using XMLSchemas as follows:

```
<ElementType name="DataFile">
    <complexType>
        <element name="SpeciesCode" minOccurs="1" type="string"/>
        <element name="SpeciesName" minOccurs="0" type="string"/>
    </complexType>
</ElementType>
```

In the above example, the keyword "minOccurs" is used to specify whether the element is optional or not by mentioning its count.

**Value Restriction**

XML Schemas supports value restriction. We can specify the possible values of an element. In other words, *XML Schemas can be used to indicate what class the value type of an element or attribute*.

The fact that the value of the element quantity should not exceed 100 can be represented in XML Schemas as follows:

```
<element name=”quantity”>
        <simpletype>
                <restriction base=”positiveInteger”>
                <maxExclusive value=”100”/>
                 </restriction>
        </simpleType>
</element>
```

**More Complicated Data Types**

The structures defined for XML Schemas rely heavily on type definitions. These allow declaring extended types that can be used throughout a schema. They will be used to specify the content and type of elements and attributes. Current XML is used mostly for general-purpose texts, where a few very basic types are enough. However, for knowledge representation and various other purposes, we need the capability to express more complicated data types and mathematic expressions. *XML Schemas can be used to represent complex data types using type definitions but not mathematical expressions*. Additional support can be added at the processing level (of the schemas) to handle mathematical expressions. By using value restriction, value extension and grouping of elements, complicated data types can be defined.

For example, a telephone number contains the format *ddd-dddd.* A new data type called TelephoneNumber can be defined that conforms to the above format as follows:

```
<simpleType name="TelephoneNumber">
```

```
   <restriction base="string">
      <length value="8"/>
      <pattern value="\d{3}-\d{4}"/>
   </restriction>
</simpleType>
```

**Strong Data Typing**

XML documents are semi-structured data, which means that data in an XML document is not as precisely structured as that in a database. However, for knowledge representation as well as many other purposes like for handling communications between different modules, XML optionally support strong data typing. What strong data typing will achieve is to ensure what is purported to be within a pair of tags is truly the type of data the tags say it is. For example, in order to ensure what is extracted out of the <birthrate> </birthdate> tag pair is truly a date type, we want to make sure there is no other random text between the tags. For example,

<date> is 4/8/99</date> is not valid, but

<date> 4/8/99 </date> is valid.

*XML Schemas support strong data typing. With the help of schema support, we can make sure the data between tags is of the type we wanted.*

For example the element money is a currency sign, followed by one or more digits, optionally followed by a period and two digits. This can be represented in XML Schemas as follows:

```
<simpleType name="money">
   <restriction base="string">
      <pattern value="\p{Sc}\p{Nd}+(\.\p{Nd}\p{Nd})?"/>
   </restriction>
</simpleType>
<element name="cost" type="money"/>
```

XML parser will validate the following elements with out any problem.
<cost>$45.99</cost> <cost>¥300</cost>

**Grammar**

Further semantic expressiveness also requires a grammar for XML Schemas. The grammar will specify the allowed formats of what goes between a pair of tags. The element type of an element can be of various forms. XML Schemas, with the help of enumeration property, can specify the allowed formats. For example, there are numerous formats for expressing <date>, such as "Month/Date/Year" or "Date/Month/Year". Knowing what formats are allowed in a certain XML document not only enables XML authors to correctly write XML documents, but also helps different software applications to correctly interpret and extract XML documents. *Currently, XML Schemas have extensive built-in data types*. More complex data types can be derived using these built-in data types.

**Validation**

Currently, validated XML documents refer to documents that conform to an XML Schema. Validation means that the document contains the allowed elements and does not violate the specified structure of the tags. *However, the validation validates the content of the document to a certain extent only*.

**Schema Interaction**

XML Schemas are specific entities that specify the structures of different XML documents. However, *the method for inter-referencing between XML Schemas is achieved through XML names spaces*. For example, the citizenship of any person needs to be of the "country" type. In any Schema that includes the <citizenship> tag, it does not have to define the country class. Through namespaces by referring the <citizenship> that

it belongs to <country> Schema, we can develop interaction between two different Schemas.

For attaining the above interaction we have to first declare the Namespace country using import statement.

<import namespace="CountrySchema" schemaLocation="Country.xsd"/>

The elements in the CountrySchema can be accessed as follows:

<element name=”citizenship” type=”CountrySchema:Country”/>

**Reusability**

Using namespaces, XML Schemas provide reusability. It also avoids the problems of *ambiguities* and *name collisions* of tags.

The features that are partially supported in XML Schemas that can be used for knowledge representation purposes are:

**Representing Axioms**

An important power of knowledge representation languages is that they support axiom representation. Current XML Schemas (XML Schemas, 2001) specification can express the attributes/element types for different attributes/elements, but does not provide methods for expressing the relationships between elements and/or between attributes. *The relationships that can be specified using XML Schemas are Inheritance and Composition.* More complex relationships can be achieved by custom validation of the documents.

With all the semantic aspects mentioned above, we have an XML Schema that provides reasoning support and is expressive in many dimensions. The semantics and expressive power of the XML Schemas is more in line with that of a general knowledge

representation language. *The positive effect that comes out of this is the same language for representing knowledge as well as data while other languages or representations fail to achieve the same*. (Kent, R 1999) has shown that DTDs are as expressive as ontologies. Since XML Schemas are more powerful than DTDs, they can be used to represent the knowledge. Michel etal. have investigated the relation between ontologies and XML Schemas and found that XML Schemas are comparable to Ontology Interface Language (OIL) (Klein etal.) XML schemas specification is widely accepted and standardized. Any application that conforms to this specification can process the document with out any modifications. This is the main advantage of representation of knowledge in XML. XML presents tremendous opportunities for the knowledge representation community. One intuition is to see an Internet of XML documents as a vast source of knowledge. Since XML is inherently structured, it is easy to extract. An XML document can be directly parsed into a knowledge base for more sophisticated inference mechanisms (Frank, 1999).

**3.4 Advantages of XML in systems integration**

Application experts need a language that is easy to use for authoring knowledge schemas. XML is an ideal language for representing the knowledge component for the following reasons:

- It is human readable. So, it can be readily understood and the knowledge schemas can be edited easily.
- It is easy to author.

- It is universally standardized.

### 3.5 XML Namespaces

XML namespaces are the solutions to the problems of ambiguity and name collisions. According to the World Wide Web Consortium's (W3C) recommendation 'Namespaces in XML' (14 January 1999), a namespace is

*… A collection of names identified by Uniform Resource Identifier (URI), that are used in XML documents as element types and attribute names.*

With the help of XML namespaces, one can standardize the definitions of various entities. He/she can also standardize the relationship and interactions between the different entities. Each namespace is identified with a URI. So, when someone refers to an element or entity with a URI, he/she can clearly identify its context (definition, structure).

Namespaces will also help XML vocabulary designers to break complex problems into smaller pieces and mix multiple vocabularies as needed to fully describe a problem in a single XML document.

#### 3.5.1 Declaring A Namespace

A namespace is declared by using the keyword **xmlns**. The value of the attribute is the URI that uniquely defines the namespace in use. The URI is often an URL pointing to a schema but it doesn't have to be. A URI, managed in such a way as to uniquely differentiate the namespace, is sufficient. An example of a namespace is described as follows:

xmlns = http://www.ai.uga.edu/forestry/forest.xml

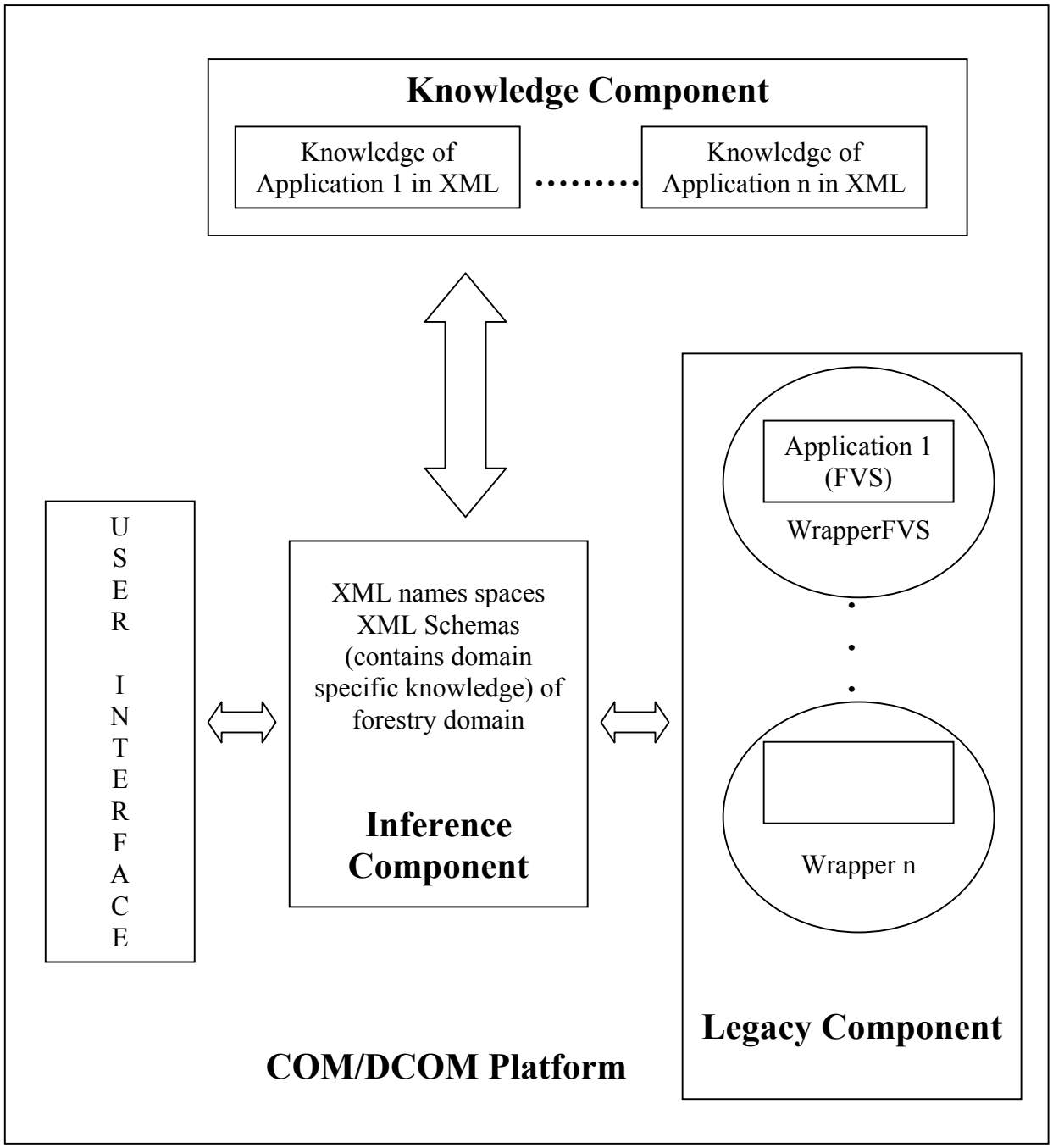


Figure 4.1 Architecture of Open Intelligent Information System in Forestry Domain

# CHAPTER 4

# DESIGN AND IMPLEMENTATION

An integrating architecture forms a general framework for information exchange and co-ordination amongst a variety of software systems. The primary aim of this thesis is to design a general-purpose framework for integrating existing and future heterogeneous applications in the forestry domain. This design takes into account the specific challenges posed by the forestry domain (Liu, 1998) and is also amenable to other domains.

## 4.1 Design Goals

An ideal FEM-DSS should be one that has most or all of the legacy systems working as a whole across forest ecosystem process. Most of the monolithic legacy FEM-DSSs works in isolation with only one or few aspects of the entire forest ecosystem process. There are no standards used in development of the FEM-DSSs. Therefore, the integration of these independently developed software solutions is desirable for building effective FEM-DSSs as software development is growing toward constructing systems from pre-existing components (Jorgensen et al. 1996.). Hence an interoperable architecture should be open, generic and general purpose. It should be a communication channel between a number of software components and independent developers. To attain this goal, the system architecture must satisfy many requirements. The design

should be language independent such that it should be able to integrate applications developed with different languages.

The design must have extensibility such that it can be used over a period of time and not become obsolete. This will enable integration of new components into the existing system with relative ease. This ensures the growth and evolution of the existing system into a new more capable system over time.

The design should provide a common platform for communication and interaction between the components of the system.

The design should be able to handle the peculiarities of the legacy systems. For example, many legacy applications are not object-oriented, and data and functionality of one application may not be readily available to other applications, even if the applications are implemented with the same programming language and run on the same machine. The design should provide guidelines in dealing with the integration of legacy systems.

Considerable amounts of effort and time will be spent in the integration of the legacy systems into a DSS. The design strategy should support the notion that instead of a legacy system being integrated into a particular DSS, it should be feasible for the legacy system to be integrated into any DSS with relative ease.

The design should also propose standards for the forestry domain that will alleviate problems of knowledge disparity.

The design should be realistically applicable when implemented.

## 4.2 Open Intelligent Information System for Forestry Domain (OIISFD)

In accordance with the above requirements, we propose a model for systems integration in the forestry domain. In this section, we describe a detailed design, Open Intelligent Information System for Forestry Domain (OIISFD) that provides a general framework for integration (fig 3.1). The basic aim of the system is to provide information to the user. The OIIFSD may perform various operations such as simple data retrieval, simulation run, complex calculations or a combination of these in order to provide the user with the information. It will make use of one or more components of the system. It is mandatory that the OIISFD interprets the users needs correctly and then undertakes the appropriate action. This entire process of information retrieval should be hidden from the user. The OIISFD comprises of three interacting components

### 4.2.1 Components of OIISFD

The OIISFD contains three components:

- Inference component
- Knowledge component
- Legacy component

Each of these components will be explained in detail in later sections. This architecture is a blend of mediation (Wiederhold, 1992) and information brokering architectures (Martin, 1997). Here, the legacy component uses wrapper technology and acts as a mediator between the legacy application and inference component. The inference component acts as an information broker by processing the user's query and issuing the action statements to the legacy component.

A typical OIISFD flow is outlined below:

The user interface sends a query and data to the inference component. The inference component based on the user's query, infers the model(s) that has to be run, using application specific and domain specific schemas. Then the input data (provided by the user) is preprocessed (in XML) using the application's input file schema. The inference component invokes the application's wrapper. The wrapper first converts the input XML data into raw application data, launches the application, runs the application, parses the output data into XML and then returns it to the inference component. If there are no models to be run, it returns the data to the user. If there are models, it will go through the above process once again.

To meet the design criteria, the framework should be language and platform independent and have inherent mechanisms for communication. COM/DCOM was selected as a platform on which the framework is implemented. This platform meets the design guidelines. The salient features of COM/DCOM are described in the next section.

### 4.2.2 COM/DCOM platform

COM or Component Object Model (COM) is Microsoft's binary standard that defines the interactions between objects. Distributed COM (Microsoft, 1996.) (DCOM) is an extension of the COM and has been called "COM with a longer wire". It allows COM objects to be activated on remote machines. To a client of a DCOM object, the location of the object (local or remote) doesn't matter. It sees a remote object as if it were a local object. Since, DCOM is simply distributed COM, both the terms COM and DCOM are often used interchangeably.

DCOM is a binary specification to integrate components. A component is a reusable piece of software in a binary form that can be plugged into other components. DCOM defines a binary interoperability standard rather than a language-based standard.

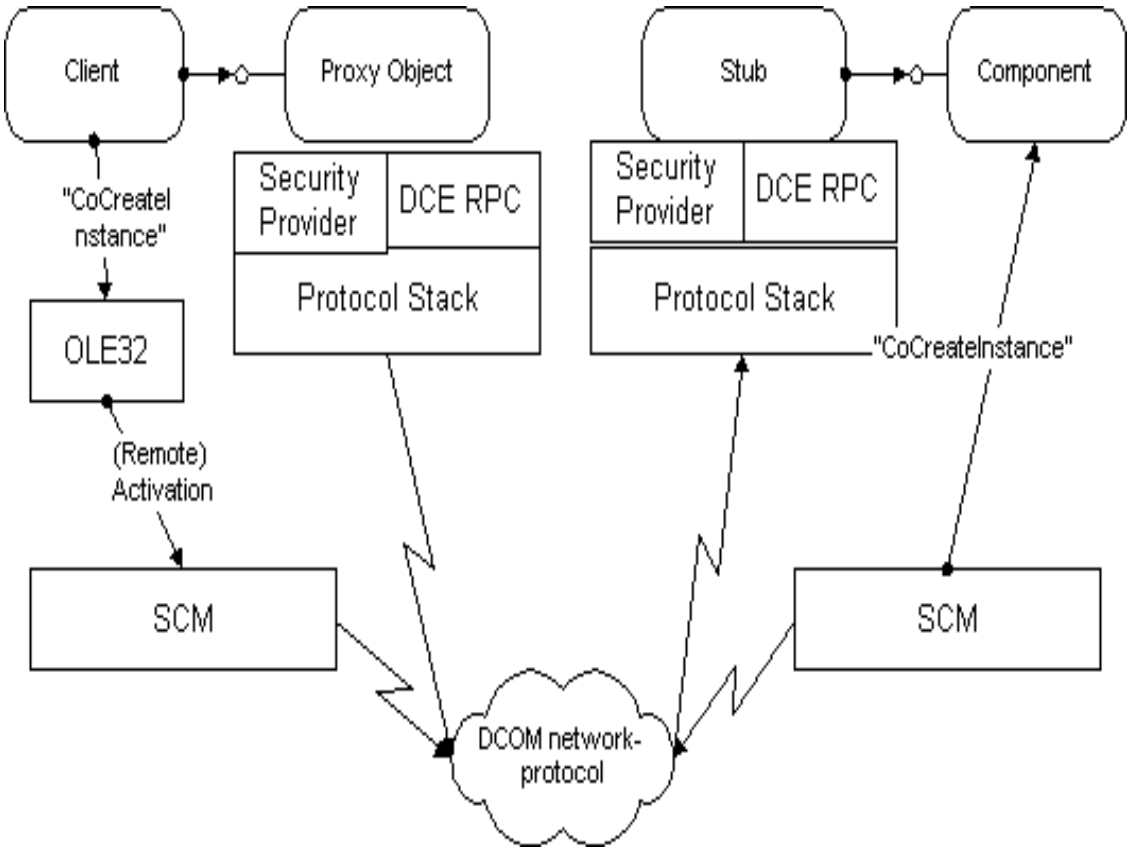


Figure 4.2: DCOM Architecture (Microsoft, 1996)

**4.2.2.1 DCOM Architecture**

Figure 3.2 shows the basic DCOM architecture. Each client uses DCOM components through interfaces. The client sends a request to activate the object of a server through CoCreateInstance (stands for Component Create Instance) and is passed back an interface pointer to that object. The client can access this interface through proxy object. The proxy object contains all the necessary network related commands. It marshals the client request and sends it through the channel called Distributed Computing

Environment Remote Procedure Call (DCE RPC) to the stub object. The stub object unmarshals the client request and invokes the method on the server object.

The client and server processes may run on different machines, possibly even different operating systems. Since DCOM is language independent and platform independent, to facilitate the communication between client and server, it provides a common language through Interface Definition Language (IDL). IDL is language independent and provides mapping to C or C++. The server's interface is developed with IDL and is compiled with Microsoft IDL (MIDL) compiler that produces proxy and stub code.

#### 4.2.2.2 Protocol Neutrality

DCOM provides this abstraction transparently: DCOM can use any transport protocol, including TCP/IP, UDP, IPX/SPX and NetBIOS. DCOM provides a security framework on all of these protocols, including connectionless and connection-oriented protocols.

#### 4.2.2.3 Platform Neutrality

DCOM is open to all approaches to cross-platform development. It does not preclude the use of platform-specific services or optimizations, nor does it favor a certain style of system services over others.

DCOM's architecture allows the integration of platform-neutral development frameworks and virtual machine environments (Java), as well as high-performance, platform-optimized custom components into a single distributed application.

#### 4.2.2.4 Cross-Platform Interoperability Standard

On the other hand, DCOM defines cross-platform services (or abstractions) for object-oriented distributed computing, including connection to, and creation of, components, locating components, invoking methods on components, and a security framework.

Beyond this, DCOM simply uses the services available on each platform to implement multithreading and concurrency control, user interface, file system interaction, non-DCOM network interaction, and the actual security provider.

#### 4.2.2.5 Available Platforms

**DCOM on Windows**

Implementations of DCOM are available today on the Microsoft Windows NT platform in Windows NT Workstation 4.0,Windows NT Server 4.0, Windows 9x, Windows 2000.

**DCOM on Apple Macintosh**

Microsoft implemented DCOM for the Apple Macintosh.

**DCOM on UNIX/Mainframe**

Versions of DCOM for UNIX platforms are developed (Software AG and Digital in close cooperation with Microsoft) and have been previewed publicly interoperating with implementations on Windows NT 4.0. DCOM is currently supported for Sun Solaris, AIX, MVS, and Linux.

### 4.3 Knowledge Component

The knowledge component contains the application specific knowledge. This application specific knowledge encodes the rules for the usage of a particular application. Depending on the application, this type of knowledge may or may not exist. If the legacy application is menu-driven, selection of different menus will lead to different outputs. This mapping between menu-selection and output can be encoded in application specific knowledge. It also contains input and output file formats. Most of the legacy systems in the forestry domain have been implemented in different languages like FORTRAN and C. The input and output formats are pretty rigid.

For describing the application we need a knowledge representation language. Since knowledge that is described in the component is sort of a gateway to the application, this knowledge representation language should be well known, user friendly, and a standard. Application experts are the people who are involved in the development of a particular legacy system and are well versed with usage of the system. On the other hand domain experts are the people who are familiar with an individual domain rather than a specific application. For example, application experts are involved in development of the knowledge component that is very application specific while, domain experts are involved in development of the inference component. Application experts who need not be experts in knowledge representation author the knowledge schema. As described earlier, XML is used for representing knowledge within the knowledge and inference components.

The applications present in OIISFD act as information sources that are highly autonomous and independent. The application experts are in charge of developing and maintaining the application schemas.

The partial knowledge schema for FVS is as follows:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
        targetNamespace="urn:fvs"
        xmlns:fvs="urn:fvs">
  <xs:complexType name="VariantType" mixed="true">
     <xs:attribute name="code" use="required" type="xs:string"/>
     <xs:attribute name="ext" use="optional" type="xs:string"/>
     <xs:attribute name="reg" use="optional" type="xs:string"/>
  </xs:complexType>
  <xs:complexType name="RegionType" mixed="true">
      <xs:attribute name="code" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="ExtensionType" mixed="true">
      <xs:attribute name="code" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="RuleType" mixed="true">
     <xs:sequence>
      <xs:element name="Variant" type="fvs:VariantType" />
     </xs:sequence>
  </xs:complexType>
  <xs:complexType name="FactType" mixed="true">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="Region" type="fvs:RegionType" />
            <xs:element name="Extension" type="fvs:ExtensionType" />
      </xs:choice>
  </xs:complexType>
  <xs:complexType name="ApplicationType">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="Rule" type="fvs:RuleType" />
            <xs:element name="Fact" type="fvs:FactType" />
      </xs:choice>
  </xs:complexType>
  <xs:element name="Application" type="fvs:ApplicationType" />
</xs:schema>
```

During system initialization, the custom XML Parser extracts rules from the knowledge component. A typical entry in the knowledge component about Forest Vegetation Simulator (FVS) is shown below:

<Variant code="se" ext=”cov”  reg="eleven">Southeast</Variant>

(If the region is “eleven” (SouthEast region) and extension is “cov” (Cover Shrubs), then

FVS variant is SouthEast).

### 4.4 Inference Component

The Inference component plays a pivotal role in the OIISFD architecture. It acts as a mediator/information broker between the user and the information sources (the legacy applications). The inference component possesses domain (forestry) knowledge, application (that are supported by OIISFD) description knowledge rather than application specific knowledge that is contained in the knowledge component. This draws a clear line between application and domain experts about who is in charge of administering the components. While domain experts develop and maintain the inference component, application experts develop and maintain the knowledge component.

One of the main requirements for an open system architecture is the use of standards wherever possible. Ideally, the forestry community should have a consensus on the semantics of the different terms used in the forestry domain. The development of such standards requires much consensus and effort. Ideally, the forestry community should have some language like a Forestry Markup Language (FML) that clearly specifies the semantics of the terms used in the forestry domain. This will alleviate the integration of various applications used in the forestry domain. For example, a plot can be described as a group of trees. However, there are no general standards that specify the number of trees that constitute a single plot. Another example of misunderstanding of terms is the usage of the term “plot”. In some applications, a “plot” signifies a particular area of land. However, in other applications, the term “plot” refers to a group of trees. This will lead to ambiguity (in the former example) and name collisions (in the latter example)

The inference component tries to solve these problems using XML namespaces (Namespaces, 1999).

Since one of the primary motivations for using namespaces is to be able to mix names from different sources, it is possible to provide an alias. This can be done by appending a colon and alias to the xmlns attribute. For example, the element plot can be uniquely identified by

xmlns:plot = “http://www.ai.uga.edu/forestry/forest.xml”

The inference component provides support for containing forestry vocabulary.

The inference component also contains application description knowledge.

### 4.4.1 Application Description Knowledge

This type of knowledge describes the application and relation(s) between applications. This knowledge aids in identifying the application for satisfying the user query. Ontologies are used to describe the knowledge of the application. In OIISFD, application’s meta-knowledge (knowledge of what it does) is represented using ontologies (in XML). The root ontology contains the following attributes:

| Slot Name | Slot Description | Slot Default Value |
|---|---|---|
| Name | The Application’s name | None |
| Version | Version of the application’s knowledge schema | 0.1 |
| Description | Brief description of the application usage (used only for documentation purposes). | This application is used in Forestry domain |
| Keyword List | List of key words associated with this application | None. |
| Species List | List of species supported by this application | None |
| Region List | List of the regions supported by this application | Entire United States |
| Dependency List | Application on which this application is depend on. And a mapping function for each | None |

| | | |
|---|---|---|
| | application above that maps output(s) of application with input(s) of the current application | |
| Owner | The original developer of the application | United States Department of Agriculture |
| Platforms | Platforms (operating systems) on which this application is supported on. | Windows |

Every application present in the OIISFD knowledge schema will be derived from this root ontology. The derived schema can possess extra slots that are pertinent to the application it represents. A reduced version of FVS's knowledge that is used in the inference component is as follows:

```
<Application>
      <name> Forest Vegetation Simulator </name>
       <Description>The Forest Vegetation Simulator is an individual-
       tree, distance- independent growth and yield model </Description>
        <KeywordList>
            <KeyWord>Simulation </Keyword>
            <KeyWord>Growth </Keyword>
            <KeyWord>Yield </Keyword>
       </KeywordList>
      <RegionList>
            <Region>Northern</Region>
            <Region>Southern</Region>
      </RegionList>
      <ExtenstionList>
            <Extension>Cover Shrubs</Extension>
            <Extension>'Regeneration Establishment'</Extension>
      <PostProcessorList>
            <PostProcessor>Standard Visualization System
      </PostProcessor>
      </PostProcessorList>
      <ConstraintList>
            <ConstraintName>Time</ConstraintName>
            <ConstraintSatisfactionMethod>
                  fvs_time_constraint_satisfaction
            </ConstraintSatisfactionMethod>
      </ConstraintList>
      </Application>
```

### 4.5 Legacy Component

The legacy component contains the legacy applications or databases and their wrappers.

These legacy applications comprise the main information sources. Some of the examples in legacy applications in forestry domains are decision support models such as growth and yield simulation models, forecasting models and visualization models. Some of these legacy applications such as visualization models cannot be wrapped, as they need to be interactive with the user. Therefore, they cannot be a part of the legacy component. Legacy applications such as forecasting models and growth and yield models are the ideal members to be part of legacy components. An example legacy application that is used here is FVS (Teck et. al, 1997).

We need to provide a mechanism by which the services of a legacy system can be accessed by the inference component. Wrappers are used for attaining this goal.

### 4.5.1 Wrappers

The wrappers around the legacy applications are an important part of the architecture. Wrappers should be designed such that, they hide idiosyncrasies of the legacy applications. They encapsulate and integrate the legacy applications with the rest of the system (Wiederhold, 1992). Wrappers basically expose the functionality of the legacy application. Since most of these legacy systems are huge, and stand alone, wrappers should provide a mechanism for invoking these applications remotely. All the communications between the legacy applications and the rest of the components are done through the wrappers. The inference component expects the output in the format specified in the outputschemafile of the applications knowledge schemas. So the wrapper should be able to parse the raw application specific data into the format specified in the outputschemafile of the applications knowledge schemas. Having the output from the legacy component in XML helps the inference component as it can easily manipulate

data of the elements present in the output file. This in turn, will help dynamic information exchanges that might be performed by the inference component.

By taking into consideration of above requirements, we have designed a custom wrapper for FVS. The wrapper in the legacy component in OIISFD is COM/DCOM based. Ideally, after developing these wrappers, the legacy applications can be invoked from any where without worrying about the mundane networking details. And also the interfaces of wrappers are clearly defined. The output and input into these wrappers are structured XML documents, rather than flat files.

Active Template Library (ATL) by Microsoft has been used for developing the wrapper for FVS. ATL has been specifically developed by Microsoft to assist developers building DCOM components in C++. It helps in the component registration process. All the custom interfaces required for the wrappers are defined using Microsoft Interface Definition Language (MIDL). The key point here is definition of language and platform neutral interfaces. From now, the clients, (who access the wrappers: in OIISFD, the inference component) who are developed using different languages and on different platforms can still be able to access the wrappers. We defined the following interface for the FVS Wrapper:

```
Interface IFVSWrapper : Iunknown
{
        [helpstring (“method RawFVSExecute”)
        HRESULT RawFVSExecute ( [in] BSTR FVSkeyfile,
         [in] BSTR FVSstandData, [out, retval] BSTR* FVStreeList, [out, retval]
        BSTR* FVSpostResult]
        [helpstring (“method FVSInputXML2Raw”)
        HRESULT FVSInputXML2Raw ( [in] BSTR FVSkeyfileXML,
        [in] BSTR  FVSstandDataXML, [out, retval] BSTR* rawFVSkeyfile,
         [out, retval] BSTR* rawFVSstandfile]
        [helpstring (“method FVSOutputRaw2XML”)
        HRESULT FVSOutputRaw2XML ( [in] BSTR FVStreelist,
```

```
        [in] BSTR  FVSpostResult, [out, retval] BSTR* FVStreelistXML,
         [out, retval] BSTR* FVSpostResultXML]
        [helpstring ("method FVSExecute")
        HRESULT FVSExecute ( [in] BSTR keyfileXML,
         [in] BSTR standDataXML, [out, retval] BSTR* treeListXML,
         [out, retval] BSTR* postResultXML]
}
```

The FVS wrapper interface (IFVSWrapper) is derived from IUnknown; all DCOM interfaces are derived from IUnknown. [in] and [out] are used to specify directions of data flow. [in] parameters are those that pass data from the client to the component. [out] parameters pass data from the component to the client. In IDL, all parameters, whether standard data types or buffer pointers, must be marked as [in], [out], or [in, out] (Microsoft 1995). The retval attribute designates the parameter that receives the return value of the interface member. All DCOM interfaces return a value of type HRESULT that returns S_OK if the operation succeeds and an error status code (E_FAIL, for example) if it fails.

The FVS Wrapper interface contains four methods:

- RawFVSExecute

  This is the main method of the interface and is internally called by other methods of the interface. This method takes stand data (the tree stand data on which the simulation has to be run), key file (the key file tells the simulation parameters that are to be used) as input and generates tree list and post process results as output.

- FVSExecute

  This method is used by the inference component to invoke the application. It takes XML streams as input and therefore hides the FVS specific format from

the user. It then converts these XML streams into FVS specific format and then calls RawFVSExecute

- FVSOutputRaw2XML, FVSInputXML2Raw

  These are the two helper functions that convert XML streams into FVS specific format and vice versa. They use inputfileformat and outputfileformat schemas present in the knowledge component.

The inference component can obtain the results from FVS in two ways:

- It can directly call FVSExecute and pass input data in XML
- Use helper functions and call rawFVSExecute.

**4.6 OIISFD Implementation**

OIISFD is implemented on a COM/DCOM platform. Currently, this framework supports two applications:

- Forest Vegetation Simulator (FVS)
- Test Application

The later application is a dummy application used for testing interoperability of the framework. The test application takes output from FVS and calculates average height of the stand before and after the simulation. The inference component acts as a controller and co-ordinates actions between various components of the framework.

OIISFD uses prolog for providing support for inference mechanisms. Amzi! Inc. provides a version of Prolog, which can be embedded in most of the popular programming languages. OIISFD's inference engine is developed using Amzi! Prolog. When the system is invoked, the inference component initializes the COM/DCOM framework. It then loads the inference engine.

The inference component contains a list of applications that are supported by OIISFD and their corresponding knowledge schemas in XML. It contains a custom XMLSchema parser that can parse the knowledge schemas. Application specific knowledge schemas are handled differently from application description knowledge schemas. If an application specific knowledge schema contains any rules, the parser parses them into Prolog production rules. For example, the following entry in application specific knowledge schema of FVS is parsed into a Prolog production rule:

```
<Variant code="se" ext=”cov”  reg="eleven">Southeast</Variant>
```

(If the region is “eleven” (SouthEast region) and extension is “cov” (Cover Shrubs), then FVS variant is SouthEast).

```
rule id0:
      [ extension(cov),
      region(se) ]
      =>
      [retract(all),
      assert(application(se))].
```

In OIISFD, ontologies are represented using frames internally. The parser parses application description knowledge schemas into frames. For example, the following entry in the application description knowledge schema of FVS is parsed into a Prolog Frame as follows:

```
<Application >
      <name> Forest Vegetation Simulator </name>
      <Description>The Forest Vegetation Simulator is an individual-
      tree, distance- independent growth and yield model
      </Description>
      <KeywordList>
            <KeyWord>Simulation </Keyword>
            <KeyWord>Growth </Keyword>
            <KeyWord>Yield </Keyword>
      </KeywordList>
      <SpeciesList>
            <Species>Hardwood </Species>
      </SpeciesList>
      <PostProcessorList>
            <PostProcessor>Standard Visualization System
            </PostProcessor>
      </PostProcessorList>
</Application>
```

```
frame(fvs, [
ako-[val forestry_application],
description-[val ‘ The Forest Vegetation Simulator is an
individual-tree, distance- independent growth and yield model’],
keywordlist-[val [simulation, growth, yield] ],
      specieslist-[val [hardwood] ],
      postprocessorlist-[val [‘Standard Visualization System’]
]]).
```

### 4.6.1 Inference Engine Implemenation

The inference engine contains two PSMs:

- Rule Interpreter
- Frame-logic Processor.

As mentioned earlier, the inference engine is implemented using Prolog. Once various rules associated with different applications are asserted in the rule base, the rule interpreter PSM can be invoked by calling the predicate *go/0*. The main processing of the rule interpreter is as follows:

```
go :-
call(rule ID: LHS ==> RHS),
try(LHS,RHS),
write('Rule fired '),write(ID),nl,
!,go.

go:-
nl, write(done), nl.
```

The second PSM present in the inference engine contained in the inference component of OIISFD is a frame-logic processor. Once various frames associated with different applications are asserted in the frame base, the frame-logic processor PSM can be invoked by calling the predicate *get_frame/2*. The main processing of the frame-logic processor is as follows:

```
get_frame(Thing, ReqList) :-
frame(Thing, SlotList),
slot_vals(Thing, ReqList, SlotList).
```

The following two predicates are used for maintaining the frame base:

```
% add_frame/2
% The add_frame predicate uses the same basic form as get_frame.
% For updates, first the old slot list is retrieved from the existing frame.
% Then the predicate add_slots is called with the old list (SlotList) and the update list (UList).
% It returns the new list (NewList).

add_frame(Thing, UList) :-
old_slots(Thing, SlotList),
add_slots(Thing, UList, SlotList, NewList),
retract(frame(Thing, _)),
asserta(frame(Thing, NewList)), !.

% del_frame/2
% The del_frame predicate uses the same basic form as get_frame.
% For deletions, first the old slot list is retrieved from the existing frame.
% Then the predicate del_slots is called with the old list (SlotList) and the update list %(UList).
% It returns the new list (NewList).

del_frame(Thing, UList) :-
old_slots(Thing, SlotList),
del_slots(Thing, UList, SlotList, NewList),
retract(frame(Thing, _)),
asserta(frame(Thing, NewList)).
```

A framework has been developed for data type conversion (from Prolog to C++ and vice-versa), handling Amzi! Prolog exceptions, for loading/unloading embedded Prolog. This framework includes an abstract layer (over the API provided by Amzi! Prolog) for making prolog queries, calling C++ methods by prolog predicates and prolog query result processing.

### 4.6.2 Query Processing Implementation

The query-processing module is part of the inference component. The OIISFD user interacts with the system by typing appropriate requests. Depending upon the query, appropriate applications are run, if needed, the system presents screens for user input and results are displayed. The inference component contains a query-processing module to handle different types of queries. The inference component with the help of query-processing module and controller module responds to the user requests. The query-processing module converts the user request into a prolog query and then hands the

prolog query to the controller module for appropriate action. For example, if the user wants to find the description of FVS, he/she will type:

```
fvs description.
```

The query-processing module converts this to a prolog query:

```
get_frame(fvs,[description-Description]).
```

The controller module then queries the knowledge base with this prolog query and displays the result to the user.

For handling sophisticated user queries, a query planner has been implemented. This query planner tries to satisfy the user constraints before running appropriate application. The main processing of query planner is as follows:

```
query_planner(Keyword, ConstraintNameList, ConstraintValueList,
ConstraintSatisfactionMethod) :-
get_frame(Frame, [keywordlist-Keyword,dependencylist-DependencyList]),
make_application_list([Frame],DependencyList,ApplicationList),
satisfy_constraint(ApplicationList, ConstraintNameList,
ConstraintValueList,ConstraintSatisfactionMethod), !,
CallMethod =.. [ConstraintSatisfactionMethod,ConstraintValueList],
CallMethod.
```

# CHAPTER 5

# RESULTS

The interfaces (IFVSWrapper, IDummyWrapper) have been successfully implemented for the FVS application. Legacy and knowledge components are developed according to the OIISFD architecture. The inference component uses an embedded inference engine for decision-making. XML is used as a communication mechanism between different modules. Additional applications can easily be added to the framework.

Since, XML is human readable and easy to author, a knowledge component can be developed with relative ease by the application experts themselves. Development of wrappers for the legacy component requires considerable programming effort. To make these wrappers reusable, the interfaces should be developed as generic as possible (similar to IFVSWrapper).

The inference component is a middleman between the user and the knowledge component. When the user forwards a request, the inference component using the prolog module identifies the needed application(s). Once it identifies the required application(s), it preprocesses the data associated with the user's requests. With the help of input-file-schemas of those applications it translates the user's data into the format required by the wrapper(s). The knowledge component also contains the location and Interface IDentifier (IID) of the wrapper. The inference component then invokes the wrapper with the help of

COM/DCOM run time services. Once it gets the wrapper's interface pointer, it runs the methods with the preprocessed (and translated) user's data as input.

The next section shows the various usage cases of the OIISFD framework.

### 5.1 OIISFD Screen shots

When the system is invoked, it presents the following screen to the user:

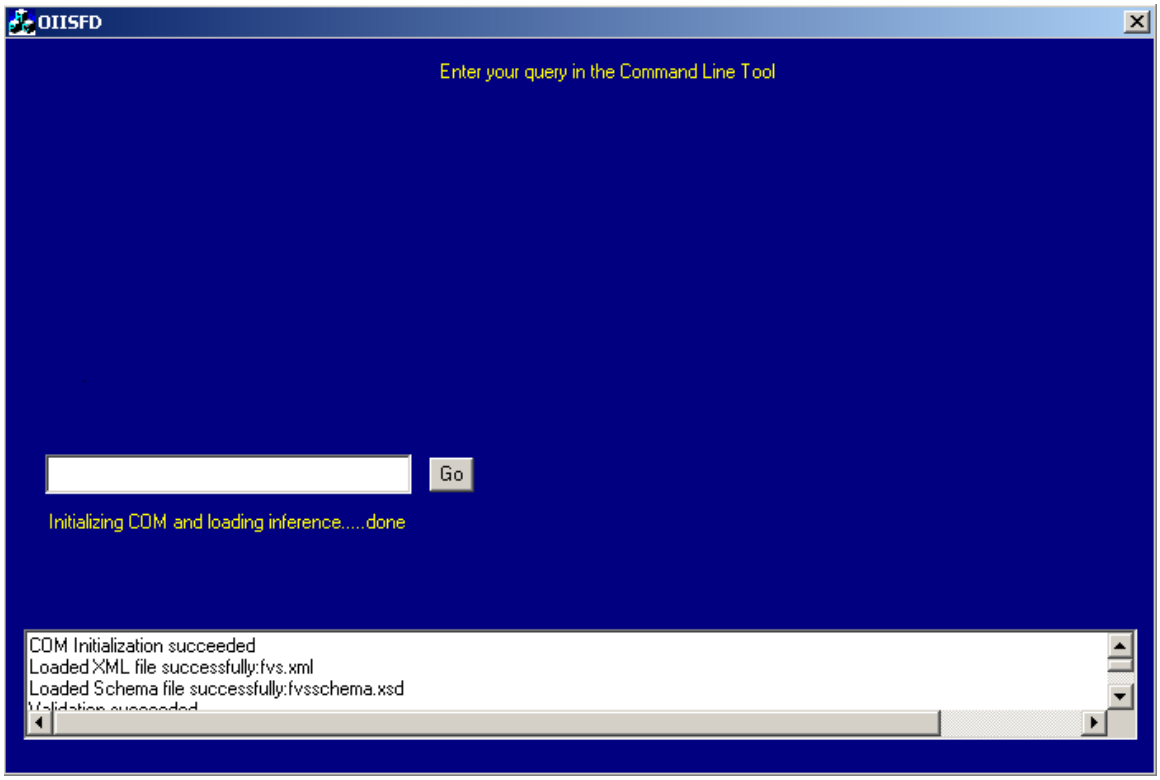


Figure 5.1 OIISFD initialization Screen

The user can ask various questions to the system by typing in the command-line box. For example, if the user wants to know various species supported by fvs, he/she will type:

```
fvs specieslist
```

The query processor converts this query to the following prolog request and presents the results to the user.

```
get_frame(fvs, [specieslist-X]).
```

The user can also query the embedded the knowledge base by prepending “prolog” to his/her query. For example, if the user wants to see rules present in the knowledge base, he/she will type “prolog rule(X).”. The query processor recognizes that this query is a prolog query and passes it unchanged to the controller. The following screen shows the above process.

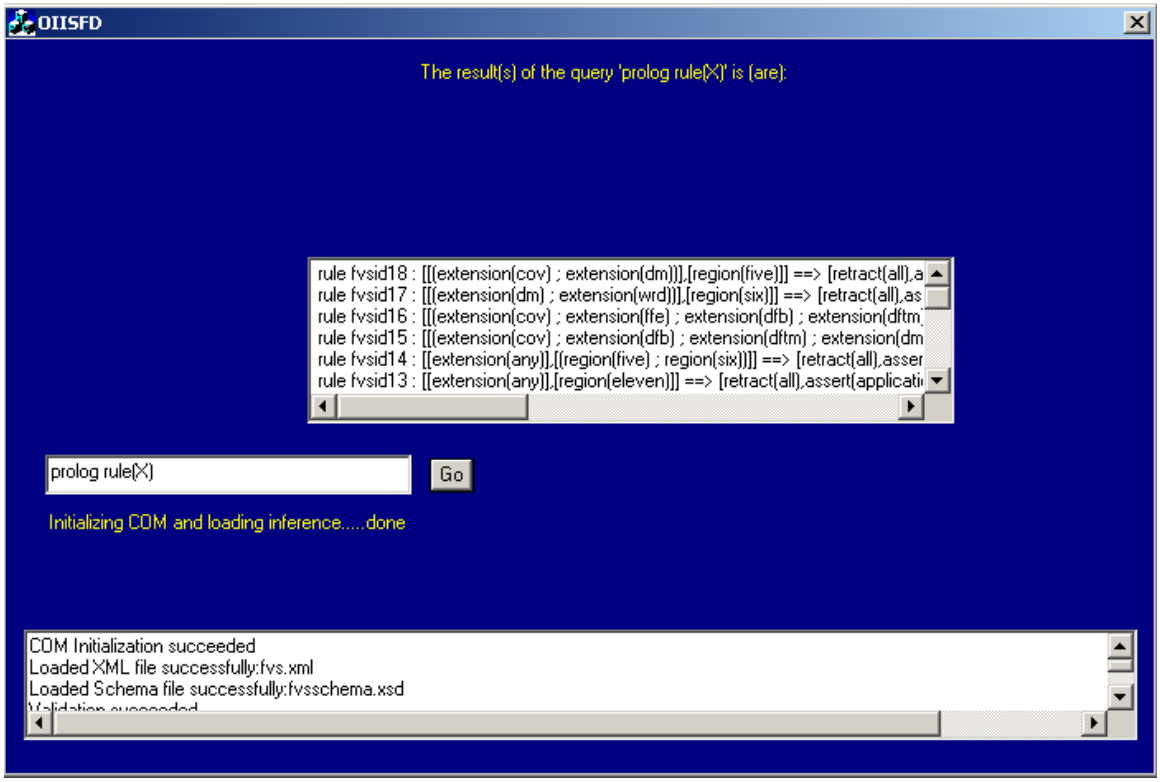


Figure 5.2 Output of the query “prolog rule(X)”

OIISFD has support for running applications based on the user query. For example, if the user wants to run a simulation, he/she will type, “*query simulation*” in the command-line box. The query processor converts this query into the following prolog query:

```
get_frame(X,[keywordlist-simulation, dependencylist-Y]).
```

Here, dependencylist contains the list of applications (and their corresponding wrapper names). Since FVS is not dependent on any applications, this list will be empty. Once the inference component infers the application as FVS, it launches the FVS screen. For

identifying the correct variant of FVS, the user has to select region and extension. The user selections are passed to the inference engine, which uses the rules associated with FVS to identify the correct variant. After identifying the appropriate variant, the controller asks the user for input data. It then runs the application and displays the results to the user.

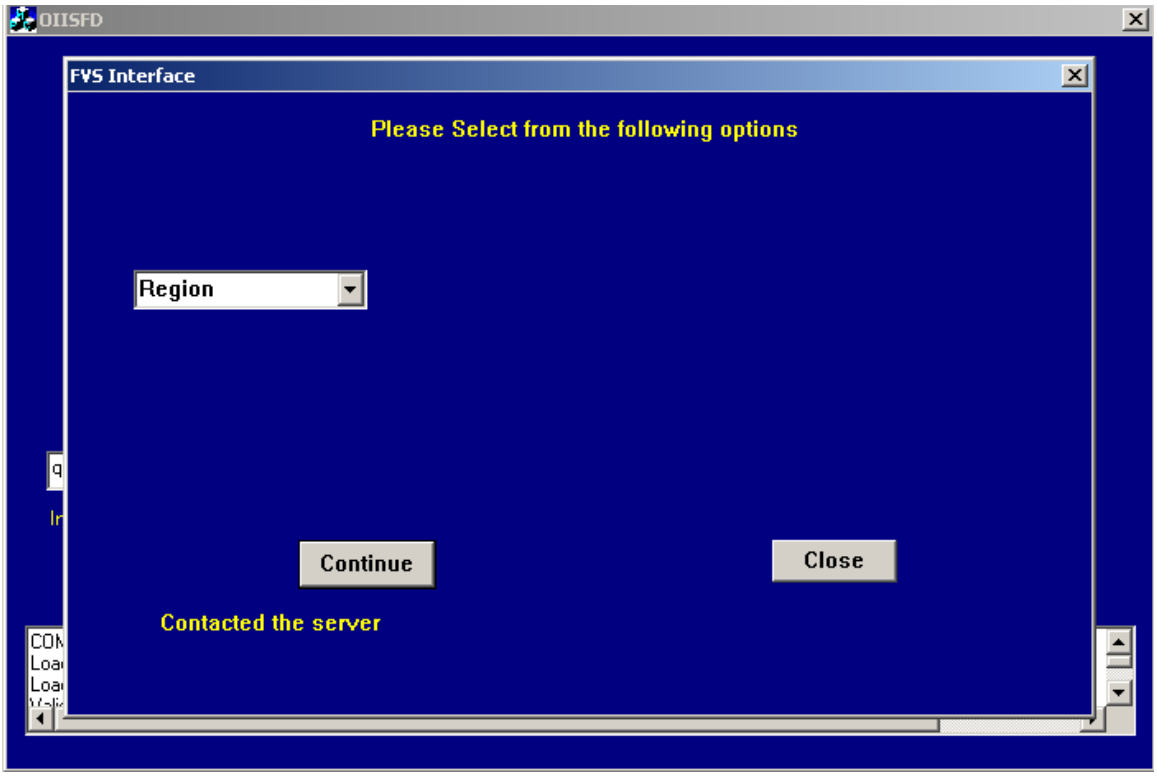


Figure 5.3 Fvs input screen.

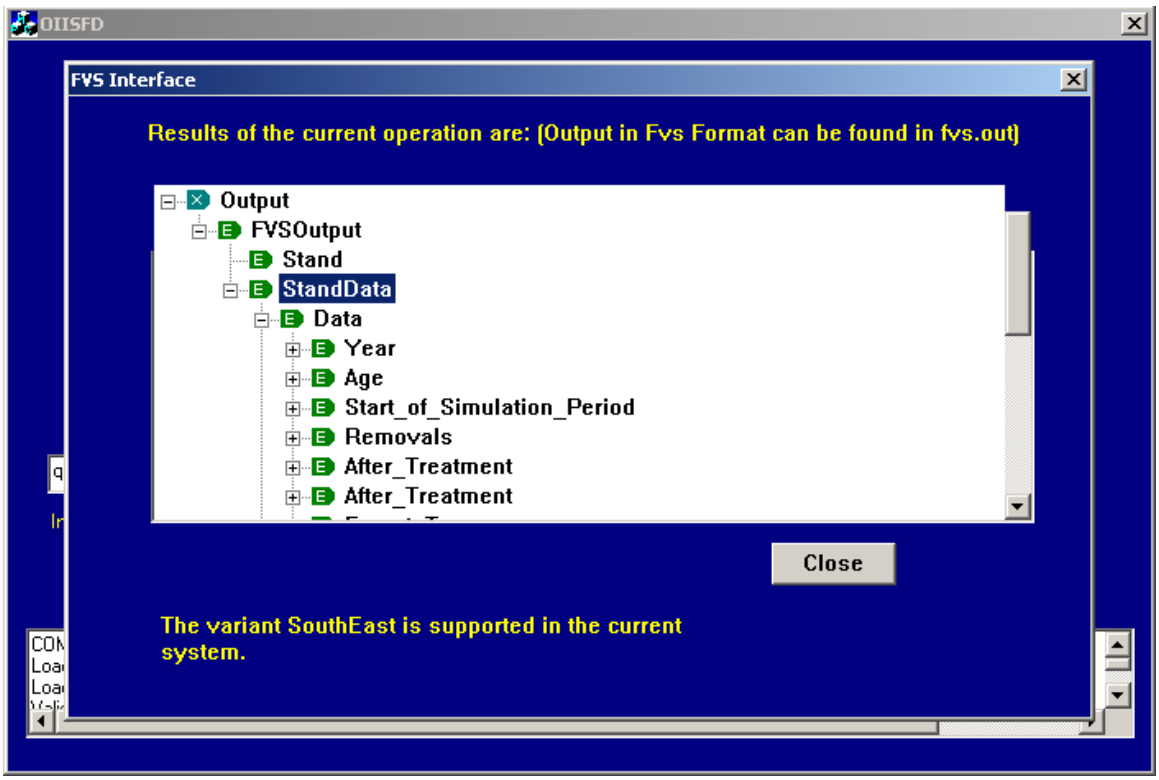


Figure 5.4 FVS output screen.

To test the interoperability framework, a test application has been added to the OIISFD. This test application calculates the average height of a stand before and after the simulation. The knowledge schema of this test application is as follows:

```
<Application >
      <name>Dummy</name>
      <Description>This application tests the interoperability of OIISFD
      architecture/Description>
      <KeywordList>
            <KeyWord>average_height </Keyword>
            <KeyWord>dummy </Keyword>
            <KeyWord>interoperability </Keyword>
      </KeywordList>
      <DependencyList>
            <DependecyApplication>fvs</DependencyApplication>
            <DependencyApplicationWrapper> fvs_dummy_conversion
            </DependencyApplicatoinWrapper>
      </DependencyList>
      <Owner>rajesh</Owner>
</Application>
```

When the user types the query “query average_distance”, the inference component identifies the application as Dummy. But since this application depends on FVS, it first runs FVS. During the system initialization all the wrapper functions are loaded in a map.

It later uses the wrapper function to convert FVS data to the dummy application input data. It then launches the dummy application that calculates the average height and then the inference component displays the results to the user.

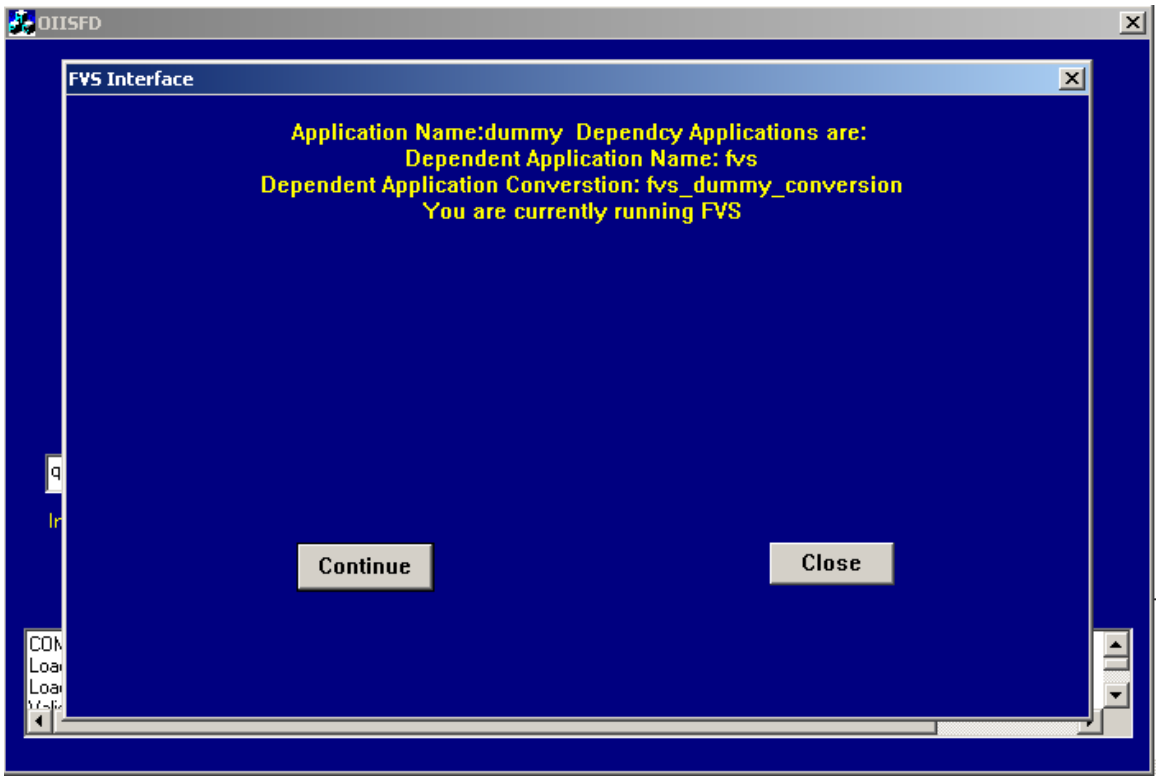


Figure 5.4: FVS is identified as an dependent application and it is run first.

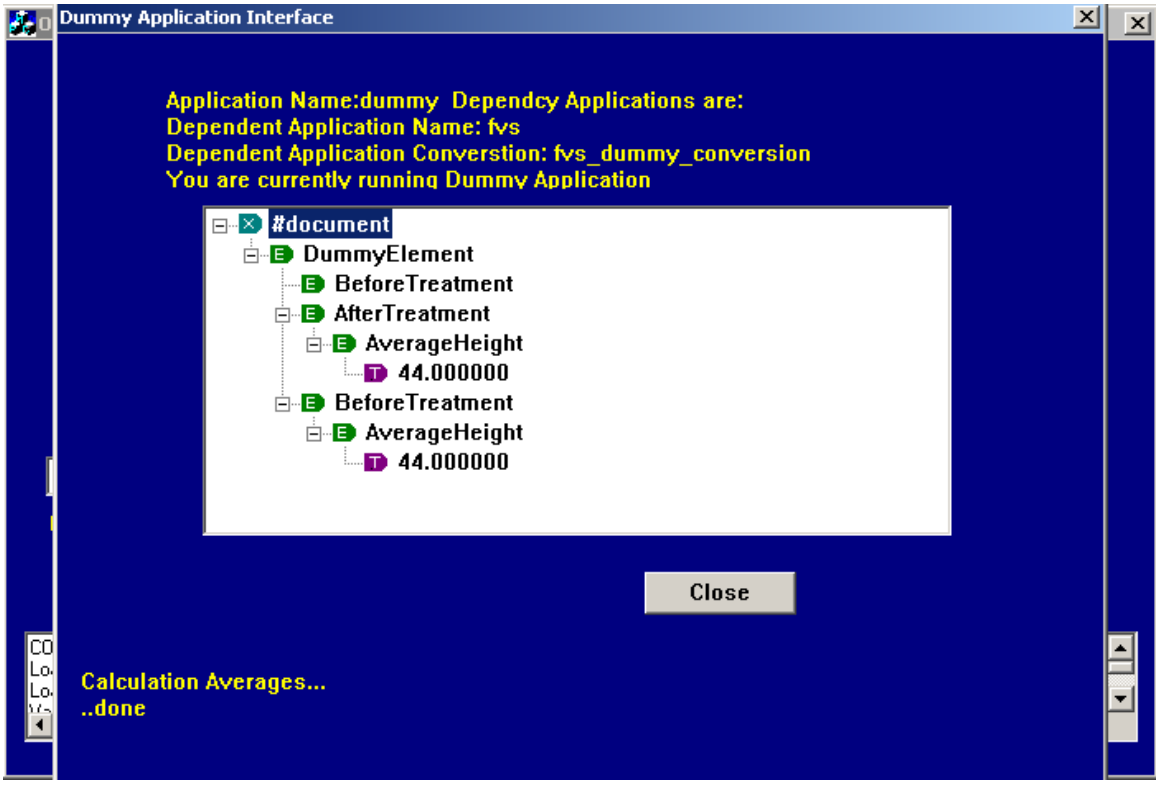


Figure 5.5: The final output of the query "query average_distance".

Some of the query satisfaction process can be automated. For example, one of the common queries for FVS is to vary the simulation time. This type of common query can be encoded as a constraint and a constraint satisfaction method can be developed that satisfies the user query without his/her intervention. For example, the user can query OIISFD as follows:

```
query –keyword average_height –constraint_name time –
constraint_value 50 –inputfile standfile
```

(Find average height of the stand present in standfile after running the simulation for 50 years).

The inference component uses query_planner/3 for satisfying this type of query. The query_planner first infers the application that can be used to satisfy the keyword. It then searches for a constraint satisfaction function that can satisfy the given constraint in the

current application and all the dependent applications. It runs the constraint satisfaction function and then runs the application and dependent applications with minimum user intervention.

# CHAPTER 6

# CONCLUSIONS AND FUTURE DIRECTIONS

The primary goal of this thesis was to propose an architecture for the integration of systems in the forestry domain. The architecture is scalable, flexible and extensible. Human roles are clearly identified in the integration of the legacy systems. Domain experts will develop and maintain the inference component, application experts maintain the knowledge component while programmers/developers will develop and maintain wrappers in the legacy component. Using COM/DCOM as a platform, on which the various components are built, the proposed architecture is language and platform neutral. With the help of the schemas that are defined in the inference component we have a common language in which various legacy components can exchange information. According to the open systems approach, standards should be used wherever possible. Since XML is a universal standard, the proposed architecture conforms to this requirement. By using XML namespaces, ambiguities in name collisions of various entities in forestry domains, can be avoided.

Another goal of this thesis was to investigate the use of XML as a knowledge representation language. It has been demonstrated that XML with schemas has enough expressiveness to use it as a knowledge representation language. This breaks down the barriers between data and information. Since XML is human readable and easy to author, the schemas can be easily maintained. The experts themselves can develop these schemas, thereby avoiding the tedious process of knowledge acquisition. Building

wrappers and knowledge schemas in the proposed way, the legacy systems can be integrated into any system with little modifications. The inference engine present in the inference component was built using knowledge engineering principles.

The forestry community does not have a common language for the integration of various systems. This hinders the process of integrating legacy systems. Efforts should be made to develop a common language (Forestry markup Language) preferably using XML.

The current work provides guidelines for integrating growth and yield models and forecasting models. The process of integrating visualization models still needs to be investigated. The current inference component contains knowledge that is built from scratch. The inference component has to be made interoperable with existing knowledge sources. More theoretical foundations have to be developed for the use of XML as a knowledge representation language.

## BIBLIOGRAPHY


A. Newell. 1982. The Knowledge Level. *Journal of Artificial Intelligence*, 18(1): pages 87-127.

D. Fensel, H. Eriksson, M. A. Musen, and R. Studer. 1996. Conceptual and Formal Specifications of Problem-Solving Methods*, International Journal of Expert Systems*, 9(4):507-532.

Durking, J. 1994. *Expert Systems: Design and development*, Prentence Hall Publications.

Grimes, R. 1997. *Professional DCOM Programming*, Birmingham, U.K: Wrox Press Ltd.

Gruber T. 1991. The role of a common ontology in achieving sharable, reusable knowledge bases. *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, Cambridge: 601-602.

Guarino N. 1998. Formal ontology and information systems. *Proceedings of the 1st International Conference on Formal Ontology in Information Systems* [FOIS'98], Torino: 3-15

J.M David, J.P. Krivine and R.Simmons, eds. 1993. *Second Generation Expert Systems*, Springer-Verlag, Berlin.

Liu, S. 1998. Integration of Forest Decision Support Systems: A search for Interoperability. *Master's Thesis, The University of Georgia*, Athens, GA.

M. Klein, D. Fensel, F. van Harmelen, and I. Horrocks: The relation between ontologies and XML schemas, to appear *in Electronic Transactions on Artificial Intelligence (ETAI)*.

Martin, D., Oohama, H., Moran, D., and Cheyer, A., 1997. Information Brokering in an Agent Architecture. *Proceedings of The Practical Application of Intelligent Agents and Multi-Agent Technology* (PAAM 97).

Microsoft. 1996. The Distributed Component Object Model: Technical overview *http://www.microsoft.com/com*

Mowrer, H.T., Barber, K., Campbell, N., Crookston, C., Dahms, J., Day, J., Laacke, J., Merzenich, S., Mighton, M., Rauscher, M., Reynolds, K., Thompson, J., Trenchi, P., and Twery, M. 1997. Decision Support Systems for Ecosystem Management: An evaluation of existing systems. *General Technical Report RM-GTR-296*. Fort Collins, CO: USDA Forest Service, Rocky Mountain Forest and Range Experiment Station.

Namespaces in XML 1.0 *http://www.w3.org/TR/REC-xml-names*, February 1999. W3C Recommendation

Kent, R.. 1999. Conceptual Knowledge Markup Language: The Central Core. Electronic *Proceedings of the Twelfth Workshop on Knowledge Acquisition, Modeling and Management*, Banff, Alberta, Canada, 16–21 October 1999.

Rudi Studer, Dieter Fensel, Stefan Decker, and V. Richard Benjamins. March, 1999. Knowledge Engineering: Survey and Future Directions. *Lecture Notes in Artificial Intelligence*, LNAI 1570, Springer-Verlag.

Sheth, A. 1998. Changing Focus on Interoperability in Information Systems: From System, Syntax, Structure to Semantics. *Interoperating Geographic Information Systems*, Kluwer Publications.

Software Engineering Institute. The Open Systems Approach at SEI, Carnegie Mellon *http://www.sei.cmu.edu/opensystems/welcome.html*

Somasekar, S. 1999. An Intelligent Information System for Integration of Forest Decision Support Systems. *Master's Thesis, The University of Georgia*, Athens, GA.

Teck, R., Moeur, M., and Eav, B. 1997. The forest vegetation simulator: A decision support tool for integrating resources science. *http://www.fs.fed.us/ftproot/pub/fmsc/fvsdesc.htm*

Twery, M.J., Bennett, D.J., Kollasch, R.P., Thomasma, S.A., Stout, S.L., Palmer, J.F., Hoffman, R.A., DeCalesta, D.S., Hornbeck, J., Rauscher, H.M., Steinman, J., Gustafson, E., Miller, G., Cleveland, H., Grove, M., McGuinness, B., Chen, N., and Nute, D.E. 1997. NED-1: An integrated decision support system for ecosystem management. *Proceedings of the Resource Technology* 1997 Meeting. Pp 331-343

Twery, M. NED: A Set of Tools For Managing Non Industrial Private Forests in the East *http://www.fs.fed.us/ne/burlington/research/ne4454/ned/pub1.htm*

Wiederhold G. 1992. Mediators in the architecture of future information systems. *IEEE Computer* 25(3): 38-49.

Wiederhold G. 1997. Integration of Heterogeneous Information at Stanford. Power point Presentation. *http://www-db.stanford.edu/pub/gio/slides/Stanford1/index.htm*

Extensible Markup Language (XML) 1.0. *http://www.w3.org/TR/REC-xml*, February 1998. W3C Recommendation

XML Schemas Part1 & Part 2. *http://www.w3.org/TR/xmlschema-1*, March 2001. W3C Recommendation

## APPENDIX A

## HEADER FILES OF IMPORTANT C++ CLASSES OF OIISFD

```
/*********************************************************
/* XML (Schema) parser class. This class serves two
/* purposes: 1. For validating a XML File against a schema.
/* 2. For parsing knowledge into Prolog rules and facts
/*
/*********************************************************

#ifndef PARSER_H__
#define PARSER_H__

#include "StdAfx.h"
#include "OIISFD.h"
#include <iostream.h>
#include "Framework.h"
#include "Prolog.h" // For inference Engine support


class Parser {

     // Initializes COM
     // Parses and validates the xml file against schemaFile
public:

     Init(const CString & xmlFileName, const CString &
     schemaFileName);

     // Extracts Rules and facts from the XML. Initializes
     //and starts Prolog Engine
     void Parse();

     //Add rules the prolog knowledge base
     int AddFrames(IXMLDOMNodeList *ruleList);

     //Add rules the prolog knowledge base
     int AddRules(IXMLDOMNodeList *ruleList);

     //Add facts to the knowledge base
     int AddFacts(IXMLDOMNodeList *factList);

     //Add internal facts to the knowledge base
     // This function is overloaded
     int AddInternalFact(CString functor, CString argument1,
     CString argument2);
```

```
        int AddInternalFact(BSTR functor, VARIANT argument1,
        BSTR argument2);

        int AddInternalFrame(BSTR functor, CStringList*
        slotList,CMapStringToList* slotValueList);

        //Add internal rules to the knowledge base
        //this function will be overloaded.
        int AddInternalRule(int ruleID, VARIANT extValue,
        VARIANT regValue, VARIANT codeValue);

        //This method Query the knowledge base and returns the
        result as a CString List
        CStringList* QueryKnowledgeBase(CString userQuery);

        //This method queries the frame base and gets the
        //application info
        CMapStringToPtr*  QueryFrameBase(CString userQuery);

        //This method Queries the knowledge base for Constraint
        //Satisifaction
        bool QueryConstraintSatisfaction(CString userQuery);

        // Utility methods to convert COM and XML Errors to
        //errors of OIISFD
        static CString GetParserExceptionMessage(const
        IXMLDOMParseErrorPtr pError);
        static CString GetCOMExceptionMessage(_com_error &e);

        //This method classifies the user query.
        UserQueryType ClassifyUserQuery(CString userQuery);
        // constructor and destructor
        Parser();
        ~Parser();

private:

        // the logic server. will contain the inference module
        //after it gets initialized
        CProlog* m_pInferenceEngine;

        // Contains the document Element
        IXMLDOMDocument2Ptr  m_pXMLDOMDocument;

        //Contains the rootElement of the document
        IXMLDOMElement *m_pRootElement;

};


#endif
```

```
/*********************************************************
/* CProlog class. This class provides an abstraction over
/* AMZI! Inc's embedded API. This the main interface for
/* embedding Prolog in C++.
/*********************************************************

#ifndef PROLOG_H__
#define PROLOG_H__

#include "StdAfx.h"
#include "OIISFD.h"
#include "Framework.h"
#include "amzi.h" //Embedded prolog support
#include <iostream.h> // for output

class CProlog : public CLogicServer
{
public:
     CProlog();
     virtual ~CProlog();

     //This function init Prolog Engine given the inference
     //module name
     TF Init(CString fileName);

     //Queries the knowledge base and returns the result in
     //a list
     CStringList* QueryKnowledgeBase(CString userQuery);

     //Query the knowledge base and returns the result in a
     //list. This method should be called if the user wants
     // to query the knowledge base directly.
     CStringList* QueryKnowledgeBase(CString userQuery, bool
     prologFormat);

     //This method is quering ontology of applications
     CMapStringToPtr* QueryFrameBase(CString userQuery);

     //This method is for constraint Satisfaction
     bool QueryConstraintSatisfaction(CString userQuery);

     //Extracts the variable from the Prolog Query
     void ExtractVariableValues(CString prologQuery, CString
     resultString, CStringList &variableValueList);

     //This method constructs the dependency list
     CStringList* ConstructDependencyList(TERM* targetList);

     //Asserts a new Prolog statement to the knowledge base
     int AddPrologStatement(CString statement);
     //
     TF time_constraint_satisfaction();
private:
```

```
     RC m_rc;
     ENGid inferenceEngine;
     CString GetPrologException(CLSException &E);
};


#endif //
!defined(AFX_PROLOG_H__577ECC45_95B5_4370_9A9E_F5E27CCB96ED_
_INCLUDED_)



/*************************************************************
/* WrapperFactory class. This class contains a map of
/* function name (for wrappers) to function pointers.
/*************************************************************************
***

#if
!defined(AFX_WRAPPERFACTORY_H__175E016E_9A50_45D6_B963_491AC
8A23A53__INCLUDED_)
#define
AFX_WRAPPERFACTORY_H__175E016E_9A50_45D6_B963_491AC8A23A53__
INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "Afxtempl.h"
#include "Framework.h"

typedef IXMLDOMDocument2Ptr
(*WrapperFunctionPtr)(IXMLDOMDocument2Ptr);
typedef CMap<CString, LPCSTR, WrapperFunctionPtr,
WrapperFunctionPtr&> MapCStringToFunctionPtr;
//typedef std::map< std::string, pfn* > pfnMap;


//This Wrapper factor is essentially a table that maps
function names to their implementations
class WrapperFactory
{
public:
     //Constructor and destructor.
     WrapperFactory();
     virtual ~WrapperFactory();
     //initializes the map and loads the existing wrapper
     void Init();
     WrapperFunctionPtr GetWrapper(CString wrapperName);
private:
     CMapStringToPtr* m_wrapperMap;
};
```

```
//All wrappers are should be declared here
IXMLDOMDocument2Ptr fvs_dummy_conversion(IXMLDOMDocument2Ptr
fvsOutput);

#endif //
!defined(AFX_WRAPPERFACTORY_H__175E016E_9A50_45D6_B963_491AC
8A23A53__INCLUDED_)


/*************************************************************
/* Framework header file. It contains some helpers methods
/* that are used all over the application.
/*************************************************************
***

#ifndef FRAMEWORK_H__
#define FRAMEWORK_H__

#include "StdAfx.h"
#import <msxml4.dll> named_guids //XML Parser support

using namespace MSXML2;

// This is the base class for the exceptions. All OIISFD
// Exceptions should derive from this Exception class.


class Exception {
private:
     CString m_exceptionMessage;
public:
     Exception() { };
     virtual ~Exception() { };
     virtual void SetErrorMessage(const CString
&errorMessage) { m_exceptionMessage = errorMessage;}
     virtual const CString GetErrorMessage() { return
m_exceptionMessage; }
};

// a string utility
CString
GetString(CString &tmp, CString Search);

// Different types of User Query
enum UserQueryType
{
     APPLICATION_QUERY=0,
     GENERAL_QUERY,
     PROLOG_QUERY,
     CONSTRAINT_QUERY,
     ILLEGAL_QUERY
};

#endif
```

# APPENDIX B

## PROLOG SOURCE FOR REASONING

- Frame Systems Reasoner (Durkin, 1994)

```
:- op(600, fy, val).     % used for facet "Value"
:- op(600, fy, calc).    % used for facet "Calculate"
:- op(600, fy, def).    % used for facet "Default"
:- op(600, fy, add).     % used for facet "Add"
:- op(600, fy, del).    % used for facet "Delele"

% retrieve a list of slot values

get_frame(Thing, ReqList) :-
frame(Thing, SlotList),
slot_vals(Thing, ReqList, SlotList).
get_frame(Thing, ReqList, FlatReqList) :- get_frame(Thing,
ReqList),write(ReqList),
                        temp_convert(ReqList,
FlatReqList).

% retrieve a list of slot values

get_frame(Thing, ReqList) :-
frame(Thing, SlotList),
slot_vals(Thing, ReqList, SlotList).
get_frame(Thing, ReqList, FlatReqList) :- get_frame(Thing,
ReqList),write(ReqList),
                        temp_convert(ReqList,
FlatReqList).

slot_vals(_, [], _).

slot_vals(T, [Req|Rest], SlotList) :-
prep_req(Req, req(T, S, F, V)),
find_slot(req(T, S, F, V), SlotList),
!, slot_vals(T, Rest, SlotList).

%This takes care when request is not a list of slot values

slot_vals(T, Req, SlotList) :-
prep_req(Req, req(T, S, F, V)),
find_slot(req(T, S, F, V), SlotList).

% prepares the formal query structure

% When the value is being sought
```

```
prep_req(Slot-X, req(T, Slot, val, X)) :- var(X), !.

% When facet is being sought

prep_req(Slot-X, req(T, Slot, Facet, Val)) :-
nonvar(X),
X=..[Facet, Val],
facet_list(FL),
member(Facet, FL), !.

% for comparision
prep_req(Slot-X, req(T, Slot, val, X)).

facet_list([val, def, calc, add, del, edit]).

%when the value is single value

find_slot(req(T,S, F, V), SlotList) :-
nonvar(V),
find_slot(req(T, S, F, Val), SlotList),
!,
(Val == V; member(V, Val)).

%when the value is a list

find_slot(req(T, S, F, V), SlotList) :-
member(S-FacetList, SlotList),
!, facet_val(req(T, S, F, V), FacetList).

% inheritance

find_slot(req(T, S, F, V), SlotList) :-
member(ako-FacetList, SlotList),
facet_val(req(T, ako, val, Ako), FacetList),
(member(X, Ako); X = Ako),
frame(X, HigherSlots),
find_slot(req(T, S, F, V), HigherSlots), !.

% for error handling
find_slot(Req, _ ) :-
error(['frame error looking for:', Req]).

% For getting the facet value

% requested facet and value are on the facet list

facet_val(req(T, S, F, V), FacetList) :-
FV =.. [F, V],
member(FV, FacetList), !.

% requested facet is val, it is on the facet list and its
value is a list

facet_val(req(T, S, val, V), FacetList) :-
```

```
member(val ValList, FacetList),
member(V, ValList), !.

% def facet value

facet_val(req(T, S, val, V), FacetList) :-
member(def V, FacetList), !.

% calc to get the facet value

facet_val(req(T, S, val, V), FacetList) :-
member(calc Pred, FacedList),
Pred =.. [Functor|Args],
CalcPred =.. [Functor, req(T, S, val, V) | Args],
call(CalcPred).

% add_frame/2
% The add_frame predicate uses the same basic form as
get_frame.
% For updates, first the old slot list is retrieved from the
existing frame.
% Then the predicate add_slots is called with the old list
(SlotList) and the update list (UList).
% It returns the new list (NewList).

add_frame(Thing, UList) :-
old_slots(Thing, SlotList),
add_slots(Thing, UList, SlotList, NewList),
retract(frame(Thing, _)),
asserta(frame(Thing, NewList)), !.

% old_slots predicate returns the slot list

old_slots(Thing, SlotList) :-
frame(Thing, SlotList), !.

old_slots(Thing, []) :-
asserta(frame(Thing, [])).

% add_slots work similar to slo_vals

add_slots(_, [], X, X).

add_slots(T, [U|Rest], SlotList, NewList) :-
prep_req(U, req(T, S, F, V)),
add_slot(req(T, S, F, V), SlotList, Z),
add_slots(T, Rest, Z, NewList).

add_slots(T, X, SlotList, NewList) :-
prep_req(X, req(T, S, F, V)),
add_slot(req(T, S, F, V), SlotList, NewList).

% add_slot predicate deletes the old slot and rebuilds the
new slot list
```

```
add_slot(req(T, S, F, V), SlotList, [S-FL2|SL2]) :-
delete(S-FacetList, SlotList, SL2),
add_facet(req(T, S, F, V), FacetList, FL2).

add_facet(req(T, S, F, V), FacetList, [FNew|FL2]) :-
FX =..[F, OldVal],
delete(FX, FacetList, FL2),
add_newval(OldVal, V, NewVal),
!, FNew=..[F,NewVal].

add_newval(X, Val, Val) :- var(X), !.

add_newval(OldList, ValList, NewList) :-
list(OldList),
list(ValList),
append(ValList, OldList, NewList), !.

add_newval([H|T], Val, [Val, H|T]).

add_newval(Val, [H|T], [Val, H|T]).

add_newval(_, Val, Val).


% delete_frame predicate

del_frame(Thing) :-
retract(frame(Thing, _)).

del_frame(Thing) :-
error(['No frame', Thing, 'to delete']).

del_frame(Thing, UList) :-
old_slots(Thing, SlotList),
del_slots(Thing, UList, SlotList, NewList),
retract(frame(Thing, _)),
asserta(frame(Thing, NewList)).

del_slots([], X, X, _).

del_slots(T, [U|Rest], SlotList, NewList) :-
prep_req(U, req(T, S, F, V)),
del_slot(req(T, S, F, V), SlotList, Z),
del_slots(T, Rest, Z, NewList).

del_slots(T, X, SlotList, NewList) :-
prep_req(X, req(T, S, F, V)),
del_slot(req(T, S, F, V), FacetList, NewList).

del_slot(req(T, S, F, V), SlotList, [S-FL2|SL2]) :-
remove(S-FacetList, SlotList, SL2),
del_facet(req(T, S, F, V), FacetList, FL2).
```

```
del_slot(Req, _, _) :-
error(['del_slot - unable to remove', Req]).

del_facet(req(T, S, F, V), FacetList, FL) :-
FV=..[F,V],
remove(FV, FacetList, FL),
!.

del_facet(req(T, S, F, V), FacetList, [FNew|FL]) :-
FX =..[F, OldVal],
remove(FX, FacetList, FL),
remove(V, OldVal, NewValList),
FNew =..[F, NewValList],
!.

del_facet(Req, _, _) :-
error(['del_facet - unable to remove', Req]).
```

- Rule Interpreter (Durkin, 1994)

```
:-op(800,xfx,==>).          % used to separate LHS and RHS
                            %      of rule
:-op(500,xfy,:).            % used to separate attributes
and values
:-op(810,fx,rule).          % used to define rule

go :-
     call(rule ID: LHS ==> RHS),
     try(LHS,RHS),
     write('Rule fired '),write(ID),nl,
     !,go.
go:-
nl, write(done), nl.


try(LHS, RHS):-
match(LHS),
process(RHS, LHS).

match([]) :- !.

match([N:Prem|Rest]):-
!,
(fact(Prem); test(Prem)), % a comparision test rather than a
fact
match(Rest).

match([Prem|Rest]) :-
(fact(Prem); test(Prem)), % condition number not specified
match(Rest).

test(X >= Y) :- X >= Y, !.
test(X = Y) :- X is Y, !. %  % use = for arithmetic
```

```
% test(X # Y) :- X = Y, !. % use # for unification

test(member(X, Y)) :- member(X, Y), !.

test(not(X)):-
fact(X),
!, fail.

process([],_) :- !.

process([Action|Rest], LHS):-
take(Action, LHS),
process(Rest, LHS).


take(retract(N), LHS) :-
(N == all; integer(N)),
retr(N, LHS), !.

take(A,_) :- take(A), !.

take(retract(X)):- retract(fact(X)), !.

take(assert(X)) :- asserta(fact(X)), write(adding-X), nl, !.

% take(X # Y) :- X=Y, !.

take(X = Y) :- X is Y, !.

take(write(X)) :- write(X), !.

take(nl) :- nl, !.

take(read(X)) :- read(X), !.


retr(all, LHS) :- retrall(LHS), !.

retr(N, []) :- write('retract error, no'-N), nl, !.

retr(N,[N:Prem|_]) :- retract(fact(Prem)),!.

retr(N, [_|Rest]) :- !, retr(N, Rest).

retrall([]).

retrall([N:Prem|Rest]) :-
retract(fact(Prem)),
!, retrall(Rest).

retrall([Prem|Rest]) :-
retract(fact(Prem)),
!, retrall(Rest).
```

```
retrall([_|Rest]) :- %must have been a test
retrall(Rest).
```

- Query Planner (for constraint satisfaction)

```
query_planner(Keyword, ConstraintNameList,
ConstraintValueList, ConstraintSatisfactionMethod) :-
get_frame(Frame, [keywordlist-Keyword,dependencylist-
DependencyList]),
make_application_list([Frame],DependencyList,ApplicationList
),
satisfy_constraint(ApplicationList, ConstraintNameList,
ConstraintValueList,ConstraintSatisfactionMethod).


make_application_list([],[],[]).

make_application_list(List,'None', List) :- !.

make_application_list([List|ListRest],
[[[DependencyApplication], [WrapperName]]|Rest],
                [List,DependencyApplication|Z]) :-
make_application_list(ListRest, Rest, Z), !.

satisfy_constraint([],_,_,[]) :- !.

satisfy_constraint([FirstApplication|RestApplication],
ConstraintList,

ConstraintValueList,ConstraintSatisfactionMethod) :-
check_application(FirstApplication,ConstraintList,
ConstraintValueList,TestSatisfactionMethod),
ConstraintSatisfactionMethod=TestSatisfactionMethod,!.

satisfy_constraint([FirstApplication|RestApplication],
ConstraintList,

ConstraintValueList,ConstraintSatisfactionMethod) :-
(\+check_application(FirstApplication,ConstraintList,
ConstraintValueList,TestSatisfactionMethod)),
satisfy_constraint(RestApplication, ConstraintList,
ConstraintValueList,ConstraintSatisfactionMethod),!.

check_application(_,[],_,_) :- !.

check_application(Application,
[FirstConstraintList|RestConstraintList],

[ConstraintValueList],ConstraintSatisfactionMethod) :-
get_frame(Application, [constraintlist-
ApplicationConstraintList]),
get_constraint_satisfaction(ApplicationConstraintList,
FirstConstraintList,
```

```
                    ConstraintSatisfactionMethod),
check_application(Application, RestConstraintList,
ConstraintValueList,ConstraintSatisfactionMethod).

get_constraint_satisfaction(
[[[FirstConstraintList],[ConstraintSatisfactionMethod]]|_],[
FirstConstraintList],
ConstraintSatisfactionMethod) :- !.
get_constraint_satisfaction(
[[FirstConstraintList],[ConstraintSatisfactionMethod]],First
ConstraintList,
ConstraintSatisfactionMethod) :- !.
get_constraint_satisfaction([First|Rest],
FirstConstraintList,
                    ConstraintSatisfactionMethod) :-
get_constraint_satisfaction(Rest, FirstConstraintList,
ConstraintSatisfactionMethod).
```