An Expert System Assistant for Gathering Expert Knowledge

by

Brodi J. Beartusk

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

December 15, 1995

Author__________________________________
Department of Electrical Engineering and Computer Science

Certified by______________________________
Professor Robert Berwick
Thesis Supervisor

Accepted by ______________________________

An Expert System Assistant for Gathering Expert Knowledge

by

Brodi J. Beartusk

Submitted to the
Department of Electrical Engineering and Computer Science

December 15, 1995

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The Expert System Assistant is a tool for eliciting knowledge from an expert in a problem domain for the purposes of building an expert system. The system incorporates many of the common tools used in knowledge base construction packages and adds additional features and procedures which enable an expert, rather than a knowledge engineer, to input the information necessary for the expert system. The additional features provided by the expert system assistant are grouped into two areas: knowledge base framework additions and default knowledge. The knowledge base framework additions are composed of facilities for dealing with and specifying incomplete knowledge, top-down and bottom-up frame insertion, knowledge correction procedures and knowledge testing procedures. The default knowledge is a minimal set of knowledge which facilitates the expert's effort by providing several frames from which to base new frames, and a number of atomic leaves of the knowledge base tree with which to ultimately describe all of the derived frames. By providing this combination of system utilities and default knowledge, the Expert System Assistant provides a tool for handling expert knowledge elicitation, one of the most difficult and time consuming aspects of building an expert system.

Thesis Supervisor: Robert Berwick
Title: Professor of Computer Science

1. Introduction

One of the most practical applications of artificial intelligence in real world situations has been the development of expert systems. The broad range of areas to which expert systems have been put to use include such things as computer hardware configuration, medical diagnosis, and even more obscure areas such as choosing the correct wine to serve with a specific meal. These expert systems rely on inference engines which are usually rule chainers. These chainers have set of facts and rules called a knowledge base which form the foundation of the expert system. A knowledge base is typically a collection, usually quite large, of both declarative and procedural information which has been gathered from experts in the domain of the expert system. For example, the knowledge base for a medical diagnosis program called MYCIN was collected by interviewing several doctors and specialists in the field and then translating the verbal and written information gained from these experts into a set of rules which could then be used by a rule chainer in the diagnosis of viral infections.

One of the major problems associated with building any expert system is the acquisition of the expert knowledge, generally known as expert knowledge elicitation. To transfer the knowledge of an expert into a set of declarations and rules, experts must either be painstakingly interviewed, wherein the interviewer basically must learn a great deal of the information known by the expert, or the expert must be taught how to encode his or her knowledge using the language of the expert system, a solution which is highly undesirable and usually impractical. Additionally, by definition, an expert is someone who possesses a great deal of valuable knowledge about his or her field of work, and is therefore usually in high demand. Therefore, it is in many cases difficult for the expert to commit the time necessary to be interviewed at length by the constructor of the expert system.

The process of obtaining the expert information and placing it into the knowledge base of an expert system would be much improved if facilities existed in the system used to build the expert system expressly for the purpose of learning new knowledge. For instance, the implementor of the expert system might initiate a segment of knowledge training by telling the system a name for the area of knowledge which was about to be explained. The expert system package might then display a number of cues concerning the piece of knowledge which could be used to complete its knowledge in that area based on both previously gained knowledge and the responses to the cues.

This thesis is an effort to construct a basic system which provides utilities and default knowledge content which will aid the user in constructing a declarative and procedural knowledge base in a restricted subject domain. The target user is an actual expert in the target knowledge domain who is given a minimal description of the process of building expert systems, and who is not expected to have a great amount of experience in knowledge base construction or programming. The system will be capable of starting from an initial state with a minimal set of base knowledge and end up with a usable set of facts and rules obtained from an expert teacher who interacts with it. The structure of mechanisms used will be derived from both standard AI and expert system tools and practices, as well as observations taken from psychological facts and theories in the area of knowledge representation, learning and curiosity.

2. Overview of the Problem Domain

Possibly from the moment people began writing the first programs for computers, people have wondered whether or not it might be possible for a computer to duplicate a human in terms of reasoning and intelligence. This curiosity has resulted in many projects aimed at giving computers "knowledge" such as a person might have, with varying degrees of success. The field of expert system development is founded on earlier research in artificial intelligence, especially in the areas of knowledge representation and computer decision making. The entire field uses many different kinds of technologies and ideas such as semantic nets to capture a dictionary-like knowledge, object oriented knowledge bases, which represent all knowledge in class hierarchies, probabilistic decision trees which make decisions based on the relative probabilities of results from known facts, and neural net systems which can "learn" patterns of input associated with specific outputs.

While expert systems in general have used all of these ideas, either singly or in combination, the overall idea has been the same: to somehow capture the knowledge a person has in a specific subject domain and use the speed and memory capabilities of the computer to either assist an expert in the field or even serve as a substitute for the expert. The following sections discuss the foundations which will be assumed for this thesis project, and explain the ideas that the Expert System Assistant is based on.

2.1 Knowledge Representation

In order to capture the knowledge and expertise of a human being, a computer must have an internal means of containing information which can generally be referred to as "knowledge." Knowledge is a very broad term which is used to describe a variety of basic ideas, as well as combinations of these ideas. Knowledge representation will consist of a set of programmatic classes which will attempt to model some specific knowledge representation. For the purposes of this thesis, the choice of knowledge representation in the expert system development environment is quite important, since in essence Expert System Assistant is a tool which facilitates a computer's learning of a knowledge domain. "The problems of teaching and learning center around the problem of representation. The problem can be viewed as that of getting the knowledge of the topic matter into the student. This requires some understanding of how knowledge is represented within a person's memory structure, "[1]

Knowledge in general can be grouped into several categories based on its content, use and relation to other types of knowledge. It is important to understand and formalize the differences between different types of knowledge in order to define an appropriate computer implementation for each type of knowledge so that the salient features of each is properly expressed. The following are some useful distinctions of knowledge type[2]:

- Procedural knowledge includes the skills an individual knows how to perform. It can be likened to "knowing how" to do something, such as the procedure for tying one's shoes.

- Declarative knowledge equates to "knowing that." It represents surface-level information that experts can verbalize, such as "the sky is blue," or "trees have bark."

- Semantic knowledge represents cognitive structure, organization and representation. This is basically knowledge about words and symbols, their meanings and relationships, and includes things like one's vocabulary and internal understandings of the definitions of words.

- Episodic knowledge is autobiographical, experimental information that a person has collected during their lifetime which is composed of temporally linked episodes of experience. This is basically knowledge which is known and related based on a type of "story telling" of the events which took place in the episode.

- Metaknowledge is knowledge about knowledge, and can be described as a conscious awareness of what we know and how we use what we know.

Many different forms of knowledge representation have been implemented and evaluated over the course of AI history. The problem of constructing a representation which is suitable for all types and applications of knowledge has proved to be quite formidable, if not unachievable without considerably more time, effort and computing time than is readily available to most research projects. Most of the successful attempts at performing general knowledge modeling have been in cases where the domain of knowledge has been restricted to a specific topic, such as medical diagnosis confined to identifying bacterial infections. Although a thorough description of the schemes which have been attempted in recent years is beyond the scope of this thesis, a few basic and important ideas and results are pertinent.

Semantic Nets

Semantic nets are a means of representing knowledge based on *nodes* and *links* between nodes. One of the first uses for semantic nets was an attempt at modeling general human knowledge in a dictionary like fashion. The basic idea is that each node of the semantic net is associated with a word in a dictionary.

[1] Norman, p. 237
[2] from McGraw/Harbison-Briggs, pp. 22 - 24

Links leading from the node have a certain type, such as *is-a*, which signifies that the parent node is a type of the node pointed to, or *has-a*, which signifies that the pointed to node is a part or component of the parent node. Figure 2-1 shows a very small portion of a semantic net pertaining to a Tree.

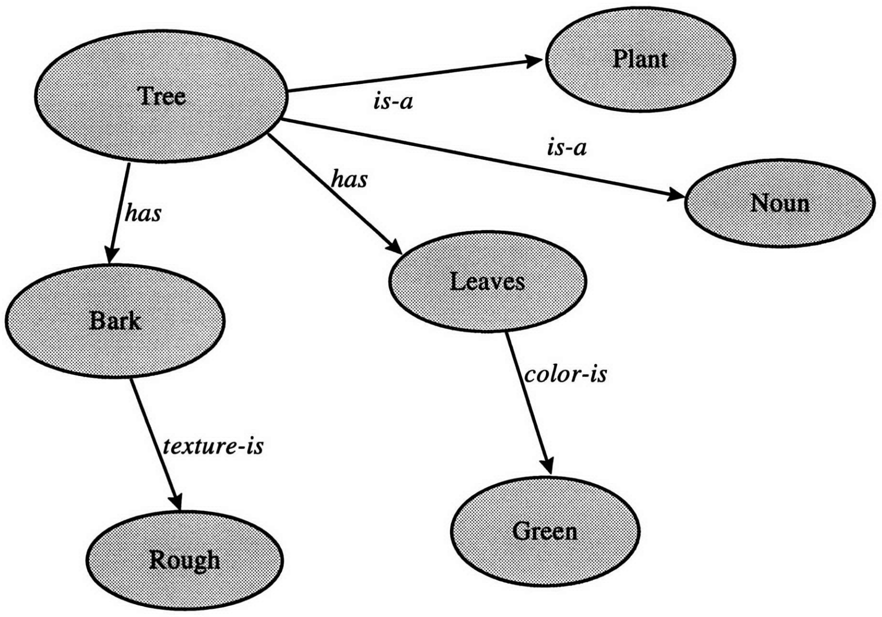


Figure 2-1: Semantic Net

A dictionary based semantic net could in theory be built from a dictionary, creating a node for each entry in the dictionary, and creating suitable links for each type of relationship between nodes encountered in the dictionary. Any particular piece of knowledge in the semantic net is then theoretically described and accessed by following descriptive and explanatory nodes to other nodes. However, there are several drawbacks to this representation.

First of all, it is not clear when traversing the semantic net at what level one should stop following links to other nodes to gain an adequate "description" of what a particular node is, since most nodes in the semantic net have links leading from them; in effect there is no "end" to the link traversing sequence. (Any word described in a dictionary are described using other words in the dictionary!) Second, the semantic net representation does not explicitly define any type of overall object hierarchy in the overall network. Although nodes can have *is-a* links, there are usually no links describing all of the objects that are a type of node - there are no back pointers to the originators of *is-a* links, so it is difficult to classify objects which are not direct descendants of each other. As a last point, the types of links are somewhat undefined, as they are simply a modifier attached to a link which is essentially a convenient name, and has little semantic content relating one node to another. Essentially, understanding the type of link requires another lookup in the semantic net to resolve the descriptors in the link, meaning that link specification falls prey to the same problems the normal resolving of ideas has in a semantic net.

Given the representational problems a semantic net has, it is not usually the representation of choice when developing expert systems. However, it does possess some beneficial traits, at least from an intuitive perspective. The idea of defining a particular concept by a node with an associated name and pointers to other ideas does make sense. Also, people can and do learn new words and ideas from dictionaries, which are basically semantic nets where each entry is a node, and each link is a word or phrase in the definition of the entry, so it would seem that the semantic net could possess at least some of the characteristics of human knowledge representation.

Frames

Frames of one type or another have been used far more often and more successfully than semantic nets in artificial intelligence projects involving knowledge representation. A *frame* is essentially a container for a number of *slots*, which, coupled with the relationship of the frame to other frames in a knowledge base, embody the knowledge represented. Figure 2-2 depicts two frames, one which has a slot which references the second. Each frame has a Name, which identifies the frame, and a list of slots which reference other frames.

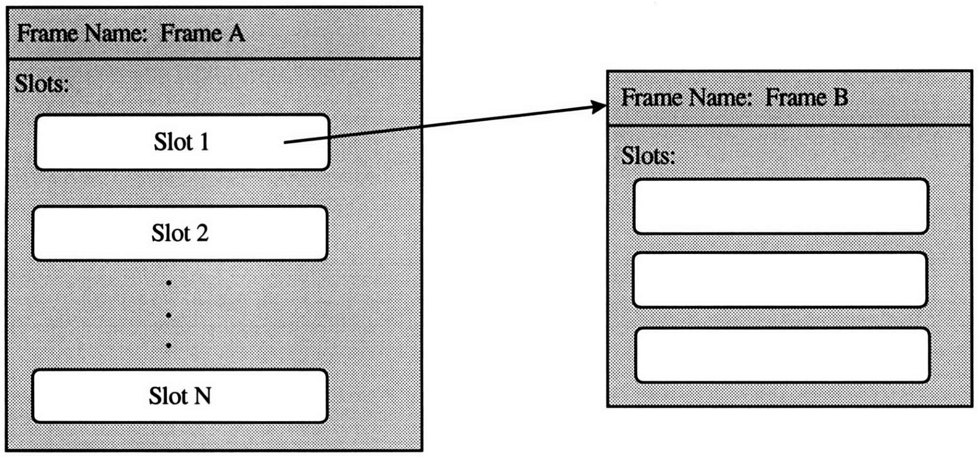


Figure 2-2: Frames

Slots within a frame can be of various forms, depending on the knowledge content they provide. The simplest type of slot is a value-attribute pair, which simply maps a slot's Name with a Value, which can either be a reference to another frame, or a reference to any defined value type such as a number or a string. For more semantic meaning inherent in each slot, additional values may be added, such as a default value, or a link modifier which references another frame defining the relationship between the slot and this referenced frame. The advantage in the second case is that relationships - not only values - could be specified for a slot. In this case, the ability to reference another frame to determine the relationship gives one the ability to define any type of relationship and not have to make do with a predetermined and semantically vacuous named link. Figure 2-3 depicts both a simple value-attribute pair slot and a more complicated slot which has additional variables for referencing a default value and a link type. In this example, the slots are taken from a frame describing a plant. The first slot is the Object Color of the plant, which is Green. The second slot describes an Object Limb of the plant. The current value of the slot is Twig, but if one no value was contained in the Value variable, the Default Value would be assumed to be Branch. The relationship of the slot to the originating frame is described by the Link Descriptor variable, which specifies that Object Limb is related to the frame in which is resides by the relationship Attached To, which in turn is a frame containing the representation of Attached To.

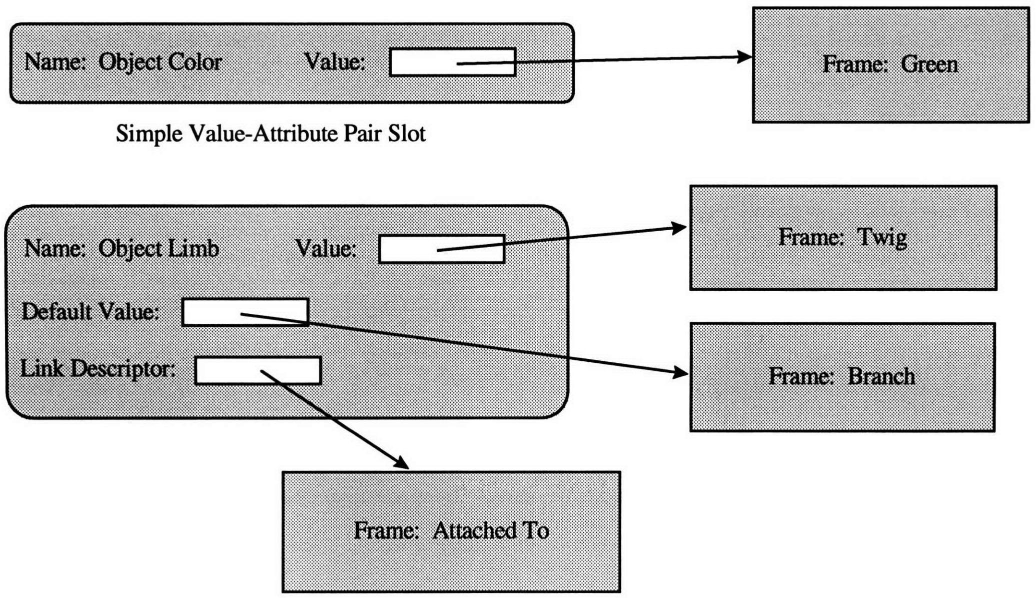


Figure 2-3: Slot Examples

Frames can be useful for many types of knowledge representation, as they are quite general since any particular frame may contain any number of slots representing any number of attributes of the frame. Frames can represent factual knowledge by providing a slot for each descriptive element necessary. Frames can also represent episodic knowledge by providing slots for the events which took place during an episode. In a similar manner, frames can be used as a basis to implement any of the above mentioned types of knowledge, at least when combined with an appropriate mechanism for interpreting the content of the slots.

Object Oriented Knowledge Bases

In developing a representation for some piece of knowledge, it can be quite helpful to group knowledge into *classes* of knowledge, where each element of a particular class shares attributes with others in the same class. By doing this, objects which have similar attributes can be grouped together under a parent class, where the shared attribute specifications are made only once, instead of for one time for each of the children of the parent class. In fact, any attribute or method associated with any class need be specified only once, both making the implementation easier, and allowing the implementation to be much more easily maintained and modified. A tree of classes all derived from a basic global class, usually named *Root*, forms an object oriented knowledge base.

An object oriented knowledge base is commonly used in expert system development tools to encode relevant pieces of knowledge. Each element in the knowledge base is derived from a parent class, and is either a class itself or an instance of a class. Classes are mainly a description of a certain type of object, while instances are considered to be specific individual objects of a certain type of class. Classes may be subclassed, while instances cannot. Classes have attributes associated with them which have values. They also generally have *methods* which are class specific procedures on the classes attribute values. Children of parent classes inherit both attributes and methods from parent classes. Figure 2-4 shows an example of an object oriented knowledge base tree containing six classes derived from Root. Each box which has arrows originating from it is the parent of a class to which it points. Each child class inherits the attributes of the parent class, as in below, where Vertebrate inherits Mobility from Animal.

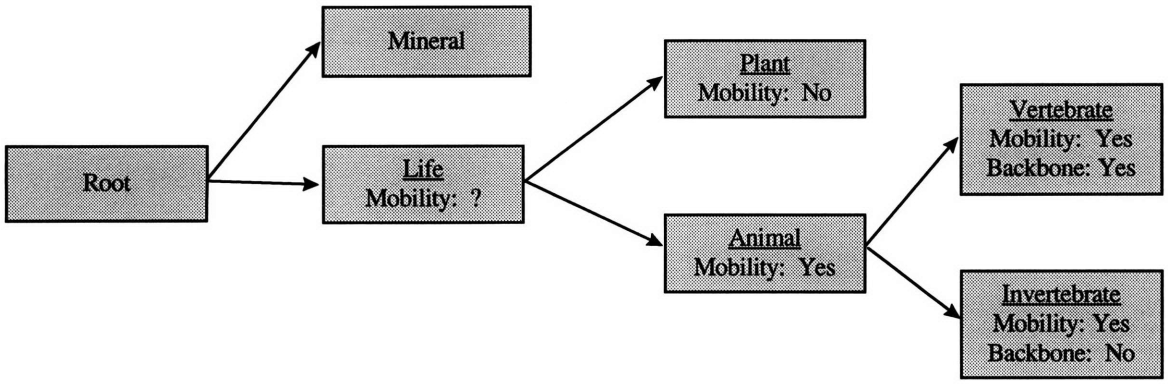


Figure 2-4: Object Oriented Knowledge Base Tree

Object oriented knowledge bases appear to be one of the most appropriate means for capturing general knowledge, especially when the class elements of the knowledge tree have a fair amount of representational power on their own. For instance, when each class in the tree is represented as a frame, the tree adds a great deal of ease to both the construction and modification of the elements of the tree through attribute and method inheritance. Additionally, the tree implicitly captures relationship knowledge about frames which have direct ascendants, allowing for heuristics based on broad classes of objects instead of on just particular frames. It is this combination of a knowledge base tree with frame elements which will be used in the implementation of the Expert System Assistant.

2.2 Expert Systems

An expert system is a computer based tool which somehow encapsulates knowledge from a human expert in a specific domain. One introduction to Artificial Intelligence[3] "defines *expert systems* as computer programs that contain both declarative knowledge (facts about objects, events, and situations) and procedural knowledge (information about courses of action) to emulate the reasoning processes of human experts in a particular domain or area of expertise."[4] As such, expert systems are commonly used in situations where experts are rare and whose time is therefore very valuable, or in situations where a large number of technically complicated problems can arise often and repeatedly, such as a manufacturing plant.

Expert systems are composed of at least three basic entities: the knowledge base, an inference engine, and a user interface.[5] The knowledge base contains the facts and rules which make up the expert knowledge, and can be organized in many forms, including the object oriented knowledge base described previously, but also including databases of rules keyed on the type of their inputs, or trees of rules linked to other rules by which rule's predicates affect the arguments of others. The knowledge base is used in coordination with the inference engine, which takes as a foundation the facts and rules contained in the knowledge base. When the user inputs new information via the user interface, the inference engine then uses the new data and any rules which pertain to it or the result of application of rules on the new data to further fill out pieces of the knowledge base and to come to conclusions or *goals* which are usually defined in the knowledge base as the eventual ending conditions for the inference engine.

In addition to simply using the inference engine to come up with results based on the knowledge base, once a condition has been achieved through rule chaining, most expert systems are capable of relaying to the user the particular chain of rules which were applied to obtain the result. This effectively gives the user a logical reason that the particular result came about from the contents of the knowledge base and his or her input, and provides a means for determining the validity of the machine's results.

Expert systems have been used quite successfully in many different applications, ranging from medical diagnosis programs like MYCIN, which diagnosed bacterial infections based on rules of thumb and user input, to a system built by Digital Equipment Corporation which helps configure the parts needed for complex computer installations.

[3] Mishkoff, 1985
[4] McGraw/Harbison-Briggs, p. 3
[5] McGraw/Harbison-Briggs, p. 4

2.3 Knowledge Elicitation

The most complex feature of a current expert system is the knowledge base of facts and rules it contains, as the trend over the last decade has been to concentrate on expert knowledge modeling more than on developing new heuristic engines. Typical knowledge bases for restricted knowledge domains range from hundreds to thousands of rules and facts, all of which must be determined from interaction with an expert and translated into the language of the expert system knowledge base. "At the beginning, expert systems were developed to fit a particular domain and a certain task, but later on researchers aimed at eliciting and representing the domain-related knowledge for a given interpreter with fixed control structure. Filling the expert system shell has been found to be the major problem when applying expert systems."[6]

Many factors contribute to the difficulty of obtaining expert knowledge. First of all, due to the fact that the knowledge which the experts have is considered valuable enough to devote the resources necessary to build and expert system, the expert's time is at a premium. In fact, one of the major reasons for implementing experts systems is to replicate the expert's knowledge in order to provide greater availability of the knowledge, even in places where the expert is not present physically. This can make it quite difficult for the expert to find the considerable amount of time he or she needs to make available to the knowledge engineer for knowledge elicitation.

Once the time for meetings and interviews between the expert and the knowledge engineer has been found and reserved, the next major obstacle is the interview process itself. The fundamental problem is that, while the expert may be very knowledgeable about the subject domain, he or she may not be very knowledgeable about the means for conveying that knowledge explicitly to another person. The following comes from a recent book on knowledge acquisition:[7]

1. Experts supply incomplete and inconsistent knowledge, because they instantly develop a naïve theory about their competence when they are being interviewed.
2. The expert's descriptions of their actions differ from their observed behavior since they tend to verbalize their knowledge but not the competence which underlies their actions.
3. Expert's modify their knowledge during the knowledge acquisition process; they adapt it to the model which has been interactively created with a knowledge engineer, because the modeling process will perhaps be the first occasion for them to develop a distinct model of their competence.

The main problem with the knowledge elicitation process lies in converting the expert's knowledge which he or she obviously knows very well, into a representation which the expert system tool can accommodate,

[6] Morik, Wrobel, Kietz and Emde, p. 1
[7] Morik, Wrobel, Kietz and Emde, p. 14

while at the same time not putting too many restrictions on the types of questions asked of the expert or of the types of responses required of the expert. The conflict exists between the computer representation of the knowledge, which must be very well structured and precisely defined, and the expert's concept of the knowledge, which, while very complete and adequately organized for the purposes of the expert, is simply inappropriate for direct input into the knowledge base without considerable processing by the knowledge engineer.

A number of possible accommodations to the problem of knowledge acquisition are possible. One possibility, and one which should not be neglected, is to place more of the burden on the knowledge engineer and require them to investigate and utilize the latest ideas and techniques regarding human learning and teaching.

Another approach, and the one this thesis is directed towards, is to move the knowledge engineer further out of the knowledge acquisition process and provide the expert with the software and knowledge necessary for them to convey their knowledge directly into computer form. However, for this to be a viable idea, the machine must absorb most of the burden of learning, and shouldn't require the expert to learn many new concepts and procedures which have nothing at all to do with their area of expertise or even interest.

3. System Design Requirements and Methodology

The driving purpose behind the construction of Expert System Assistant is to ease many aspects of the construction of expert systems. Therefore, the system is based on a core of common knowledge base tools, upon which is built an extra set of enabling tools and default knowledge. In the design of the system, several requirements as to functionality and features were determined prior to making firm decisions about the application structure. Among these requirements were the following, which are discussed in further detail later in this section:

- basic features common to many graphically based knowledge base construction tools
- a common means of representing knowledge which is built into the system in a logical way
- a set of extra features which enables the system to be more easily used by someone with very little experience in building expert systems

Since knowledge induction capabilities such as an inference engine are out of the scope of this thesis, they are not a part of the system. This makes conformity to basic features essential to the entire idea behind Expert System Assistant, since to be useful it must be able to accumulate knowledge in a format which can be easily converted to other knowledge base storage formats. Expert System Assistant is meant to be used mainly in the knowledge acquisition stage of the development process. However, this is a tradeoff which was made more in the interest of complexity and the time involved in development. The addition of rule chaining capabilities to Expert System Assistant would not only have made it a complete package for developing expert systems, but might also could have offered some additional facilities to collect and test expert knowledge.

3.1 Standard Features

Basic functionality for the Expert System Assistant is essentially equivalent to that which is provided by many mainstream development tools. It includes the ability to create new knowledge bases, add and remove elements to the knowledge base, modify elements in the knowledge base, and the ability to save the contents of a knowledge base and load it again at a later date of editing. Additionally, the system must offer convenient and useful facilities for examining the contents of the knowledge base and important features of the knowledge base.

All of the features were required to be readily available in a simple manner in the interface. Since Expert System Assistant was developed in and for the Windows 95 environment, whenever possible common controls were used. Using common controls shortens the time necessary for a user to become acquainted with a software package and provide easier use once they become familiar with the system. For instance, in saving and loading knowledge bases, the standard Windows Save and Load dialogs were used which are used by most Windows applications and the operating system itself, and which are already familiar to the user. Standard controls available in the Microsoft Foundation classes were also used for familiarity and features. These include pulldown menus to access all functions provided by the system as well as dockable toolbars with custom bitmap buttons and popup help tabs for commonly used functions.

3.2 Knowledge Representation

Expert System Assistant needs to be capable of representing most of the common data types offered by conventional expert system development tools. This will ensure that it is possible to write translation routines which convert Expert System Assistant's internal knowledge base representation into a format readable by other packages which posses an inference engine. This constraint will define the structure and content of the knowledge base building blocks as used by the system. Additionally, this set of data types should be versatile enough to represent most or all of the pertinent information necessary for a computer to become an "expert" in the knowledge domain. "A major prerequisite for the support of knowledge acquisition is to define an adequate representation system."[8] The following sections describe the targeted types of knowledge which will be supported.

[8] Morik, Wrobel, Kietz and Emde, p. 5

Facts

A fact in Expert System Assistant is a piece of knowledge referenced by a name which is unique in the knowledge base. Facts are frame-like entities which are initially derived from Root in the knowledge base, and which therefore have a parent and may have children which inherit their properties. Facts have a number of properties associated with them which are essentially slots. Each property has a value, which is either a reference to another piece of knowledge in the knowledge base, or simply a stored value such as a number or a string. The number of properties possessed by a particular fact is not limited. Facts are the most loosely structured components of the knowledge base and as such are the most versatile in terms of user definability.

Rules

Expert System Assistant has the ability to encode rules which are relatively basic in format. These rules consist of simple conditions coupled with predicates which are to be performed if the condition evaluated to TRUE. The format of a rule in the system is pictured in Figure 3-1. The rule consists of four basic parts: a condition list which references specific slots in facts and an associated value for that Fact:Property, a Boolean term which uses the numbered conditions described in the Condition List which evaluates to TRUE when the rule should be applied, a Predicate List which matches specific Properties in certain Facts to a Value to which the Property should be set if the rule is applied, and finally, a Rule Name which uniquely identifies the rule in the knowledge base.

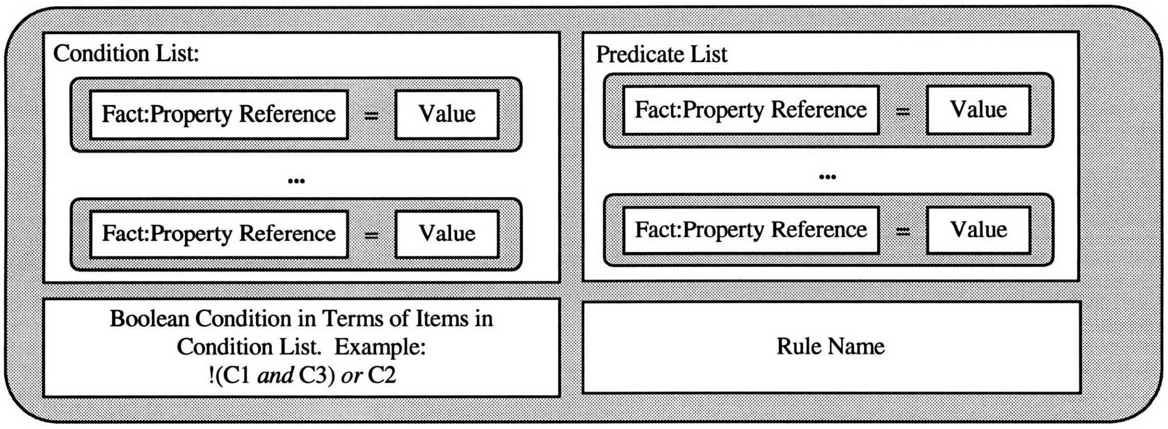


Figure 3-1: Supported Rule Format

This format for a rule provides all of the information necessary to describe a rule in most available expert system development tools, while at the same time providing some extra access to object information which Expert System Assistant needs.

Goals

The knowledge inference engine in any expert system development tool requires a goal to work towards at each step of its chaining process. The goal specifies a set of conditions which should be worked towards, such as the value of a property within an instance of a fact being set to a value as the result of the chaining. The rule chainer picks rules from the knowledge base depending on whether the chaining is forward or backwards, and based on the goal which the chainer is given. The form of goal supported by Expert System Assistant is similar to the condition portion of a rule in that it has a Condition List. However, as shown in Figure 3-2, no value is associated with each element in the condition list. Therefore, what is implied by the presence of each element is that there is in fact a value associated with the Fact:Property. As with Rules, each goal has a name which is unique in the knowledge base.

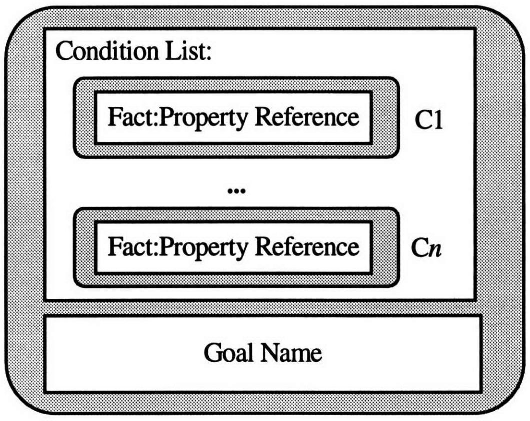


Figure 3-2: Goal Format

3.3 KB Construction Enabling Features

In addition to the knowledge base construction tools which are normally available to a user in any of the available expert system development tools, Expert System Assistant needs to provide an additional set of tools which facilitate the process. Ideally, this set of tools would allow an expert with minimal experience in building expert systems to easily construct a knowledge base from their expertise. This section will discuss several techniques from learning theory and knowledge acquisition which will be used to form some of the design criteria for Expert System Assistant.

The actions of Expert System Assistant can be viewed as a type of teaching and learning which are done by the expert and the computer, respectively. As the most basic starting point, two foundation conditions must be met. The first, which is met by the knowledge representation requirements outlined previously, is the ability of the system to represent the knowledge being learned. Second, the teacher, which is in this case the user/expert, must have at least a simple concept of the underlying representation used by the student, which is the computer. One of the major problems in translating the model which exists in an expert to the representation system used by the computer are the differences between the human method of representation and the machine method. The success of expert system development tools based on a framelike representation system may be somewhat due to the similarity of frames to some of the structures the human mind uses to encode knowledge. "Current work in psychology, linguistics and artificial intelligence converge in suggesting that experiences are coded by rather elaborate mental entities: frames, scripts and schemata."[9] The extra features of Expert System Assistant are aimed at easing the conversion of the complex data structures present in the expert's mind to the simpler frame system used by the application.

As a first step, the nature of the extra tools should be defined. For example, should the entire process be completely directed by the machine or by the user? The case of the entire burden of learning control lying on the user is essentially that of existing tools, where the user is only given the facilities of adding elements in a restricted manner to a knowledge base, which is not very well suited to a non-knowledge engineer. However, the other extreme, where the computer is responsible for driving the entire process, is also not very desirable. "The creation of a knowledge base merely on the basis of machine learning procedures is not desirable, as the users (the so-called knowledge engineers) like to add or modify facts and rules themselves."[10] Because of this, the extra features in Expert System Assistant should concentrate on exploiting possible synergy between the expert and the machine, rather than relying completely on machine learning techniques.

The next step in defining a set of helpful tools is to consider the input that would be available to them, and also what inputs *should* be available to them. Obviously, the tools will have the current contents of the knowledge base available to them. However, these contents themselves are not sufficient for an adequate learning mechanism. "Merely possessing a certain symbolic expression that encodes semantic content is, by itself, insufficient to produce behavior ... What is needed is a set of mechanisms to make the system run."[11] This implies that any learning tools must make use not only of semantically encoded content, but must also make use of mechanisms which use user input to modify learning behavior. This is not to say that the user must be involved in every step of a particular tool's operation, just that the input of the user can be every bit as important as the machine tool in some instances and should therefore not be neglected as a resource.

The tools will leverage the superior capabilities of the computer in certain areas like memory speed and knowledge set working size to enable the user to more easily develop and modify large knowledge bases. "In order to automate an on-going activity of knowledge acquisition and knowledge enhancement, a learning system has to recognize by itself the need when and where to improve the knowledge base and at

[9] Millward, p. 261
[10] Morik, Wrobel, Kietz and Emde, p. 2
[11] Pylyshyn, p. 30

least to provide a few suggestions on how to achieve a better knowledge base."[12] This statement suggests that a couple of the means to leverage the specific abilities of the computer in 1) identifying places to improve the knowledge base, and 2) based on the area where improvement is needed, suggest a means to make an improvement. Based on this, at least some of the knowledge base construction enabling tools offered by Expert System Assistant will offer this type of functionality. Since it is relatively easy for a computer to search for a given condition once it is adequately defined and given that the data set is stored in an efficient format, the major considerations in defining such tools are in defining conditions in the knowledge base which are "good" candidates for knowledge base improvement.

When developing a scheme for identifying likely candidate areas of the knowledge base to make improvements, Expert System Assistant will have available only the contents of the knowledge base and the most recent actions of the user as a guide. The learning features might then take a form which operates in two stages: a data examination stage which is independent of immediate user intervention, and a second stage which determines action after presenting the user with the results of the first stage. This model fits well considering the expected amount of knowledge in the subject domain which the system can be expected to possess at any point in the learning process - basically, very little.

During most of the learning cycle, the application can be considered to be a *naïve organism.* "A naïve organism can be though of as one without concepts appropriate for the environment it finds itself in ... The naïve organism is generally data driven rather than goal directed."[13] Basically, since an undeveloped knowledge base does not usually contain enough information to define solid and useful goals, a naïve organism usually seeks to add data to its knowledge base in any way it can, but preferably based on the data already contained within it. Therefore it will use known data to the best of its abilities to seek out more data, which should be a requirement for knowledge acquisition enabling tools in Expert System Assistant.

3.3.1 Incomplete Knowledge Entry

As mentioned earlier, an expert typically will actually develop a model of his or her expertise on the fly during the course of interviewing. The relaying of undeveloped model information during the construction of a knowledge base will certainly be fraught with a great deal of uncertainty at any particular stage in the process. Because of this, Expert System Assistant will be faced with the problem of dealing with a knowledge base which is the result of this type of "sloppy modeling." Sloppy modeling is a technique which does not place tight accuracy constraints on a model in any development stage. Rather, it concedes that there will always be parts of the model which are incomplete, inconsistent, or even completely

[12] Morik, Wrobel, Kietz and Emde, p. 2
[13] Millward, p. 266

incorrect, but that nevertheless, the model can still be reasonably accurate when taken as a whole. "The basic set of assumptions summarized by the term *sloppy modeling* are:

- A model is always incomplete, on the one hand, because the domain changes, and on the other hand, because our knowledge of the domain changes, too. Modeling is an infinite process.
- A model is more or less adequate, on the one hand for the uses of the knowledge, and on the other hand for the tasks the model is to perform. Modeling is an approximate process, which is expected to provide increased adequacy.
- A model is to be consistent in itself and reflect the present state of the art. This cannot be achieved at once. Modeling is a deficient process."[14]

In order to be useful, the knowledge acquisition tools provided by Expert System Assistant must accommodate a sloppy modeling process.

One of the main difficulties which standard expert system development tools have with a sloppy modeling process is their inability to deal with incomplete knowledge. This comes about as a result of a couple of assumptions about the knowledge base which are only valid if one requires that the concept of the knowledge base is completely developed. First, most tools assume that the overall structure of the knowledge base has been determined *before* any information is put into the knowledge tree. This generally means that these tools provide only a limited set of restricted tools to add or modify elements of a knowledge base, at least when it comes to things like overall placement of a class in the knowledge base tree; basically, these tools assume that since a user placed a class or frame at a particular location, then that user was absolutely sure that that element did indeed belong where they placed it. Obviously, this assumption does not mesh very well with sloppy modeling, where the entire structure of the knowledge base may frequently need to be changed as a model develops.

Another assumption that is commonly made in expert system development tools, and which is also related to the first assumption, is that references inside of elements of a knowledge base to other elements of the knowledge base are to other elements which are already in the knowledge base. Because of this, whenever a variable or slot is added to a member of the knowledge base which has as a value a reference to another element in the knowledge base, the type of the referenced value must be explicitly stated as belonging to a particular class which has already been defined. This places some restrictions on the order in which elements of the knowledge base must be defined: any object referenced by a different object must be defined *before* the second object. Again, this restriction creates problems given the sloppy modeling paradigm.

[14] Morik, Wrobel, Kietz and Emde, p. 14

To be useful to a sloppy modeling process, Expert System Assistant must be able to deal with incomplete knowledge in many forms. Two requirements for dealing with incomplete knowledge can be borrowed from similar work done on the MOBAL system[15]. "On the whole, the system:

- has to accommodate the continuous increase in knowledge, and
- must be operational for use even if the model is incomplete"

Expert System Assistant addresses these issues by providing a great deal of flexibility in both the knowledge base structure and content.

Expert System Assistant has several methods which provide knowledge base structure flexibility. First of all, it allows the placement of new elements of the knowledge base in almost any position, instead of only as a child of an element already in the knowledge base. This allows elements which are either initially forgotten or even unknown to be placed easily into the knowledge base at any time in the development of the model. Facilities also exist within Expert System Assistant to easily rearrange elements of the knowledge base after they have already been placed in the knowledge tree structure. This ability facilitates a sloppy modeling process, allowing the user to place a piece of knowledge in the knowledge base based on an initial assumption which might turn out incorrect at a later point, when the mistake can then be easily corrected.

Expert System Assistant also allows the user to put references to unimplemented objects in any piece of the knowledge base. This is useful when defining a particular class of object because it does not require that all of the components of the object be previously defined. The user then has a great deal of freedom with the order of definitions of objects in the knowledge base. This allows the complete definition of any piece of the knowledge base without having to worry about whether or not the descriptive components of the variables contained within the knowledge being encoded have been implemented themselves or not. The main advantage to this is that the user can describe anything that he or she feels is pertinent at that point of the knowledge base development, using any terms they wish, and fully carry out his or her description of the object without having to stop and implement other objects. This can be especially useful considering some findings on the nature of expert knowledge which show that expert knowledge is typically stored in "chunks or larger units of knowledge rather than individual fragments."[16] It is much more convenient to finish the description of a large piece of knowledge in one unbroken session, rather than being diverted many times to define subparts of the whole. Note that while this facility adds a great deal of convenience to the user, it also means that the system must somehow keep track of any unreferenced values in the entire knowledge

[15] see Knowledge Acquisition and Machine Learning, 1993
[16] LaFrance, p. 65

base and must also be able to inform the user of unreferenced values so that they may be implemented later on.

3.3.2 Mistake Correction

Since Expert System Assistant must be able to deal well with a sloppy modeling approach, it must allow for easy mistake correction and revision of the knowledge base when errors, inconsistencies or new data are identified. Errors can be anything from a misspelled element or frame name, to a frame which has been inserted at an incorrect place in the knowledge base tree. Inconsistencies in the knowledge base can come about through a variety of simple and obscure events, from mistakenly defining identical knowledge concepts using two different names, to inadvertently creating a loop or cycle in the class hierarchy when moving a branch of the knowledge base. New data often must be inserted into a knowledge base at a place other than a leaf of an existing class, or may require that certain sections of a knowledge base be rearranged in order to more accurately reflect the knowledge domain which is being modeled. Expert System Assistant must offer easy tools to deal with and even identify these problems.

Editing mistakes, such as misspelled words and such, are only interesting here because of Expert System Assistant's required capability to allow unreferenced values in slots of the knowledge base. Since these unreferenced values are tracked by name, a misspelling can complicate the process by listing entries as an undefined value which is to be later defined in a case where the name is simply misspelled. This could possibly lead to some confusion when a user defines a value which has been previously referenced. He or she would expect the unreferenced status to disappear, but the misspelling would prevent this. Solutions to this type of problem are more linguistic in nature, and are out of the scope of this thesis, but nonetheless interesting and practical, as the implementation of Expert System Assistant is founded on giving unique names to objects in the knowledge base. One possible solution might be to use a dictionary to check each word as it was inputted to the system, and if it wasn't found, notify the user. However, many scientific and technical terms are not found in common dictionaries, which would limit the usefulness of this approach. Of course, the system might incorporate custom user dictionaries which would add new entries based on user validated words put into the knowledge base.

The problem of correcting topological mistakes is mainly due to the sloppy modeling process. When the expert's internal model is either refined or modified so that it is inconsistent with the contents of the knowledge base, parts of the knowledge base tree must be rearranged or deleted from the knowledge base. Deletions can take the form of either deleting an entire branch of the tree, or simply removing a single frame which is in the simplest case a leaf of the knowledge base tree, or, more complicated, somewhere in the middle of the tree. Expert System Assistant must be able to handle both cases, and must be able to query the user for the procedure for dealing with the children of the deleted frame, in the second case. Rearranging the knowledge base will be restricted to simply moving an entire branch of the knowledge base so that it has a different parent. The effect will be that the selected frame and all children will be removed from their current position and added as a child branch to their selected destination.

3.3.3 Base Knowledge

One of the major difficulties when building a knowledge base lies in deciding on a set of primitives to define which will aid in the later definitions of other objects. For instance, when developing an expert system to identify plants based on their physical characteristics, it might be useful to define a whole series of colors which could then be used in the frames defining specific types of trees. *Base knowledge* is a term defined here which is a set of default knowledge in a knowledge base which is provided to the user of the system for the purpose of providing a common foundation on which to begin the process of knowledge elicitation. It can be thought of as the knowledge in the knowledge base for which there need not be any further derivation or definition - it just *is*; alternately, base knowledge may be thought of as those concepts which are often referenced by other higher level concepts, but which are either difficult to put into words and unnecessary to further define because they are foundation ideas, like numbers or colors.

"The notion of a discrete atomic symbol is the basis of all formal understanding. Indeed, it is the basis of all systems of thought, expression or calculation for which a notation is available."[17] Base knowledge in Expert System Assistant will be a set of these basic discrete atomic symbols which may be used in combination to define the rest of the concepts in the knowledge base, much as algebraic expressions can be written using a common basic set of symbols to represent a limitless number of functions. It should be noted that the actual definition of "atomic" is dependent upon the application domain. In one instance, say, leaf identification, defining a series of atomic simple colors may suffice, while in another application, say, photography, color may actually be composed of atomic quantities like saturation, brightness and hue. For Expert System Assistant, the burden of defining a suitable set of base atomic quantities will fall on the knowledge engineer instead of the expert.

Although the choices for items which are base knowledge do depend heavily on the knowledge domain, some general guidelines can be derived considering the nature of such knowledge. At the very most basic level, base knowledge can be concepts which are derived wholly or mostly from the senses. Obviously, a meaningful computer based representation of such a construct would be very difficult, as it would be hard to quantify the actual sensation elicited by say, the retina and the resulting combination of nerve firings based on a certain light pattern falling upon it. Base knowledge of this type can be thought of as coming from a person's senses, which are essentially transducers from the physical environment to electrical signals which the brain interprets as it can. "A transducer is just another primitive operation, albeit one responsive

[17] Pylyshyn, p. 51

primarily to the environment rather than the cognitive process ... At some level the following *must* be the case: An organism's contact with the environment must, at some level, be decoupled from its cognitive processes; otherwise the organism has no stable base of causal interactions with the world from which to develop veridical perceptions"[18]

Shared knowledge, such as the atomic definitions of quantities like emotion or aesthetic feel are another good candidate for base knowledge. Such knowledge is easily understood when mentioned using descriptive words, but is really hard to quantify if it is assumed that the listener has no internal notion of them. This is a quality that such things share with sensations such as color and smell, wherein one essentially learns to associate particular words such as "blue" or "rose-scented" with particular remembered encounters with the sensation evoking objects. Shared knowledge then is a set of things which most people are assumed to experience in a similar manner and have names for, but which is usually defined and learned only by direct contact with both the knowledge and the name for the knowledge. Shared knowledge is generally atomic in nature and is not composed of other definable named pieces of knowledge in conjunction.

Convenient knowledge can be defined as a set of commonly used terms and ideas in a knowledge domain which are commonly used without a definition without losing a great deal of representational ability. Such knowledge is rather high level and is non-atomic. However, the knowledge domain for which such knowledge is used to describe is at an even higher level, making the availability of the high level base knowledge convenient for the expert for use in describing his or her knowledge. As an example, consider an expert system built to solve geometric proofs. It would be very useful to include as base knowledge the Pythagorean Theorem which could then be used without proof in the system. Obviously, this theorem is not atomic, for it itself is a geometric theorem which can be proved. However, this system would be aimed at the more restricted knowledge domain of proving higher level theorems than the Pythagorean Theorem which could be based on the Pythagorean Theorem. By making the theorem available to the knowledge expert, the task of relating the knowledge becomes much easier.

In essence, base knowledge is a form of layman knowledge which makes the task of translating expert information into computer form easier by providing a common, understandable and convenient base from which to start knowledge descriptions. In the best case, the base knowledge provided to the expert in Expert System Assistant would be precisely the intersection between the knowledge of the expert and the knowledge of the knowledge engineer providing the base knowledge. This condition would maximize the input of the knowledge engineer and minimize the work needed of the busy expert to define basic concepts. Additionally, since base knowledge is knowledge which might be possessed by most people in some form

[18] Pylyshyn, p. 155

or another, it may be very reusable. Certainly base knowledge which concerns human perception would be quite easily shared by many unrelated knowledge domains. More specific and higher level base knowledge, while not applicable in general to many systems, would at least be applicable to families of knowledge, such as mathematics, or biology.

3.3.4 Knowledge Testing

Since Expert System Assistant is intended to be used by an actual expert in the knowledge domain rather than a knowledge engineer, it is important to provide facilities to the user to verify that the model contained within the system is a good match to his or her own personal model. If the system is to be used successfully by such a knowledge engineering novice, extra tools must be provided which can be used in place of a knowledge engineer's experience with knowledge models. These tools are meant to partially replace some of the judgment of a knowledge engineer that applies to confirming that the knowledge base at any particular stage of development is an appropriate model. The task of quantifying all of the judgment knowledge of a knowledge engineer is quite complicated, and in certain cases is even impossible, since a great deal of personal opinion and previous experience plays a part in a knowledge engineer's final decision as to the validity of a representation. The tools provided by Expert System Assistant will therefore be a small set of tools which seek to provide simple consistency checks of specified frames in the knowledge base.

Visual Topographical Verification

Almost all expert system development shells provide a means for visualizing the contents of the knowledge base, usually in the form of a graphical tree of knowledge base elements. This visualization serves several important purposes. First, it provides immediate feedback to the user of the results of an operation which changes the structure of the knowledge base in any way, which is of course important because the user absolutely must be sure that the action he or she has just performed is in fact what he or she intended; note that this is especially important in the case of a novice user, since he or she would be likely to be unsure about the effects of many operations. A second important use of the topological feedback is that it is useful in determining what changes to make to the knowledge base. By inspecting the overall structure, one may decide to proceed in the knowledge elicitation by noting a portion of the knowledge base which is incomplete or even incorrect.

The main problem when only using topological features for knowledge base verification comes about when the knowledge base becomes even moderately large. While it is generally quite easy to interpret a knowledge base tree when it only has a handful of elements, the problem quickly becomes intractable when the number of elements is such that their display area is larger than the application window display area. At this point, large parts of the knowledge base are not visible concurrently, so that comparisons between the sections are either inconvenient or even impossible. When this occurs, the user must simply be intimately familiar with the structure of the knowledge base to make quality judgments on it, something that may be difficult for users who have not had a lot of experience in developing exert systems.

Property Testing

The ability to test certain properties of elements of the knowledge base adds a great deal of power to a system which relies primarily on visual verification. Property tests allow a user to specify an element of a knowledge base and some optional comparison elements and return to the user some useful information about the requested properties, either by themselves, or in comparison to the properties of another element in the knowledge base.

Almost all expert system development environments provide a means for viewing and editing the property list of any element of the knowledge base. This is the simplest and most direct means of examining testing the properties of a piece of knowledge: all of the available properties of the piece are simply displayed as is. However, other than the descriptive names and types of contents, this inspection does not test semantic knowledge, but rather relies on the judgment of the user to judge the semantic content of the displayed knowledge. While certainly important to provide feedback about the contents of individual elemtents of the knowledge base, more tools are necessary if the system is to be used easily by a novice expert.

Several different types of knowledge content tests can be defined and used as a framework upon which to base useful tests. *List tests* are similar to the simple property inspection mentioned above, and return a subset of the properties of a piece of knowledge which match an optional set of input parameters. This type of test is useful in checking that a knowledge construct's contents at least match what the user expects.

Hierarchy tests are tests which ask a certain question about a knowledge construct in relation to another construct in the knowledge base. For example, a particular tool might be available to which the user could input a frame which is in question, and a frame to which it is compared. The tool might then return the relationship between the two frames: direct descendant, direct ascendant, or relative through a common ancestor in the class hierarchy of the knowledge base. This type of test can be used to confirm the relationship which the user intended between different pieces of the knowledge base actually exists in the proper form.

Comparison tests are tests which compare the properties of specified elements of a knowledge base. The comparison can be based on a variety of methods, from comparing the names of properties of frames, to comparing the actual values of the properties of frames. Additionally, the comparison may be directed to search for either elements which are the same in both frames or different among frames. As an example of a comparison test, consider a situation where the user wanted to test the difference between a Mollusk description and a Fish description in a knowledge base concerning marine life. Such a difference comparison test might return among the set of different properties the differing value of a Has_A_Backbone property of the two frames. If this was not returned among the set of differences, and the user considered the difference important to the knowledge content of the system, then he or she would thereby be made aware of a possible omission of knowledge.

4. Expert System Assistant

Expert System Assistant was developed as an SDI application to run under Windows 95 using Microsoft Visual C++ and the Microsoft Foundation Classes, which provides many useful classes for the user interface and data types. Pictured below in Figure 4-1 is an image of the application when it is first started. The user is provided with a menu and button bar to access the functions of the system, and a single document window in which to view the current selected piece of the knowledge base.

The menu of the application window is used to access pull down menus which in turn allow access to knowledge base functions. Each item in the menu list is a category of function types which grouped together according to their use. The available categories are as follows:

- File: functions which are used to create a new knowledge base, save and load a knowledge base, printing functions and a recent file list which gives easy access to the most recent four knowledge bases which have been edited.
- Edit: contains the standard windows edit functions Copy, Cut and Paste which are useful mostly in the function dialogs.
- View: allows the user to choose the current viewing mode and currently selected objects to view in the knowledge base.
- Knowledge Base: this is a list of the functions available to the user which are used to edit the knowledge base such as Add Frame, Remove Frame, Add Slot, Remove Slot, and Edit Slot.
- Knowledge Test: provides access to functions which assist in the building of a knowledge base such as Is a ___ ?, A ___ is a ...?, and the Difference test.

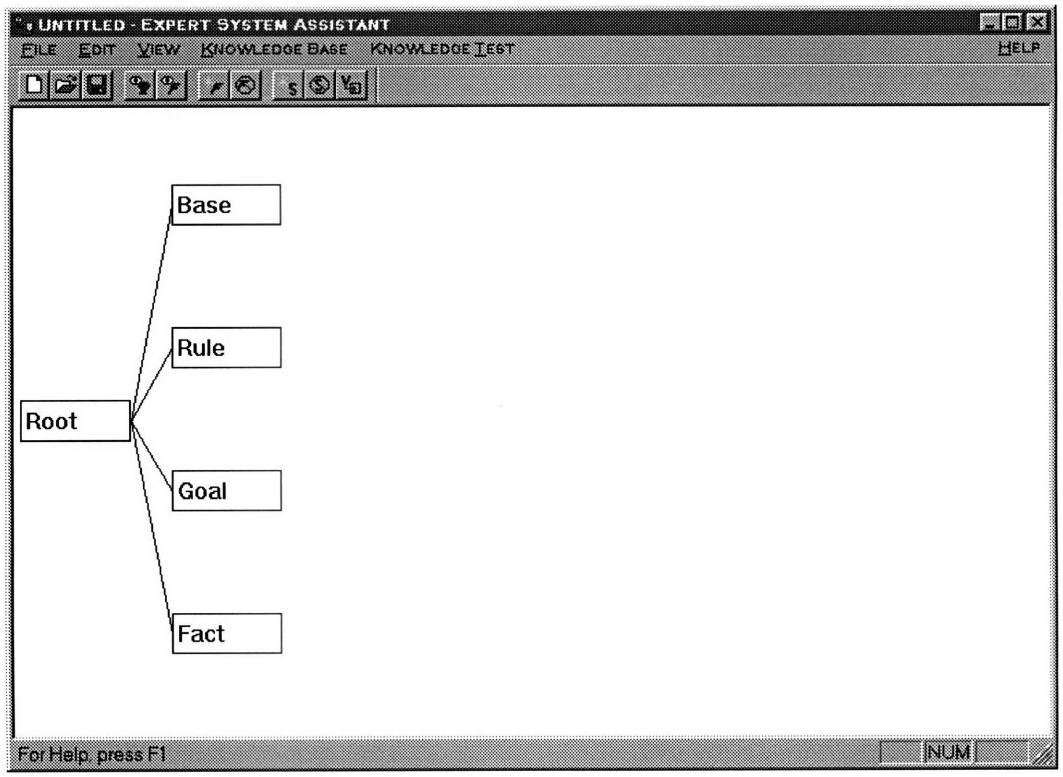

UNTITLED - EXPERT SYSTEM ASSISTANT
FILE EDIT VIEW KNOWLEDGE BASE KNOWLEDGE TEST HELP
Base
Rule
Root
Goal
Fact
For Help, press F1
NUM

Figure 4-1

The dockable toolbar which is initially docked just below the menu items contains a subset of the most commonly used functions in Expert System Assistant, allowing single mouse click access to common functions such as Add Frame, Remove Frame, Add Slot, Remove Slot, Set Slot Value, and Display Tree. Clicking on any of these buttons provides the same effect as choosing the associated function from the menu. To provide easy and intuitive use, each button displays a small bitmap which is meant to express the action which the function performs. Also, each button has a Tool Tip which pops up if the mouse pointer is left motionless over the button for about a second which displays the name of the function associated with the button. In addition to the Tool Tip, a description of the function is given in the help area immediately under the client area of the window.

In addition to the main application window, most of the functions in Expert System Assistant are used in conjunction with a dialog window which appears when the function is selected form the menu or a tool button. Each dialog provides a means for supplying any of the allowable parameters to the function which uses it through common controls such as edit boxes, check boxes and command buttons. For instance, Figure 4-2 shows the dialog window for the function Add Frame. It contains an edit field for both the Name of the frame to add and the name of the Parent frame to which the new frame will be a child. Also contained in the dialog are two optional controls contained within a graphical frame which allow the frame to be inserted either before a single frame or before all of the children of the Parent frame. After entering the appropriate values into the dialog, the user may choose either cancel to abort the operation, or OK which will apply the function with the chosen values.

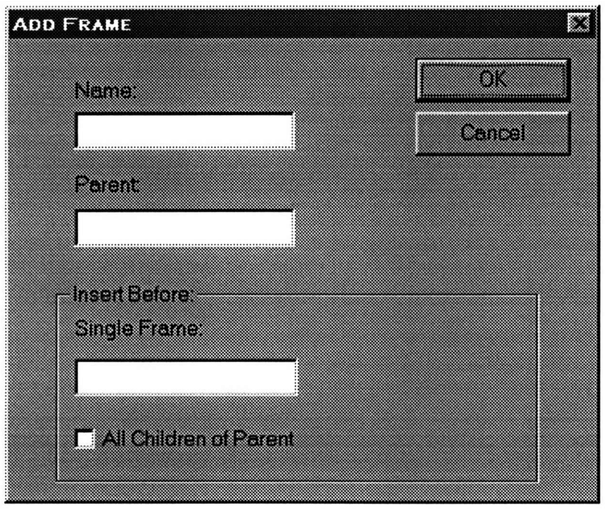


Figure 4-2

4.1 Basic Framework Constructs

The knowledge base of Expert System Assistant is implemented using two main classes: Frames and Slots. Frames generally contain Slots, and form the tree structure of the knowledge base. The following sections describe the Frame and Slot classes and also discuss some of the design assumptions and considerations in the implementation.

4.1.1 Frames

Frames are the fundamental unit in Expert System Assistant. Every object in the knowledge base is represented as a frame. The knowledge base is a tree of frames which are all derived from a frame named Root, which is one of the initial frames contained in an empty knowledge base upon creation. Frames are implemented as a C++ class with the following main attributes and methods:

Attributes:
- mName - name of the frame
- mParentName - name of the frame's parent
- mChildren, mNumberChildren - a linked list of the frame's children and the number contained within the list
- mSlotMap - a collection of the frame's slots

Methods:
- default and parent reference constructor, destructor

- CreateChild, AddChild, Remove Child - methods for maintaining children frames
- CreateSlot, AddSlot, RemoveSlot - methods for maintaining slots. Create add a new slot to the Slot List, while Add adds a referenced slot which has already been created.
- ResolveSlot - returns the values and references associated with a slot
- MoveChild - moves a child frame from the caller frame to a specified frame
- CreateInstance - instantiates an instance of the class

These attributes and methods are discussed in detail in the following sections. A complete listing of the attributes and methods is contained in Appendix A. Note that the main attributes and methods listed above are also listed in the header file frame.h. The class description for the Frame class also includes a number of attributes and methods which are used in managing the knowledge base serialization (saving and loading files) as well as for the purposes of displaying knowledge base trees and frame contents. Some of these methods are described later in this document under the heading **Supporting Operations**.

For the purposes of Expert System Assistant, the Frame class is a minimal implementation of frames discussed in earlier sections. These frames are capable of being arranged in a class-like hierarchy, where all frames but the Root frame posses a parent frame, and each frame may have child frames. Each child frame inherits all of the slots and values of slots possessed by the parent frame at the time the child is created. As listed above, frames have methods for managing their children which allow for the addition and removal of child frames, as well as the creation of frames during the addition of a child to a parent frame. Frames also possess a collection of slots, or properties, which map slot names which are unique within the frame to values associated with the slot.

4.1.2 Slots

The Slot class is an implementation of slots as described earlier in this document, but which adds several components to facilitate the slot requirements of the Expert System Assistant knowledge base. While the simplest implementation of a slot usually just associated a slot name with a single slot value, the Slot implementation described here adds several components:

Attributes:

- mName - the name of the slot
- mValue - the name of the frame of the value of the slot
- mReference - a pointer referencing the frame named by mValue, if it exists
- mMod - a string holding some extra user supplied information intended to be used as a modifier of the value

Methods:

- Constructor by name and destructor

Every slot has a string name, which is required to be unique within its owner frame (a requirement which is satisfied by using the AddSlot method in the Frame class). To accommodate Expert System Assistant's ability to deal with incomplete knowledge base specification, a slot has two attributes to represents its value. First it has a string name of the frame which is the slo'ts value. Since this is a string, the frame need not actually be defined anywhere in the knowledge base, allowing the user to name the values of any slot in any frame at essentially any time they wish. Associated with this name is a possibly NULL reference to the named frame value. If the named frame exists in the knowledge base, then the value of mReference is non-NULL. The value of mReference is checked at any stage of knowledge base editing that may change it, such as changing the name of the value, adding a frame, deleting a frame, or renaming a frame. Note that all of these events and the subsequent check of the existence of the value are carried out by either an instance of a frame or by the Document class which provides an interface to the knowledge base in the application. This is because the availability of frame existence information is more readily available at the frame management level. The mMod attribute of the Slot class is a quantity which can be used completely at the discretion of the user. Its value is never looked up and checked for a reference, but it could possibly add some semantic information which could be used in the translation of the Expert System Assistant knowledge base format into a conventional expert system development format, so it is included to provide extra descriptive capability to a slot.

4.1.3 Names

Within Expert System Assistant the concept of a unique name for every frame in the knowledge base, and for a unique name for each slot in the scope of a frame is built into the design. In the case of slot names, there was little choice but to require the unique names, since doing otherwise makes little sense from the point of view of slots as Attribute/Value pairs. The decision to require unique names for frames was based both on ease of use and clarity, as well as because it makes the implementation quite a bit easier. From the standpoint of the user, having the ability to use non-unique names has advantages and disadvantages. The main advantage is that, since the English language often uses the same word to name entirely different concepts, the user would not have to modify names or come up with synonyms when naming frames in the knowledge base. The disadvantage to allowing non-unique names is that all non-unique names in the knowledge base must then be specified by either a complete class path specification, or by some manner of unique identifier system, both of which add complexity and confusion, especially to the intended users of Expert System Assistant, who are not assumed to be proficient with computer programming methods and practices. As far as implementation goes, requiring unique names allows the use of simple string to pointer reference maps for tracking objects in the knowledge base, without the need for either large class hierarchy path names or the extra indirection step of converting a name into a unique identifier.

4.2 Supporting Operations

As described in earlier sections, if Expert System Assistant is to be successful in its task of helping to automate knowledge elicitation, it must provide the user with a set of small, readily available tools which aid the user if different areas of knowledge model building. These tools attempt to provide the user with as much flexibility as possible in most common development operations. The system provides tools in three general categories, corresponding to the menu items Knowledge Base, View and Knowledge Test, respectively. Knowledge Base operations are tools which allow the user to add, remove or modify the contents of the knowledge base is some manner. View operations are tools which allow the user to view certain parts of the knowledge base in a certain manner to inspect its current state. Knowledge Test operations provide a small, simple set of tools which the user can use to verify his or her desired contents of the knowledge base.

4.2.1 Frame and Slot Operations

Frame and Slot Operations give the user the ability to manage the frames and slots contained in the knowledge base. To satisfy the needs of the user in light of the fact that the knowledge base at any particular stage of development is incomplete and possibly incorrect in many different ways, Expert System Assistant provides several options for adding and removing both frames and slots, rearranging frames and subtrees within the knowledge base, and for modifying the contents of slots.

Adding Frames

Figure 4-3 shows the dialog with which the user is presented for adding a frame to the knowledge base. The user must provide both a Name for the new frame, as well as a Parent frame. To enable the construction of a knowledge base which is the result of sloppy modeling, frames can be added in one of two manners. First, frames can be added by *Top Down Insertion.* This method is what is usually provided by mainstream expert system development tools, and allows the user to add a child as a leaf to any frame in the knowledge base. This is useful if the construction of the knowledge base proceeds strictly in a Root to leaf fashion, such as is the case when the complete tree is decided upon beforehand, or in an orderly top down manner, which does not mesh well with the sloppy modeling paradigm.

The second method for adding frames provided by Expert System Assistant is to add frames by *Bottom Up Insertion.* This method allows the user to add a frame *between* two existing frames in the knowledge base. In the dialog below, the Insert Before bounding frame holds two controls to allow the user to choose how the in between insertion is carried out. If the user enters a name in the Single Frame edit box, then the new frame is inserted between the parent and this specified frame. If the user checks the All Children of Parent check box, then the new frame is inserted between the parent and all children of the parent. The ability to insert a frame at any point in the knowledge base tree increases the usability of the system a great deal. It allows for the relatively unstructured approach to knowledge base construction required by sloppy modeling in that it does not limit the user to a strictly top down model. It also allows the user to make mistakes when inputting the knowledge base tree structure, and allows the user to correct mistakes of omission, which are quite common in an undeveloped model, without having to delete entire subtrees of the knowledge base and rebuild them with the omitted frame in its correct position.

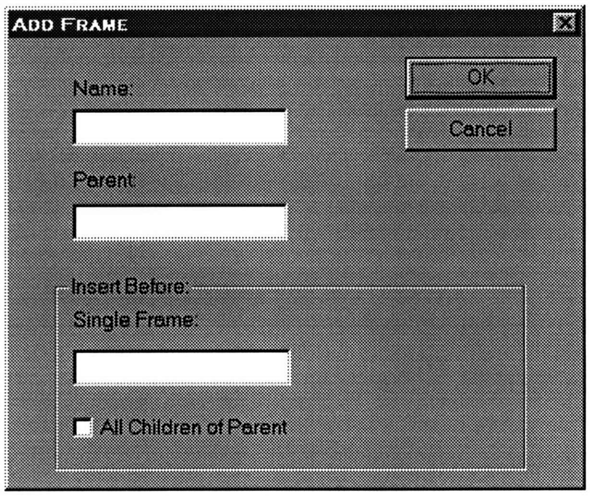


Figure 4-3: Add Frame Dialog

Removing Frames

The counterpart to adding frames, removing frames, is accessed through the dialog shown in Figure 4-4 below. Expert System Assistant allows two main methods for removing frames from the knowledge base tree. The first method is the standard approach, whereby a frame and all of the frame's children are deleted by using remove on the frame, so that an entire subtree is removed starting at the specified frame. The second method accommodates sloppy modeling, and allows the user to remove only the frame, and merges the frames children into the parent of the frame. This allows the user to at one point decide on using a frame, then later on determine that it is not necessary and remove it, without requiring that the user delete all of the work which went into developing the subtree below the frame, which the user may want to keep.

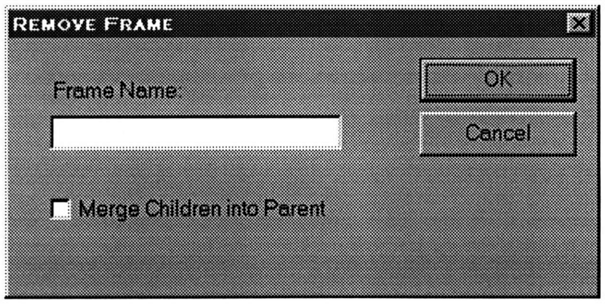


Figure 4-4: Remove Frame Dialog

Rearranging Frames

As a knowledge base model evolves over the course of the knowledge elicitation, major structural flaws may become evident to the user. At this point in a standard expert system development environment, the user is usually forced to delete large sections of the knowledge base and reimplement them. Expert System Assistant offers a couple of alternatives to this, and is therefore more accommodating of structural mistakes made during the modeling process.

The first tool to correct structural errors is the ability to move entire subtrees beginning with a specified node to an entirely different position in the knowledge base. Figure 4-5 depicts the dialog used for moving a subtree in Expert System Assistant. The user supplies the name of the frame at which the subtree begins, and the destination frame to which the subtree is to be added. Once the subtree is moved, any slots of the new parent which are not already present in the root of the subtree are added and propagated throughout the tree. However, slots which have the same name in both the destination parent frame and the subtree root name are not changed in the subtree, a fact which the user should be aware of. If the tool is used correctly, though, this should not be much of a problem, since a subtree will usually be moved between frames of the knowledge base which are similar; otherwise the move makes little semantic sense, since children of a frame are a type of subclass of the parent.

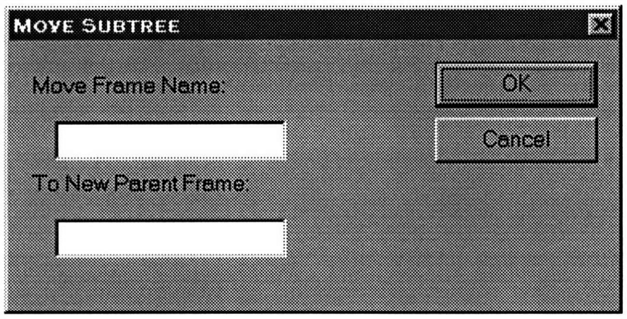


Figure 4-5: Dialog for Move Subtree

A second method for rearranging the structure of the knowledge base is by merging nodes of the hierarchical tree. The user supplies the names of two frames which are to be combined, one of which will remain in the knowledge base after the operation, and the other which will be merged into the first by adding all of its children to the first. The result of the operation is essentially the same as if the subtree of the second tree were first moved to the first frame, after which the second frame is deleted and the frames children are merged into the first. If the merged-to frame contains slots which are not possessed by the elements of the new subtree, they are added, similarly to Move Subtree.

Adding and Removing Slots

Adding a slot a frame is a relatively straightforward operation. When the user chooses to add a slot, he or she is presented with the dialog as shown in Figure 4-6. The required parameters are the Frame Name to which to add the slot and the name of the slot to add. Two other values can be specified. The first is the Slot Modifier which is an additional uninterpreted string value which can be associated with any slot. The second is the slot value, which is given as the name of a frame to which the value of the slot should reference. As mentioned previously, the slot value does not have to reference a frame which is already contained within the knowledge base, and may be any name. When the slot is added, the name is looked up, and if it is found in the knowledge base, a real reference to the frame value is inserted into the Slot class structure. At no point does the user have to supply the actual reference to the value of a slot, or even to specifically initiate the process of determining whether or not slot value names have existing referenced frames.

Removing slots is also very straightforward. Figure 4-7 shows the dialog used for removing slots. The dialog simply asks for the name of the frame which contains the slot and the name of the slot. If both exist, then the slot is removed.

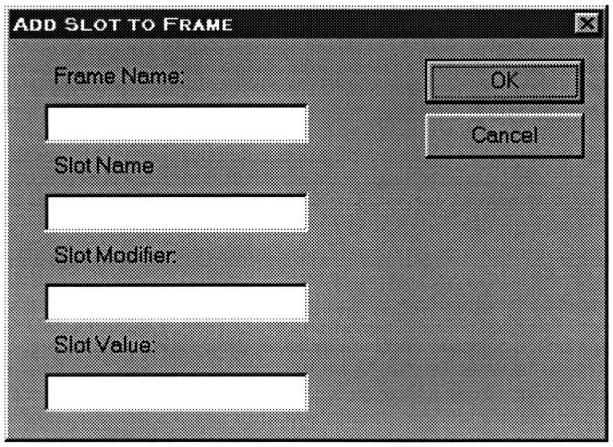


Figure 4-6: Dialog for Add Slot to Frame

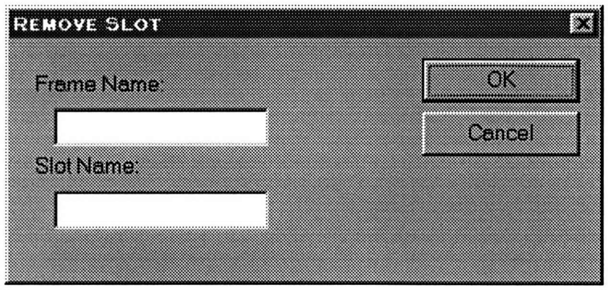


Figure 4-7: Dialog for Remove Slot

4.2.2 Knowledge Testing Operations

Expert System Assistant provides several tools to the user for testing the contents of the knowledge base. These tools fall into the categories describe earlier: list tests, hierarchy tests and comparison tests. The tests are useful in confirming that there is a match between the expert's internal mental model of the knowledge domain and the model contained in the knowledge base of the system. They are intended as simple integrity tests, and as such do not use much in the way of knowledge inference, other than that which can be gained by straight forward examination of the knowledge base hierarchy, since Expert System Assistant does not contain an inference engine. If later versions are implemented which contain an inference engine, metaknowledge rules could be added which would greatly enhance the testing abilities of the application, by allowing actual interpretation and subsequent judgment of the *contents* of frames in the knowledge base instead of just hierarchical tests. However, the testing tools provided by Expert System Assistant in its present form do provide a good means of making sure that what is in the knowledge base is what is intended.

List tests as described earlier are provided mainly through graphical views of the knowledge base either as a whole or in specified sections. These views are described in detail in the next section.

The system provides a couple of hierarchy tests, which can help the user determine the relationship of a specified frame in the knowledge base to other frames in the knowledge base. The first of these is a test labeled "Is a ... ?" in the Knowledge Test menu. This test presents the user with a dialog which allows him or her to enter a name of a frame to be tested as well as a second frame against which to judge the first frame. If the second frame is an ancestor frame of the first, then the tool returns a True condition, otherwise it returns a False. This tool is useful for large knowledge bases where the entire knowledge base tree does not fit on a single display screen. It also allows the user to confirm at their discretion that any frame in the knowledge base is descended from what their expectations are.

A second hierarchy test is listed in the Knowledge Test menu as "A ___ is a ..." This test takes as input a single frame name, and returns a list of all of the ancestors of the frame. This test is similar to the first hierarchy test, but returns a complete listing of all parent frames of the specified frame, allowing the user to determine the precise derivation for the frame all the way back to the Root frame.

Expert System Assistant also possesses a comparison test, labeled "Difference" on the Knowledge Test menu. This test lets the user input the names of two frames, as well as choosing the method of comparison. The first method is Common Slots, which will cause the tool to return a list of all of the slots common to both of the specified frames. The other method available is Disparate Slots, which will cause the function to return two lists, the first of which is a list of the slots contained in the first frame but not in the second, and the second, which contains all of the slots contained within the second frame but not the first. Only slot names are compared, not slot values, so the test is less of knowledge content test than an internal property structure test. The test can provide the user with valuable information for both evaluation of the current state of the two frames as well as giving the user directions for improving the model based on what differences or commonalties *should* exist between two frames of the knowledge base.

4.2.3 Knowledge Viewing Operations

Expert System Assistant offers several different methods for viewing the contents of the knowledge base. This allows the user to inspect many different aspects of an evolving knowledge base, from viewing the entire knowledge base tree, to inspecting the individual property slots of a frame in the knowledge base. Overall, there are two basic types of view, a tree view, which displays a portion of the knowledge base in the form of a graphical hierarchical tree, and a frame view, which displays each of the property slots of a frame along with all of the relevant information contained within the slot.

Tree View

The frame view in Expert System Assistant presents the user with a graphical tree which represents the hierarchy of frames in a subtree of the knowledge base. There are several default modes available from the View menu, as well as an option for a custom view. Figure 4-8 shows the default Fact subtree view mode for an small example knowledge base. The default modes correspond to each of the four default frames which are available in each new knowledge base, which are discussed in the next section. These modes are Fact, Rule, Goal and Base knowledge views. Each of these views displays the subtree of the knowledge base rooted at each of the four default frames. In addition to the default modes, the user may also choose the "Display Tree at Root" option from the View menu. This allows the user to specify the root frame of the subtree for display, which is useful when any of the default subtrees grows to a size which is inconvenient to view all at one time on the screen.

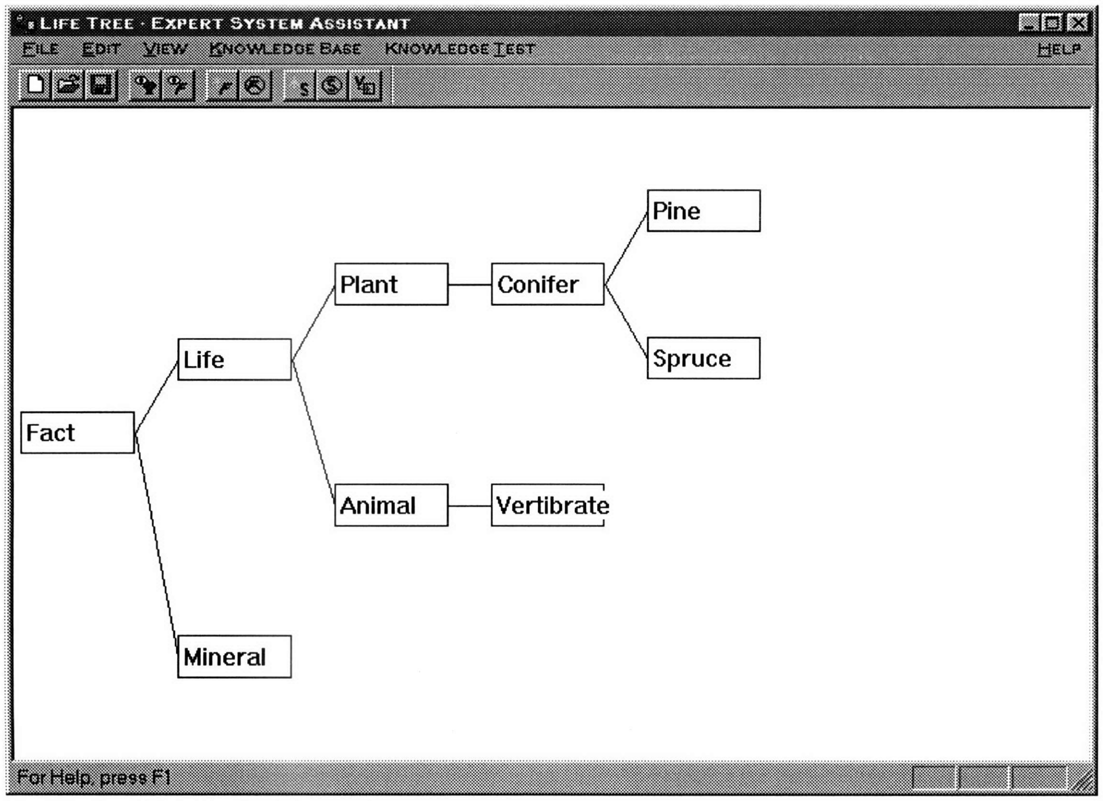
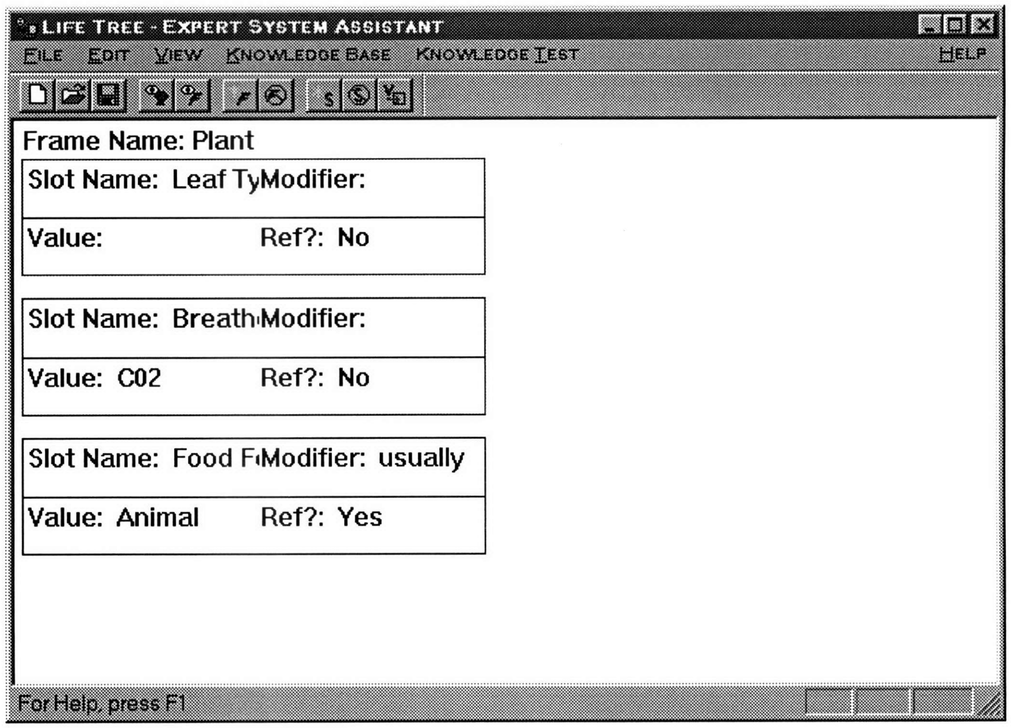

Life Tree - Expert System Assistant
File Edit View Knowledge Base Knowledge Test Help
Pine
Plant
Conifer
Life
Spruce
Fact
Animal
Vertibrate
Mineral
For Help, press F1

Figure 4-8: Tree View of Fact Subtree

Frame View

This viewing mode allows the user to inspect the property slot attributes of a specified frame. Figure 4-9 shows an example of the frame view for the Plant frame in the above Tree View example. The view displays the name of the displayed frame, along with a series of boxes which represent each of the slots of the frame. Each of these boxes has four displayed values: the Slot Name, the slot Modifier, the string name Value of the Slot, and a Boolean Ref? value which signifies whether or not the frame specified by Value exists in the knowledge base. The frame view is useful in determining the values or existence of any of the properties of a frame.


Life Tree - Expert System Assistant
File Edit View Knowledge Base Knowledge Test Help
For Help, press F1

Frame Name: Plant

Slot Name: Leaf TyModifier:
Value: Ref?: No

Slot Name: BreathiModifier:
Value: C02 Ref?: No

Slot Name: Food FiModifier: usually
Value: Animal Ref?: Yes

Figure 4-9: Example Frame View

Frame Existence Cues

Since Expert System Assistant is capable of dealing with incomplete knowledge bases which may contain references to non-existent frames, a problem arises in identifying frames which have been referenced but not implemented. To aid in this task, the application provides a visual cue to identify objects in the knowledge base which reference incomplete information in both tree and frame views. In the tree view, any frame which references incomplete knowledge has its bounding box outlined in blue, while frames which only reference complete knowledge are outlined in black. By noting which frames in the tree are colored blue, the user can determine which areas of the knowledge base need further information defined. Likewise, in the Frame view, any slot which references incomplete knowledge is also outlined in blue, drawing attention to any reference to unspecified frames at a glance without having to examine the Ref? value in each slot. This use of color provides a convenient, automatic and intuitive means for guiding the modeling process of an evolving knowledge base, and is particularly useful once the knowledge base grows large, as it is quite easy to forget what exactly has or hasn't been implemented in large systems.

4.3 Knowledge Content

Part of the specification for Expert System Assistant outlined in previous sections called for the inclusion of some default knowledge upon which a novice user could base a usable model of their own. The default guiding knowledge provided by the system falls into two categories, a set of default frame types with default slots which the user can user to subclass their own frames, and a subtree of Base Knowledge which the user can use as a basis for the slot descriptions. For a novice builder of knowledge bases, the availability of base knowledge is essential to guide the efforts in constructing an accurate and usable model of his or her expert knowledge domain.

4.3.1 Default Frame Types

Whenever a new knowledge base is created, four default frames are automatically created which are subclassed from the Root class. These frames are the Fact, Rule, Goal and Base frames. Each of these frames has associated with it a default view which will display the subtree rooted at each, as described previously. These four frame types make up what is basically a minimal set of frame types which are needed to implement most expert systems. Each contains default slots which are appropriate for each default frame type.

The Fact frame is the most general of all the default frames. It contains no default slots, and is meant to be mostly a convenient root at which to place all of the declarative knowledge in the knowledge base. It does not attempt to impose any structure on the rest of the frames which are descended from it.

The Rule frame is a simple implementation of knowledge base rules. It contains six slots which make up a rule structure which accommodates two rule conditions and one rule predicate, a common type of rule in any expert system. Each rule condition is composed of two slots. The first slot has as a value the name of a frame in which a condition is to be checked. The slot modifier of this slot should be used for the name of the slot which is specified by the condition. Optionally, a "!" can be placed in front of the name in the modifier to specify the negation of the condition. The second slot making up the rule condition specifies the value which the first slots reference to a frame/slot should have if the condition is True. The predicate portion of the rule is also composed of two slots. The first slot is used in a similar manner to the first slot of a condition, in that the value specifies the frame and the modifier specifies the slot in the frame which should take on a new value. This new value is specified by the second slot which makes up the predicate. The value of the second slot is the value which the slot in the specified frame should be set to if the conditions of the rule are met. A rule must have at least one condition complete and the predicate complete in order to be meaningful. Leaving one condition empty corresponds to a rule with only one condition.

The Goal is similar to a Rule frame, but has only four slots which compose two conditions at which the goal has been achieved. The format of the slots in the conditions is identical to that for the default Rule frame. Note that other formats for Goals and Rules will undoubtedly be useful to define, and should be patterned after the preceding descriptions of Rule and Goal conditions and predicates, so that a suitable translating procedure can be implemented to translate the Expert System Assistant format to a general expert system development environment format.

The default Base frame serves as a root for a subtree of Base Knowledge, the contents of which is discussed in the next section. It, like the Fact frame, is very general in the sense that it does not have any default slots, and is meant more for the convenience of organizing base knowledge and for providing a suitable root at which to display the base knowledge subtree. However, base knowledge in Expert System Assistant is considered a distinct section of knowledge which is somewhat independent of the rest of the knowledge contained in the knowledge base. The application provides facilities for saving the contents of the base knowledge separately from the rest of the knowledge base, and also for importing a the base knowledge into an existing knowledge base. Note that when importing base knowledge, it is possible to merge the contents of the existing base knowledge with the contents of base knowledge file to be loaded. However, since these two sets of base knowledge may have a non-empty intersection which may be implemented in different manners, this merging can cause some inconsistencies. Therefore, the default operation of loading base knowledge is to simply replace the existing base knowledge with that contained within the base knowledge file.

4.3.2 Base Knowledge

As mentioned in the previous section, base knowledge in Expert System Assistant exists as a discrete portion of the entire knowledge base and can be saved on loaded separately from the entire knowledge base. Because of this, the base knowledge itself is not so much a concrete part of the system as it is an knowledge domain specific feature which has special support. Base knowledge is intended to be developed by an experienced knowledge engineer and provided to an knowledge domain expert who will actually use the system to elicit his or her knowledge into the knowledge base. The actual format and requirements of the base knowledge was discussed in earlier sections and will not be repeated here, though some implementation specific details will.

The base knowledge in Expert System Assistant is not treated differently from other knowledge in the system when it comes to modification of the knowledge such as frame addition, removal and rearrangement, so it should not be viewed as static or unchangeable in any way. In fact, the opposite is true, and the user should be encouraged to add or modify the base knowledge in any way he or she sees fit, as the base knowledge is really only provided as a suggested starting point meant to make the knowledge domain expert's task easier.

When a new base knowledge file is first loaded, the user should take the time to become familiar with the contents of the base knowledge in order to maximally exploit its usefulness and not repeat or neglect any of the possibly useful definitions of the base knowledge. Since most elements of the base knowledge should be atomic, that is, they do not have slots which reference other frames in the base knowledge, the task of familiarization can be accomplished simply by studying the default Base tree view. The exception to this is if any higher level concepts have been placed into the base knowledge for convenience. The knowledge engineer who provides the base knowledge should advise the user of any base knowledge frames which are non-atomic, since in the tree view, these will not be obviously different from the atomic frames and could cause the user to overlook important features.

It is very likely that the full utilization of the base knowledge will be one of the major difficulties in using Expert System Assistant, since it is largely dependent on the knowledge domain and therefore non-standard across knowledge domains. The user should be encouraged to try to place as much of their knowledge descriptions in the terms of the base knowledge as possible. He or she should be encouraged to view the combined system of Expert System Assistant and the custom base knowledge as a kind of limited student who possesses both mechanisms for learning (Expert System Assistant) as well as a limited vocabulary and knowledge of the topic being taught (the base knowledge). This may help to optimize the effectiveness of the base knowledge, as it may be easier for the knowledge domain expert to view the system as a type of student than as simply a number of computer commands and computer data.

5. Tree Identification Example

As an example of the usage of Expert System Assistant, a very simple knowledge base for tree identification is described in this section. The example consists of a set of Base Knowledge which seems to be applicable to the knowledge domain, as well as a set of facts which are intended to represent some declarative knowledge about several types of trees that I personally know of. As I am hardly an expert on tree identification, the example presented here may or may not be an accurate representation of *real* tree identification, but it should serve to show that the system works well for an "expert" even when the expert's model isn't explicitly known to the expert beforehand.

5.1 Base Knowledge

The Base Knowledge depicted in Figure 5-1 is an example set of atomic information which is useful in describing some simple varieties of trees. It includes several hard to quantify or easily define notions such as color, texture and smell, as well as some higher level ideas which, while not necessary atomic ideas, could possibly be difficult and inconvenient to define, such as Seed and Fruit. The entire set seems to embody a subset of the knowledge a non-expert would possess which would allow an expert to educate the student about the knowledge domain of tree identification.

This Base Knowledge was defined solely on my personal simple understanding of what named concepts I would need to explain my knowledge of trees to a student. I decided to include representation of several sensory perceptions which a person would need to understand trees. The most important sense seemed to be sight and color perception, which is why that category below is the most developed. I chose a set of colors which are common to trees: green for leaves and such, yellow and brown for wood colors, and blue for a sky or flower color. Since green is a very important color for trees, it is further subdivided into three categories of green, dark, medium and light green. Smell and taste also seem important for some trees, especially pungent trees and flowers, which can be fragrant, citrus or piney, if they have a smell, or trees which have edible or even inedible fruit or leaves, which are usually sweet or bitter. The sense of touch is accounted for by both surface and edge textures, which are rough and smooth for surface like bark and leaves and jagged or smooth for edges, usually of leaves.

I decided to include the higher level knowledge concepts of Seed, Leaf and Fruit because they seem like quantities common to many different types of trees, and often are useful in determining what kind of tree one is dealing with.

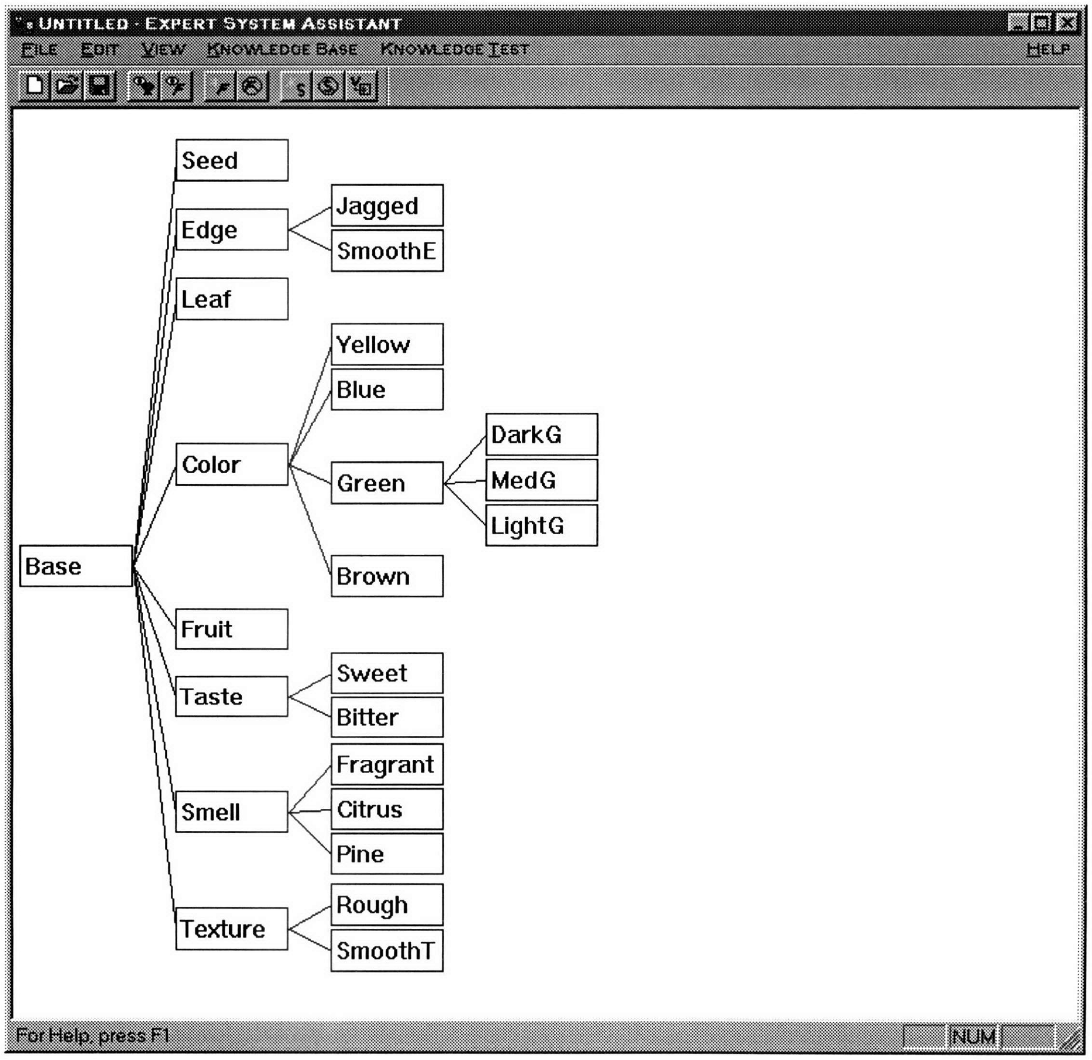


Figure 5-1: Base Knowledge For Tree Identification Example

As a general procedure for determining an applicable set of base knowledge, a type of two cycle derivation stage seems to work well when repeated to refine the knowledge. To begin with, simply write down anything that appears to be integral to understanding the simplest constructs of the knowledge domain. Then, as the first stage of the iterative refinement cycle, the knowledge engineer should choose elements of his or her personal knowledge and test them to see if it would be possible to describe each in terms of the current state of the base knowledge. The second part of this cycle of examining the usefulness of the base knowledge in describing the simple knowledge domain is to add any new base knowledge which is revealed by the inspection in the first part of the cycle. Obviously, the choices for the base knowledge are highly subjective, and would well benefit from previous experiences in choosing base knowledge. However, this is exactly the area in which the experience of the knowledge engineer with knowledge representation can best be leveraged to provide an adequate and useful set of base knowledge.

5.2 Factual Tree Identification Knowledge

Using the base knowledge described in section 5.1 above, a simple knowledge base which could be used to identify a number of trees was quickly developed and is shown in figure 5-2 below. All of the frames shown in the Fact subtree of the Tree Identification knowledge base below contain descriptive slots which have values which are either from the base knowledge, or from frames which ultimately reference base knowledge frames. Note that while the knowledge base subtree below appears to have a definite and purposeful organization and hierarchy, no effort was made before data entry began to formulate such a structure, in order to most closely approximate the use of the system by a knowledge domain expert who may be unfamiliar with determining such detailed models.

The definition of the subtree shown here interestingly enough *did* follow a recognizable and logical ordering of descriptions, but this ordering had little in common with the final tree hierarchy ordering of the frames. The frames here were described in an order which would be suitable for explaining the difference among Spruce and Maple trees to a person whose only knowledge of trees is that which is contained in the base knowledge, mainly such quantities as color, texture, taste and smell, as well as the higher level facts contained in the knowledge base. The basic order of definitions were mostly what one would expect given the "learner's" assumed knowledge. First, the concept of Tree was introduced, along with the definition of the individual parts of a Tree, which consisted of about four different slots: Trunk, Branches, Roots and Leaves, all of which were determined "on the fly" in a type of speculative manner wherein any likely useful descriptive value is added as a slot. Since none of these values were in the knowledge base at this point, their unreferenced values were noted by blue borders in both the tree view and frame view of Tree.

Working from the noted unreferenced values, the individual descriptive elements of Tree were defined, though in a rather circuitous manner. This indirect manner resulted from the fact that some of the descriptive elements of Tree are related; specifically, Trunk, Branches and Roots all eventually become descendants of a WoodPart frame. Initially, all three of these were simply defined as descendants of Fact, but upon noticing that they were all related by the fact that they are made of Wood (which also was defined at this point) they were grouped under the WoodPart frame to represent their relation to each other. Afterwards, Roots and Trunk were also put under their own frame since they compose the major part of the tree, while branches are smaller parts of a tree which are attached to the major parts. The overall pattern of definitions that resulted was basically a depth first investigation of the knowledge domain, very similar to the process which people go through when explaining unknown facts to each other: any unknown fact is usually explained fully before moving on to the definition of other unknown facts, which makes sense given that people do not have the large temporary storage needed for a breadth first version of the same definitions, where many facts are in a state of partial definition at one time.

After the underlying foundations of what it meant to be a Tree were defined, I moved on to defining two trees, Spruce and Maple. A similar method of definition resulted, where both Spruce and Maple were initially added directly to Tree. However, after further consideration and evolution of my internal mental model, I decided to insert the types of trees between the general Tree type and the specific, since it seemed important to differentiate them by classes of tree, resulting in the PineTree and Deciduous frames. After defining these two general types of trees, differences between them were noted, which resulted in adding a slot for Smell in PineTree, and setting the specific value of the Leaves slot in each to Needle and Broad, respectively. Then, to add additional differentiation capabilities, their individual Seed values were set to Pinecone and Acorn, neither of which were in the knowledge base, resulting in the usual display of unreferenced values and then their implementation.

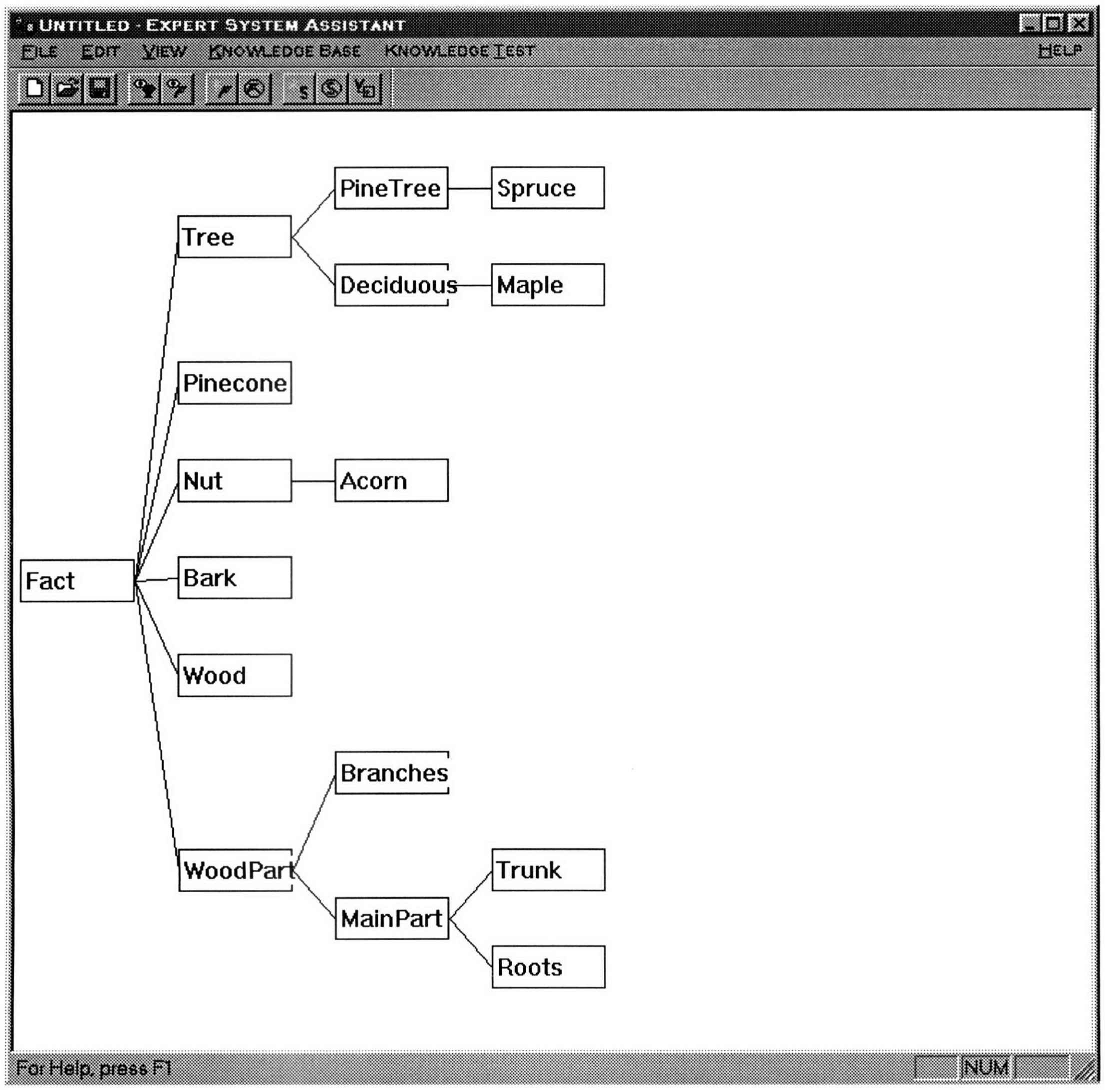


Figure 5-2: Tree Identification Example Fact Subtree

Overall, Expert System Assistant provided facilities which enabled a very natural and unrestricted modeling process which was forgiving enough to deal with incomplete definitions and the subsequent corrections made, such as moving frames or grouping frames into new parent frame classes. Without the freedom of frame manipulation and the ability to use unreferenced values, the above knowledge base subtree would have been much more difficult to build entirely in the application. Rather, the tree would have had to have been well thought out beforehand and implemented more according to the knowledge base tree hierarchy, instead of the more natural ordering used here.

The base knowledge described before was also quite helpful, especially if it were simply given to me, instead of having to make it up myself. Most of the elements contained within it were very useful and applicable to the resulting example. However, the most useful of the base knowledge frames turned out to be the frames which were as close to atomic as possible. These frames were the easiest to incorporate into the descriptions of higher level frames, and took the place of frames which otherwise might have been hard to define and develop if I were unclear about what sort of basic knowledge I should use in descriptions.

However, the frames which were of higher level concepts than senses and such were a little more problematic. It became clear when defining new frames incorporating the Seed base knowledge frame that it may have been inappropriate, since it was necessary to define both a Pinecone, which could reference it, but was also necessary to subclass from it, in the form of specific seeds such as Acorn. Notice that since specific instances of Seed could very well be common in a Tree Identification expert system, many such inappropriate derivations could be necessary. To correct for this, Acorn was derived from Fact, and given a slot which referenced Seed as a value, with a slot modifier of type, which meant to signify that Acorn had type Seed. Obviously, this is not optimal, since the implementation loses some of the semantic content embodied by having Acorn descended from Seed. This problem brings to light an important consideration in defining base knowledge: if a high level frame is included in the base knowledge, there should be a minimum number of cases where it should need to be subclassed in the knowledge base. Ideally, if these subclasses are necessary, they should be included in the base knowledge, and not delegated to the user for implementation.

6. Conclusion

Expert System Assistant provides many features which enable it to be used by an expert in a knowledge domain instead of a knowledge engineer which aid in acquiring expert knowledge. It attempts to provide an alternative to the conventional knowledge acquisition process by which a knowledge engineer interviews at length concerning the domain of the expert knowledge. This process is usually quite lengthy and complicated, and is usually the most difficult aspect of constructing any expert system. The approach which Expert System Assistant takes to ease the knowledge acquisition process is to essentially replace the knowledge engineer administered interviewing process with a combination of a convenient knowledge modeling environment and simple built in domain knowledge provided by the knowledge engineer upon which the expert himself can construct a knowledge base.

Expert System Assistant as described in this thesis seems to be a step in the right direction towards the goal of computer enable knowledge acquisition with minimal intervention from a knowledge engineer. The tools it provides for knowledge modeling, while not entirely different from those offered by conventional expert system development tools, do provide a greater measure of ease of use and the capability of dealing with sloppy modeling, which is much less constrained and strictly organized than that which is expected and demanded by most tools. The most useful of the capabilities of Expert Assistant, at least from developing the simple example in the previous section are the many different methods for adding elements to a knowledge base, the capability of adding an element of the knowledge base at any location one chooses, and the ability of the system to effectively deal with knowledge which is incompletely specified. The combination of all of these capabilities means that a user of the system is only very minimally constrained in his or her actions, and may go about the modeling process in a very free manner, without having to have a complete model developed before building the knowledge base, which is a very common requirement from many expert system development environments. This amounts to a system which is much more appropriate for use by a novice user, one who may have very little experience in developing knowledge bases of the type supported by Expert System Assistant, such as an expert in the knowledge domain under consideration.

The capability of defining a set of base knowledge serves a couple of essential purposes. First, it gives the knowledge engineer a way in which to shape the underlying structure of a knowledge base by suggesting useful elements with which to describe higher level constructs in the knowledge base. Without this suggestive control, the expert using the application may not be able to construct a knowledge base which is structured in an appropriate manner, since he or she is not assumed to have any experience with constructing proper knowledge base structure. Additionally, by providing a complete set of base knowledge, the knowledge engineer enables the expert user to concentrate only on providing expert knowledge which is *not* already known by the knowledge engineer, thereby utilizing the expert's precious time in the more efficient manner. It makes little sense to waste the expert's time in having him or her develop knowledge that is readily available from the knowledge engineer, who in any case is much more qualified to construct this base knowledge in a consistent and appropriate manner.

An added benefit of base knowledge is that in many cases it is applicable to several different knowledge domains. This means that if a set of expert systems which were all closely related were to be developed, the work put into developing the base knowledge could be reused in each, which is certainly much more efficient than having to build all of the related experts systems from the ground up each time. In fact, entire complete sets of base knowledge could be developed and stored in a library. When an expert system needs to be developed, the appropriate set of base knowledge could simply be chosen and used from the library. This could facilitate commercial versions of applications like Expert System Assistant which use base knowledge. Such applications would be very appropriate for small businesses and projects which need to develop an expert system but have limited resources with which to employ a knowledge engineer to develop an expert system for them.

Directions for Further Study

In its current form, Expert System Assistant is very convenient for use by a novice in the expert system development field. However, it would benefit from some additional features, some of which would require major additions to the architecture and some which would not. The easiest additions to the system would be a larger set of knowledge tests and knowledge verification procedures which attempt to aid the user in evaluating a knowledge base structure.

One such type of test which would be useful, especially considering the sloppy modeling paradigm, would be a test which examines a subtree of the knowledge base by looking at the slots contained within the frames of the subtree. Based on these contents, the test could attempt to determine if it would be useful to either condense separate frames into one, or split some frames into two or more frames. As an example, if two children of a parent frame each possessed the same different slots from the parent, it might be appropriate to condense them into one frame, as they embody the same information. The possibilities of such knowledge tests are really only limited by creativity and the complexity of their implementation. However, the total number of such tools should ideally be kept small, as was recommended in several other works done in this problem domain as mentioned in previous sections, which recommend that knowledge acquisition tools provide a small set of readily accessible tools. The use and effects of the tools should also be kept straight forward and minimal so that even the novice user may use them effectively without extensive training.

Another useful capability to add might be the ability to assign more than one name with a single frame, since many concepts in almost any knowledge domain have synonyms. This is especially important when the system is being developed by more than one expert, as different people often use different terminology to refer to the same concepts. Notice that this argument also applies even if only one person uses Expert System Assistant, since the base knowledge is usually developed by a separate knowledge engineer who may user different terminology. In fact, the entire naming scheme of Expert System Assistant could benefit from using a different approach, because in its current implementation, each frame in the system has a name which is unique in the entire knowledge base, while in the real world of knowledge, disparate concepts can often have the same name. Whether or not a convenient implementation of such a system could be implemented easily is another matter, as it would not really be desirable to require the user to specify the complete tree path of a frame when referencing a frame. Such a system might attempt to keep track of all frame names which are repeated in the knowledge base, give each a unique internal identifier, and then, when a frame of a synonym name is referenced, require the user to specify the exact reference.

An addition to Expert System Assistant which would most extend its capabilities, and one which really does belong in any expert system development tool, would be the inclusion of an inference engine. At a most minimal level, this addition would allow the user to actually test rules and goals in the knowledge base, where now there is only a means of examining the rules and goals in a static state, rather than by running a chainer with them and examining the results. More importantly, however, if an inference engine were integral to the application, it could actually be used to implement a sort of expert system which is expert in building knowledge bases. The knowledge engineer could place a set of base metaknowledge rules in the knowledge base which could be used to evaluate the semantic contents of the knowledge base. In effect, a user of the system would then have the knowledge and experience of the knowledge engineer available to them in the development environment to critique choices of frames, warn of possible problems due to overall knowledge structure, and even suggest necessary and useful additions to the knowledge base. However, though the idea is attractive, the addition of the inference engine would entail some major architectural changes in Expert System Assistant, while the implementation of the metaknowledge and its use would take a great deal more time and consideration than the current implementation, especially if the system were to remain simple and easy enough to use for a novice.

7. Bibliography

Aptitude, Learning and Instruction Volume 2: Cognitive Process Analyses of Learning and Problem Solving. 1980, Lawrence Erlbaum Associates, Inc.

Millward, Richard B. "Models of Concept Formation."

Norman, Donald A. "Discussion: Teaching, Learning, and the Representation of Knowledge."

McGraw, Karen L. and Harbison-Briggs, Karan, 1989. Knowledge Acquisition: Principles and Guidelines. Prentice Hall, Englewood Cliffs, NJ.

McGraw, Karen L. and Westphal, Christopher R., eds., 1990. Readings in Knowledge Acquisition: Current Practices and Trends, Ellis Horwood Limited, West Sussex, England.

LaFrance, Marianne. "The special structure of expertise."

Mishkoff, H., 1985. Understanding Artificial Intelligence, Dallas, TX: Texas Instruments, Inc.

Morik, K., Wrobel, S., Kietz, J.U. and Emde, W., 1993. Knowledge Acquisition and Machine Learning: Theory, Methods, and Applications, Academic Press, New York, NY.

Pylyshyn, Zenon W., 1984. Computation and Cognition: Toward a Foundation for Cognitive Science, The MIT Press, Cambridge, MA.

8. Appendix A: Source Code

The source code contained here is a subset of the entire source code for Expert System Assistant. Since the application was developed using MFC classes and an MFC application framework, much of the actual source code, which is on the order of 100 pages, comprises user interface and display code, which, while important to the overall application, is somewhat independent of the problem domain of Expert System Assistant. Therefore, only four source files are included here: Frame.h and Frame.cpp, which are the header file and implementation file, respectively, for the Slot and Frame classes used by Expert System Assistant. These are essentially container classes with some methods for managing the tree structures which are implemented using class attributes. Also inluded are Thesisdoc.h and Thesisdoc.cpp, which are a class description and implementation of the applications Document class, which manages the knowledge base, provides application methods on the knowledge base, and also provides the handlers for user interface events.

Frame.h

```
// frame.h
// An C++ implementation of Frames
// written by Brodi Beartusk
// October 3, 1995
#ifndef FRAME_H
#define FRAME_H

#include <afxtempl.h>

class Slot : public CObject {

// constructors and destructors
public:
        Slot();
        DECLARE_SERIAL( Slot )
        ~Slot();
        void Serialize( CArchive& ar );

        Slot *Copy();


// Implementation
protected:



public:
        CString mName;
        CString mMod;
        void *mReference;
        CString mValue;
```

```
            CString mDefault;
            Slot *mNext;
            Slot *mPrev;
            SlotID mID;

private:

};


class Frame : public CObject {

// constructors and destructors
public:
            Frame();
            DECLARE_SERIAL( Frame )
            Frame(Frame *parent);
            ~Frame();
            void Serialize( CArchive& ar );

// Implementation
protected:


public:
            CString mName;
            CString mParentName;
            Slot *ResolveSlot(CString name);
            Frame *mpParent;
            void EnumChildren(void *fxn(LPARAM d, Frame *f), LPARAM d);
            Bases mBase;
            Frame *CreateChild(CString name);
            Frame *AddChild(Frame *pNFrame);
            int DeleteChild(Frame *f);
            Frame *CreateInstance(CString name);
            int SetDefaultValue(Frame *f, Slot *s);
            int CreateSlot(CString name, CString framename, Frame *reference, CString mod, BOOL
propagate = TRUE);
            void RemoveSlot(CString name);
            void MoveChild(Frame *f, Frame *dest);

            Frame *FindFrame(CString name);

            static CTypedPtrMap<CMapStringToOb, CString, Frame*> *msTreeMap;
            CTypedPtrMap<CMapStringToOb, CString, Slot*> *mSlotMap;

            void AddToFrameMap(CTypedPtrMap<CMapStringToOb, CString, Frame*> *pMap);
            void RemoveFromFrameMap(CTypedPtrMap<CMapStringToOb, CString, Frame*> *pMap);
            void ToCMap(CMap<CString, LPCTSTR, CString, CString&> *pMap);
            Frame *Copy();

            int mNumberLeaves;
            int CountLeaves();
```

```
        Frame *mNext;
        Frame *mPrevious;
        int mNumberChildren;
        Frame *mChildren;
        Slot *mSlots;
        Slot *mSlotsTail;
        int mNumberSlots;

        // marker to identify if any contained slot has an unreferenced
        // value for use in display coloring
        BOOL mSlotUnreferenced;
        void CalcSlotUnreferenced();

        Frame *mChildrenTail;

        // marker to identify overall frame type:
        // Base, Fact, Rule or Goal
        int mType;

        Frame *DetermineType();

private:
        int mNumberInstances;
        Frame *mInstances;
        Frame *mInstancesTail;
};

#endif
```

Frame.cpp: Implementation file for the Slot and Frame classes used in Expert System Assistant

```
// frame.cpp
// implementation file for frames and associated classes
// written by Brodi Beartusk
// October 3, 1995
#include "stdafx.h"
#include <afx.h>
#include "frame.h"

IMPLEMENT_SERIAL( Slot, CObject, 1 )
IMPLEMENT_SERIAL( Frame, CObject, 1 )

Slot::Slot() {



	mDefault = "no";
	mNext = NULL;
	mPrev = NULL;
}

Slot::~Slot() {
}

void Slot::Serialize(CArchive& ar) {

	CObject::Serialize(ar);

	if (ar.IsStoring()) {
		ar << mName;
		ar << mMod;
		ar << mValue;
		ar << mDefault;
	}
	else {
		ar >> mName;
		ar >> mMod;
		ar >> mValue;
		ar >> mDefault;
	}
}

Slot *Slot::Copy() {
	Slot *temps = new Slot();
	temps->mName = mName;
	temps->mMod = mMod;
	temps->mValue = mValue;
	temps->mDefault = mDefault;

	return(temps);
}

// static data members of frame class
CTypedPtrMap<CMapStringToOb, CString, Frame*> *Frame::msTreeMap = NULL;
```

```
Frame::Frame(Frame *parent) {
	int i, count;
	POSITION posi;
	CString name;
	Slot *temps;

	mSlotUnreferenced = FALSE;

	mpParent = parent;
	mNumberSlots = 0;
	mChildren = NULL;
	mNumberChildren = 0;

	if (parent == NULL)
		mBase = root;
	else
		mBase = normal;

	mSlots = NULL;
	mSlotsTail = NULL;
	mChildren = NULL;
	mChildrenTail = NULL;
	mInstances = NULL;
	mInstancesTail = NULL;

	mSlotMap = new CTypedPtrMap<CMapStringToOb, CString, Slot*>;

	if (parent == NULL) {
		mType = F_NONE;
		return;
	}

	// set the type to that of the parent
	mType = parent->mType;

	// add in all of the parents slots to the new child
	count = parent->mSlotMap->GetCount();
	posi = parent->mSlotMap->GetStartPosition();

	for (i = 0; i < count; i++) {
		parent->mSlotMap->GetNextAssoc(posi, name, temps);
		CreateSlot(temps->mName, temps->mValue, (Frame*) temps->mReference, temps-
>mMod);
	}


}

Frame::Frame() {
	mpParent = NULL;
	mNumberSlots = 0;
	mChildren = NULL;
```

```
        mNumberChildren = 0;

        mSlotUnreferenced = FALSE;

        mBase = normal;
        mSlots = NULL;
        mSlotsTail = NULL;
        mChildren = NULL;
        mChildrenTail = NULL;
        mInstances = NULL;
        mInstancesTail = NULL;

        mSlotMap = new CTypedPtrMap<CMapStringToOb, CString, Slot*>;


}

Frame::~Frame() {
        Frame *pFrame1, *pFrame2;

        if (msTreeMap != NULL)
                msTreeMap->RemoveKey(LPCTSTR(mName));

        // delete all children
        pFrame1 = mChildren;
        while (pFrame1 != NULL) {
                pFrame2 = pFrame1->mNext;
                delete pFrame1;
                pFrame1 = pFrame2;
        }

        // delete all instances
        pFrame1 = mInstances;
        while (pFrame1 != NULL) {
                pFrame2 = pFrame1->mNext;
                delete pFrame1;
                pFrame1 = pFrame2;
        }

        // remove serialization map
        delete mSlotMap;
}

void Frame::Serialize( CArchive& ar ){

        CObject::Serialize(ar);

        if (ar.IsStoring())
        {
                ar << mName << mParentName << (LONG&) mNumberLeaves;
                mSlotMap->Serialize(ar);
                ar << (WORD&) mSlotUnreferenced;
        }
        else
        {
```

```
			ar >> mName >> mParentName >> (LONG&) mNumberLeaves;
			mSlotMap->Serialize(ar);
			ar >> (WORD&) mSlotUnreferenced;
		}
}

Slot *Frame::ResolveSlot(CString name) {
		Slot *pSlot;
		//Slot *pSlot = mSlots;

		/* old stuff
		// inspect all local slots for a match
		while (pSlot != NULL) {
			if (pSlot->mName == name)
				return(pSlot);
			pSlot = pSlot->mNext;
		}
		// slot not found in this frame

		// if this is the root Frame, return NULL
		if (mBase == root)
			return(NULL);
		*/

		if (mSlotMap->Lookup(LPCTSTR(name), pSlot))
			return (pSlot);
		else
			pSlot = mpParent->ResolveSlot(name);
		// call ResolveSlot using the Frame's parent
		//pSlot = mpParent->ResolveSlot(name);
		return(pSlot);
}

// adds a new slot to the frame
// creates a new Slot, sets its attributes based on the
// arguments and returns 1 if successful, -1 if Slot
// could not be created.
int Frame::CreateSlot(CString name, CString framename, Frame *reference, CString mod, BOOL
propagate) {
		Slot *pNSlot = new Slot;


		if (pNSlot == NULL)
			return(-1); // function failed



		// insert new slot into mSlotMap
		mSlotMap->SetAt(LPCTSTR(name), pNSlot);

		// set the attributes of the new slot
		pNSlot->mName = name;
		pNSlot->mValue = framename;
		pNSlot->mReference = reference;
		pNSlot->mMod = mod;
```

```
        // set mSlotUnreferenced
        if (reference == NULL)
                mSlotUnreferenced = TRUE;

        // increment mNumberSlots
        mNumberSlots++;

        if (propagate) {
                int i;
                Frame *tempf = mChildren;
                for (i = 0; i < mNumberChildren; i++) {
                        tempf->CreateSlot(name, framename, reference, mod);
                        tempf = tempf->mNext;
                }
        }

        return(1);
}


Frame *Frame::CreateChild(CString name) {
        Frame *pNFrame = new Frame(this);
        Frame *pTemp;

        // check creation of new frame
        if (pNFrame == NULL)
                return(NULL);

        // insert the new frame into the mChildren list
        // in alphabetical order
        pTemp = mChildren;

        if (pTemp != NULL) {
                while ((pTemp != NULL) && (name > pTemp->mName))
                        pTemp = pTemp->mNext;


                if (pTemp != NULL) {
                        // name is a repeat of one already defined in local frame
                        if (pTemp->mName == name) {
                                delete pNFrame;
                                return (NULL);
                        }

                        // insert new frame into mChildren
                        pNFrame->mNext = pTemp->mNext;
                        pTemp->mNext = pNFrame;
                        if (mChildrenTail == pTemp)
                                mChildrenTail = pNFrame;
                }
                else {
                        mChildrenTail->mNext = pNFrame;
                        mChildrenTail = pNFrame;
                        pNFrame->mNext = NULL;
```

```
				}
		}
		else {
				mChildren = pNFrame;
				mChildrenTail = pNFrame;
				pNFrame->mNext = NULL;
		}

		mNumberChildren++;

		pNFrame->mName = name;
		pNFrame->mParentName = mName;
		pNFrame->mpParent = this;

		return(pNFrame);
}

Frame *Frame::AddChild(Frame *pNFrame) {

		Frame *pTemp;
		CString name = pNFrame->mName;

		// check creation of new frame
		if (pNFrame == NULL)
				return(NULL);

		// insert the new frame into the mChildren list
		// in alphabetical order
		pTemp = mChildren;

		if (pTemp != NULL) {
				while ((pTemp != NULL) && (name > pTemp->mName))
						pTemp = pTemp->mNext;


				if (pTemp != NULL) {
						// name is a repeat of one already defined in local frame
						if (pTemp->mName == name) {
								return (NULL);
						}

						// insert new frame into mChildren
						pNFrame->mNext = pTemp->mNext;
						pTemp->mNext = pNFrame;
						if (mChildrenTail == pTemp)
								mChildrenTail = pNFrame;
				}
				else {
						mChildrenTail->mNext = pNFrame;
						mChildrenTail = pNFrame;
						pNFrame->mNext = NULL;
				}
		}
		else {
				mChildren = pNFrame;
```

```
                mChildrenTail = pNFrame;
                pNFrame->mNext = NULL;
        }

        mNumberChildren++;
        pNFrame->mpParent = this;

        return(pNFrame);

}


Frame *Frame::CreateInstance(CString name) {
        Frame *pNFrame = new Frame(this);
        Frame *pTemp;

        // check creation of new frame
        if (pNFrame == NULL)
                return(NULL);

        // insert the new frame into the mChildren list
        // in alphabetical order
        pTemp = mInstances;
        while ((pTemp != NULL) && (name < pTemp->mName))
                pTemp = pTemp->mNext;

        // name is a repeat of one already defined in local frame
        if (pTemp->mName == name) {
                delete pNFrame;
                return (NULL);
        }

        // insert new frame into mChildren
        pNFrame->mNext = pTemp->mNext;
        pTemp->mNext = pNFrame;
        if (mInstancesTail == pTemp)
                mInstancesTail = pNFrame;

        mNumberInstances++;

        pNFrame->mName = name;
        pNFrame->mBase = base;

        return(pNFrame);
}

Frame *Frame::FindFrame(CString name) {
        int i;
        Frame *tempf, *returnf;

        // Success condition
        if (mName == name)
                return(this);

        // search all children
```

```
	returnf = NULL;
	tempf = mChildren;
	for	(i = 0; i < mNumberChildren; i++) {
		returnf = tempf->FindFrame(name);
		if (returnf != NULL)
			return (returnf);
		tempf = tempf->mNext;
	}

	return (NULL);
}

int Frame::DeleteChild(Frame *f) {
	Frame *tempf, *tempfbefore;
	CString tempname = f->mName;



	if ((f->mName) == (mChildren->mName)) {
		mChildren = f->mNext;
		mNumberChildren--;
		delete f;
		return(1);
	}
	else {
		tempf = mChildren->mNext;
		tempfbefore = mChildren;
		while ((tempf != NULL) && (tempf->mName != tempname)) {
			tempfbefore = tempf;
			tempf = tempf->mNext;
		}

		if (tempf->mName != tempname)
			return(0);

		// remove tempf from list
		tempfbefore->mNext = tempf->mNext;
		if (mChildrenTail == tempf)
			mChildrenTail = tempfbefore;
		mNumberChildren--;
		delete tempf;
		return(1);
	}
}

int Frame::CountLeaves() {
	int i, temp = 0;
	Frame *tempf;

	// if this is a leaf, return 1,
	// else sum the number of leaves in the
	// child trees
	if (mNumberChildren == 0)
		temp = 1;
	else
```

```
        {
                tempf = mChildren;
                for (i = 0; i < mNumberChildren; i++) {
                        temp += tempf->CountLeaves();
                        tempf = tempf->mNext;
                }
        }

        // set mNumberLeaves and return
        mNumberLeaves = temp;
        return (temp);
}

void Frame::CalcSlotUnreferenced() {
        int count, i;
        POSITION posi;
        Slot *temps;
        CString name, value;
        Frame *fvalue, *tempf;

        // check to see if any slot values are unreferenced
        // in the current frame
        mSlotUnreferenced = FALSE;
        count = mSlotMap->GetCount();
        posi = mSlotMap->GetStartPosition();
        for (i = 0; i < count; i++) {
                mSlotMap->GetNextAssoc(posi, name, temps);
                value = temps->mValue;
                if (!msTreeMap->Lookup(LPCTSTR(value), fvalue)) {
                        mSlotUnreferenced = TRUE;
                        break;
                }
        }


        // call function in all children
        tempf = mChildren;
        for (i = 0; i < mNumberChildren; i++) {
                tempf->CalcSlotUnreferenced();
                tempf = tempf->mNext;
        }

}


Frame *Frame::DetermineType() {

        if ((mName == "Fact") || (mName == "Base") || (mName == "Goal") || (mName == "Rule") ||
(mName == "Root"))
                return(this);
        else
                return(mpParent->DetermineType());
}
```

```
void Frame::AddToFrameMap(CTypedPtrMap<CMapStringToOb, CString, Frame*> *pMap) {
        int i;
        Frame *tempf;

        // add self to map
        pMap->SetAt(mName, this->Copy());

        // add all children to map
        tempf = mChildren;
        for (i = 0; i < mNumberChildren; i++) {
                tempf->AddToFrameMap(pMap);
                tempf = tempf->mNext;
        }

}

// copy routine used for copying serializable values of a frame
Frame *Frame::Copy() {
        int i, count;
        POSITION pos;
        Slot *temps, *temps2;
        CString tname;

        // create a new frame
        Frame *tempf = new Frame();
        // set serializable values
        tempf->mName = mName;
        tempf->mParentName = mParentName;
        tempf->mNumberLeaves = mNumberLeaves;
        tempf->mSlotUnreferenced = mSlotUnreferenced;

        // copy the mSlotMap
        count = mSlotMap->GetCount();
        pos = mSlotMap->GetStartPosition();
        for (i = 0; i < count; i++) {
                mSlotMap->GetNextAssoc(pos, tname, temps);
                temps2 = temps->Copy();
                tempf->mSlotMap->SetAt(tname, temps2);
        }

        return(tempf);
}


void Frame::RemoveFromFrameMap(CTypedPtrMap<CMapStringToOb, CString, Frame*> *pMap) {
        int i;
        Frame *tempf = mChildren;

        for (i = 0; i < mNumberChildren; i++) {
                tempf->RemoveFromFrameMap(pMap);
                tempf = tempf->mNext;
        }

        mNumberChildren = 0;
        mChildren = NULL;
        pMap->RemoveKey(mName);
}

void Frame::ToCMap(CMap<CString, LPCTSTR, CString, CString&> *pMap) {
        int i;

        pMap->SetAt(mName, mName);

        // insert children's name into map
        Frame *tempf = mChildren;
        for (i = 0; i < mNumberChildren; i++) {
                tempf->ToCMap(pMap);
                tempf = tempf->mNext;
        }

}


void Frame::RemoveSlot(CString name) {
        Slot *temps;
        if (mSlotMap->Lookup(LPCTSTR(name), temps))
                mSlotMap->RemoveKey(LPCTSTR(name));
}


void Frame::MoveChild(Frame *f, Frame *dest) {
        Frame *tempf = mChildren;
        Frame *tempfnext = mChildren->mNext;

        if (f != mChildren) {
                while (tempfnext != NULL) {
                        if (tempfnext == f)
                                break;
                        tempf = tempfnext;
                        tempfnext = tempfnext->mNext;
                }
                // f not a child
                if (tempfnext == NULL)
                        return;
                else {
                        tempf->mNext = tempfnext->mNext;
                        if (mChildrenTail == tempfnext)
                                mChildrenTail = tempf;
                        tempf = tempfnext;
                }
        }
        else {
                if (mChildrenTail == tempf)
                        mChildrenTail = NULL;
                mChildren = tempf->mNext;
        }
        mNumberChildren--;
        dest->AddChild(f);
}
```

Thesisdoc.h: header file for Document class of Expert System Assistant application, providing much of the management of the tree of Frames and associated Slots through user interface events.

```
// thesidoc.h : interface of the CThesisDoc class
//
/////////////////////////////////////////////////////////////////////////////
#include "frame.h"
#include "dialogad.h"
#include <afxtempl.h>


class CThesisDoc : public CDocument
{
protected: // create from serialization only
	CThesisDoc();
	DECLARE_DYNCREATE(CThesisDoc)

// Attributes
public:
	Frame *mRoot;
	Frame *mSelectedFrame;
	Frame *mSelectedRoot;
	CTypedPtrMap<CMapStringToOb, CString, Frame*> *mTreeMap;
	CTypedPtrMap<CMapStringToOb, CString, Frame*> *mBaseMap;
// Operations
public:

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CThesisDoc)
	public:
	virtual BOOL OnNewDocument();
	//}}AFX_VIRTUAL

// Implementation
public:
	virtual ~CThesisDoc();
	virtual void Serialize(CArchive& ar);   // overridden for document i/o
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif
	void DisplayTree(Frame *f, int x, int y1, int y2, CView *view);
	void DisplayTreeRoot(Frame *f);
protected:


	void AddAllFramesToListBox(CDialogAddFrame *dialog, Frame *f);
	void DisplayFrame(Frame *f);
// Generated message map functions
protected:
	//{{AFX_MSG(CThesisDoc)
	afx_msg void OnDebugDisplaytree();
	afx_msg void OnKnowledgebaseAddframe();
	afx_msg void OnKnowledgebaseRemoveframe();
```

```
        afx_msg void OnKnowledgebaseAddslot();
        afx_msg void OnDebugDisplayframe();
        afx_msg void OnViewFacts();
        afx_msg void OnViewGoals();
        afx_msg void OnViewRules();
        afx_msg void OnViewBaseknowledge();
        afx_msg void OnFileSavebaseknowledge();
        afx_msg void OnFileLoadbaseknowledge();
        afx_msg void OnKnowledgebaseRemoveslot();
        afx_msg void OnKnowledgebaseMovesubtree();
        afx_msg void OnKnowledgebaseSetslotvalue();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
```

Thesisdoc.cpp: implementation file for application document, which provides all of the management routines for the knowledge base, from creation and deletion, to saving and loading, to implementation of routines called on user interface events such as button presses and menu commands.

```
// thesidoc.cpp : implementation of the CThesisDoc class
//

#include "stdafx.h"
#include "thesis.h"
#include "thesidoc.h"
#include "thesivw.h"
#include "dialogdi.h"
#include "dialogdt.h"
#include "dialogre.h"
#include "dialogas.h"
#include "dialogfr.h"
#include "dialores.h"
#include "diamoves.h"
#include "dsetslot.h"



#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif


// file filter for base knowledge
static char BASED_CODE szBaseFilter[] = "Base Knowledge (*.kbs) | *.kbs | All Files (*.*) | *.* ||";
/////////////////////////////////////////////////////////////////////////////
// CThesisDoc

IMPLEMENT_DYNCREATE(CThesisDoc, CDocument)

BEGIN_MESSAGE_MAP(CThesisDoc, CDocument)
	//{{AFX_MSG_MAP(CThesisDoc)
	ON_COMMAND(ID_DEBUG_DISPLAYTREE, OnDebugDisplaytree)
	ON_COMMAND(ID_KNOWLEDGEBASE_ADDFRAME, OnKnowledgebaseAddframe)
	ON_COMMAND(ID_KNOWLEDGEBASE_REMOVEFRAME,
OnKnowledgebaseRemoveframe)
	ON_COMMAND(ID_KNOWLEDGEBASE_ADDSLOT, OnKnowledgebaseAddslot)
	ON_COMMAND(ID_DEBUG_DISPLAYFRAME, OnDebugDisplayframe)
	ON_COMMAND(ID_VIEW_FACTS, OnViewFacts)
	ON_COMMAND(ID_VIEW_GOALS, OnViewGoals)
	ON_COMMAND(ID_VIEW_RULES, OnViewRules)
	ON_COMMAND(ID_VIEW_BASEKNOWLEDGE, OnViewBaseknowledge)
	ON_COMMAND(ID_FILE_SAVEBASEKNOWLEDGE, OnFileSavebaseknowledge)
	ON_COMMAND(ID_FILE_LOADBASEKNOWLEDGE, OnFileLoadbaseknowledge)
	ON_COMMAND(ID_KNOWLEDGEBASE_REMOVESLOT, OnKnowledgebaseRemoveslot)
	ON_COMMAND(ID_KNOWLEDGEBASE_MOVESUBTREE, OnKnowledgebaseMovesubtree)
	ON_COMMAND(ID_KNOWLEDGEBASE_SETSLOTVALUE, OnKnowledgebaseSetslotvalue)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
/////////////////////////////////////////////////////////////////////////////
// CThesisDoc construction/destruction

CThesisDoc::CThesisDoc()
{
	Frame *tempf;
	CString tempname = " ";
	// create the root and set its name
	mRoot = new Frame(NULL);
	mRoot->mName = "Root";
	// add the root to the mTreeMap
	mTreeMap = new CTypedPtrMap<CMapStringToOb, CString, Frame*>;
	mTreeMap->SetAt("Root", mRoot);

	// creat the mBaseMap
	mBaseMap = new CTypedPtrMap<CMapStringToOb, CString, Frame*>;

	mRoot->msTreeMap = mTreeMap;

	mSelectedFrame = mRoot;
	mSelectedRoot = mRoot;

	tempf = mRoot->CreateChild("Base");
	tempf->mType = F_BASE;
	mTreeMap->SetAt("Base", tempf);

	tempf = mRoot->CreateChild("Rule");
	tempf->mType = F_RULE;
	mTreeMap->SetAt("Rule", tempf);

	tempf = mRoot->CreateChild("Fact");
	tempf->mType = F_FACT;
	mTreeMap->SetAt("Fact", tempf);

	tempf = mRoot->CreateChild("Goal");
	tempf->mType = F_GOAL;
	mTreeMap->SetAt("Goal", tempf);

	mRoot->CountLeaves();
	mRoot->CalcSlotUnreferenced();
}

CThesisDoc::~CThesisDoc()
{

	delete mRoot;
	delete mTreeMap;
	delete mBaseMap;

}



BOOL CThesisDoc::OnNewDocument()
```

```
{
        Frame *tempf;

        if (!CDocument::OnNewDocument())
                return FALSE;

        // TODO: add reinitialization code here
        // (SDI documents will reuse this document)
        delete mRoot;
        mTreeMap->RemoveAll();
        mRoot = new Frame(NULL);
        mRoot->mName = "Root";
        mRoot->mParentName = "";
        mTreeMap->SetAt("Root", mRoot);

        mRoot->msTreeMap = mTreeMap;

        mSelectedFrame = mRoot;
        mSelectedRoot = mRoot;

        // create Base frame
        tempf = mRoot->CreateChild("Base");
        tempf->mType = F_BASE;
        mTreeMap->SetAt("Base", tempf);

        tempf = mRoot->CreateChild("Rule");
        tempf->mType = F_RULE;
        mTreeMap->SetAt("Rule", tempf);

        tempf = mRoot->CreateChild("Fact");
        tempf->mType = F_FACT;
        mTreeMap->SetAt("Fact", tempf);

        tempf = mRoot->CreateChild("Goal");
        tempf->mType = F_GOAL;
        mTreeMap->SetAt("Goal", tempf);

        mRoot->CountLeaves();
        mRoot->CalcSlotUnreferenced();

        return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CThesisDoc serialization

void CThesisDoc::Serialize(CArchive& ar)
{
        if (ar.IsStoring())
        {
                mTreeMap->Serialize(ar);
        }
        else
        {
                int count, i;
                Frame *ftemp, *ftempparent;
                POSITION pos;
                CString tempname, tempparentname;
                mTreeMap->Serialize(ar);

                // recontstruct the tree from the map
                count = mTreeMap->GetCount();
                pos = mTreeMap->GetStartPosition();
                for     (i = 0; i < count; i++)     {
                        mTreeMap->GetNextAssoc(pos, tempname, ftemp);
                        tempparentname = ftemp->mParentName;
                        if (tempname != "") {
                                if (mTreeMap->Lookup(LPCTSTR(tempparentname), ftempparent))
                                        ftempparent->AddChild(ftemp);
                        }
                }
                mTreeMap->Lookup("Root", mRoot);
                mSelectedRoot = mRoot;
                mSelectedFrame = mRoot;
                mRoot->CountLeaves();
                mRoot->CalcSlotUnreferenced();

        }
}

/////////////////////////////////////////////////////////////////////////////
// CThesisDoc diagnostics

#ifdef _DEBUG
void CThesisDoc::AssertValid() const
{
        CDocument::AssertValid();
}

void CThesisDoc::Dump(CDumpContext& dc) const
{
        CDocument::Dump(dc);
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CThesisDoc commands



void CThesisDoc::OnDebugDisplaytree()
{

        CDialogDTree dialog;
        Frame *f;
        int DResult;

        DResult = dialog.DoModal();

        if (DResult == IDCANCEL)
```

```
			return;

		mRoot->CountLeaves();

		f = mRoot->FindFrame(dialog.m_Name);
		if (f != NULL)
			DisplayTreeRoot(f);
		else
			DisplayTreeRoot(mRoot);

}

void CThesisDoc::DisplayTreeRoot(Frame *f) {
		POSITION p = GetFirstViewPosition();
		CThesisView *view = (CThesisView*) GetNextView(p);
		CRect rect;
		view->GetClientRect(&rect);
		int ys;
		mSelectedRoot = f;
		view->mViewMode = TREE;
		view->Clear();
		ys = rect.Height();
		DisplayTree(f, 5, 20, ys - 20, view);
}

void CThesisDoc::DisplayTree(Frame *f, int x, int y1, int y2, CView *view) {
		int i, numchildren = f->mNumberChildren, pleaves, cleaves;
		int ystart, xnew, y, ynew, yrange, ymid, yend;
		Frame *childlist;
		CThesisView *tview = (CThesisView*) view;
		yrange = y2 - y1;
		y = (int)(yrange/2) - 15 + y1;
		xnew = x + 110;
		ystart = y1;

		tview->DrawFrame(f, x, y);
		childlist = f->mChildren;
		pleaves = f->mNumberLeaves;
		for (i = 0; i < numchildren; i++) {
			cleaves = childlist->mNumberLeaves;
			ynew = (int)((cleaves * yrange) / (pleaves * 2));

			TRACE("parent leaves:  %i,  child leaves:  %i\n", pleaves, cleaves);
			ymid = ystart + ynew;
			yend = ymid + ynew;
			TRACE("ystart:  %i  ynew:  %i  ymid:  %i  yend:  %i\n", ystart, ynew, ymid, yend);
			tview->DrawFrameLine(x, y, xnew, ymid - 15);
			DisplayTree(childlist, xnew, ystart, yend, view);
			ystart = yend;
			childlist = childlist->mNext;
		}
}

void CThesisDoc::AddAllFramesToListBox(CDialogAddFrame *dialog, Frame *f) {
		int i;
```

```
	Frame *ftemp = f->mChildren;

	//dialog->m_List.AddString(LPCTSTR(f->mName));

	for (i = 0; i < f->mNumberChildren; i++) {
		AddAllFramesToListBox(dialog, ftemp);
		ftemp = ftemp->mNext;
	}
}

void CThesisDoc::OnKnowledgebaseAddframe()
{
	Frame *Parent, *tempf, *tempf2, *Children;
	int DResult, i;
	POSITION p = GetFirstViewPosition();
	CThesisView *v = (CThesisView*) GetNextView(p);
	CRect rect;

	v->GetWindowRect(&rect);

	CDialogAddFrame dialog;


	//AddAllFramesToListBox(&dialog, mRoot);

	DResult = dialog.DoModal();

	if (DResult == IDCANCEL)
		return;

	// check to see if name is already in use
	if (mTreeMap->Lookup(LPCTSTR(dialog.m_Name), tempf)) {
		MessageBox(NULL, "Name already exists!", "Error", MB_APPLMODAL);
		return;
	}


	//if (Parent != NULL)
	if (mTreeMap->Lookup(LPCTSTR(dialog.m_Parent2), Parent) && dialog.m_Name != "") {

		// insert before a single child
		CString SingleChild = dialog.m_SingleChild;
		if ((SingleChild != "") && (!dialog.m_AllChildren)) {
			if (!mTreeMap->Lookup(LPCTSTR(SingleChild), Children)) {
				MessageBox(NULL, "Single Child not found!", "Error",
MB_APPLMODAL);
				return;
			}
			tempf = Parent->mChildren;
			tempf2 = tempf->mNext;
			if (Children == tempf)
				Parent->mChildren = tempf2;
			else {
				int found = 0;
				for (i = 1; i < Parent->mNumberChildren; i++)  {
                        }
                                        if (tempf2 == Children) {
                                                found = 1;
                                                break;
                                        }
                                        else {
                                                tempf = tempf2;
                                                tempf2 = tempf2->mNext;
                                        }
                                }
                                // child is not if frame
                                if (found == 0) {
                                        MessageBox(NULL, "Single Child not in Parent!", "Error",
MB_APPLMODAL);
                                        return;
                                }
                                // child was found
                                // remove child from frame;
                                tempf->mNext = tempf2->mNext;
                                if (Parent->mChildrenTail == tempf2)
                                        Parent->mChildrenTail = tempf;
                                        tempf = tempf2;
                        }
                        Parent->mNumberChildren--;
                        tempf2 = tempf;
                        // child is not removed from parent and is referenced by tempf2
                        tempf = Parent->CreateChild(dialog.m_Name);
                        // add single child to new frame
                        tempf->AddChild(tempf2);
                }

                // insert new frame between parent and all children of the Parent
                if (dialog.m_AllChildren) {
                        // save children information from Parent
                        Children = Parent->mChildren;
                        i = Parent->mNumberChildren;
                        tempf2 = Parent->mChildrenTail;
                        // remove all Children from the parent
                        Parent->mChildren = NULL;
                        Parent->mChildrenTail = NULL;
                        Parent->mNumberChildren = 0;
                        // create the new frame
                        tempf = Parent->CreateChild(dialog.m_Name);
                        // transfer the old child information to the new frame
                        tempf->mNumberChildren = i;
                        tempf->mChildren = Children;
                        tempf->mChildrenTail = tempf2;
                        // set the children's new parent to tempf
                        for (i = 0; i < tempf->mNumberChildren; i ++) {
                                Children->mpParent = tempf;
                                Children->mParentName = dialog.m_Name;
                                Children = Children->mNext;
                        }
                }

                if ((!dialog.m_AllChildren) && (dialog.m_SingleChild == ""))
```

```
                    tempf = Parent->CreateChild(dialog.m_Name);

                // new frame is tempf
                mTreeMap->SetAt(LPCTSTR(dialog.m_Name), tempf);
                mSelectedRoot = tempf->DetermineType();
        }
        mRoot->CountLeaves();
        mRoot->CalcSlotUnreferenced();

        DisplayTreeRoot(mSelectedRoot);
}

void CThesisDoc::OnKnowledgebaseRemoveframe()
{
        int i;
        CString rname;
        Frame *ftemp, *ftempparent;
        CDialogRemoveFrame dialog;
        int DResult;
        DResult = dialog.DoModal();
        if (DResult == IDCANCEL)
                return;

        rname = dialog.m_Name;

        if ((rname != "") && (rname != "Root")) {
                // old!!! ftemp = mRoot->FindFrame(rname);

                if (mTreeMap->Lookup(LPCTSTR(rname), ftemp)) {
                        ftempparent = ftemp->mpParent;
                        i =     ftempparent->DeleteChild(ftemp);
                        mTreeMap->RemoveKey(LPCTSTR(rname));
                        mRoot->CountLeaves();
                        mRoot->CalcSlotUnreferenced();

                }
        }

        DisplayTreeRoot(mRoot);


}

void CThesisDoc::OnKnowledgebaseAddslot()
{
        Frame *tempf, *tempreference;
        CDialogAddslot dialog;
        CString framename;
        int DResult;

        // Show dialog
        DResult = dialog.DoModal();
        if (DResult == IDCANCEL)
                return;
        framename = dialog.m_Frame;
        if ((framename == "") || (dialog.m_SlotName == ""))
                return;

        // Find specified frame
        if (!mTreeMap->Lookup(LPCTSTR(framename), tempf))
                return;
        else
                mSelectedFrame = tempf;
        /* old stuff
        tempf = mRoot->FindFrame(framename);
        if (tempf == NULL)
                return;
        else
                mSelectedFrame = tempf
        */;

        tempreference = mRoot->FindFrame(dialog.m_Value);
        tempf->CreateSlot(dialog.m_SlotName, dialog.m_Value, tempreference, dialog.m_Modifier);

        DisplayFrame(tempf);
}

void CThesisDoc::DisplayFrame(Frame *f) {
        POSITION pos = GetFirstViewPosition();
        CThesisView *view = (CThesisView*) GetNextView(pos);
        CDC *pDC = view->GetDC();
        int count, x, y, i, ystep = 90;
        Slot *temps;
        POSITION posi;
        CString name, value;
        Frame *fvalue;

        // clear display
        view->Clear();
        // display frame name
        pDC->TextOut(5, 3, "Frame Name: " + f->mName);
        // set the update mode
        view->mViewMode = FRAME;
        mSelectedFrame = f;

        // set initial x and y values for Slot info positioning
        x = 5;
        y = 25;

        count = f->mSlotMap->GetCount();
        posi = f->mSlotMap->GetStartPosition();
        for (i = 0; i < count; i++) {
                f->mSlotMap->GetNextAssoc(posi, name, temps);
                // update reference information
                value = temps->mValue;
                if (mTreeMap->Lookup(LPCTSTR(value), fvalue))
                        temps->mReference = fvalue;
                else
                        temps->mReference = NULL;
                view->DisplaySlot(temps, x, y);
```

```
				y += ystep;
		}

		/* old stuff
		temps = f->mSlots;
		for (i = 0; i < f->mNumberSlots; i++) {
				view->DisplaySlot(temps, x, y);
				temps = temps->mNext;
				y += ystep;
		}
		*/

}

void CThesisDoc::OnDebugDisplayframe()
{
		CDialogFrameInfo dialog;
		Frame *f;
		int DResult;

		DResult = dialog.DoModal();

		if (DResult == IDCANCEL)
				return;

		// old!!!  f = mRoot->FindFrame(dialog.m_Name);

		if (!mTreeMap->Lookup(LPCTSTR(dialog.m_Name), f))
				return;

		DisplayFrame(f);

}

void CThesisDoc::OnViewFacts()
{

		Frame *f;


		f = mRoot->FindFrame("Fact");
		if (f != NULL) {
				mSelectedRoot = f;
				DisplayTreeRoot(f);
		}
		else {
				mSelectedRoot = mRoot;
				DisplayTreeRoot(mRoot);
		}

}

void CThesisDoc::OnViewGoals()
{
		Frame *f;
```

```
	f = mRoot->FindFrame("Goal");
	if (f != NULL) {
		mSelectedRoot = f;
		DisplayTreeRoot(f);
	}
	else {
		mSelectedRoot = mRoot;
		DisplayTreeRoot(mRoot);
	}
}

void CThesisDoc::OnViewRules()
{
	Frame *f;


	f = mRoot->FindFrame("Rule");
	if (f != NULL) {
		mSelectedRoot = f;
		DisplayTreeRoot(f);
	}
	else {
		mSelectedRoot = mRoot;
		DisplayTreeRoot(mRoot);
	}
}

void CThesisDoc::OnViewBaseknowledge()
{
	Frame *f;


	f = mRoot->FindFrame("Base");
	if (f != NULL) {
		mSelectedRoot = f;
		DisplayTreeRoot(f);
	}
	else {
		mSelectedRoot = mRoot;
		DisplayTreeRoot(mRoot);
	}
}

void CThesisDoc::OnFileSavebaseknowledge()
{
	CFileDialog *filedialog;
	int dresult;
	CString filename;

	// save file dialog
	filedialog = new CFileDialog(FALSE, ".kbs", NULL, OFN_HIDEREADONLY |
OFN_OVERWRITEPROMPT, szBaseFilter);
	dresult = filedialog->DoModal();  // OK or Cancel
	if (dresult == IDCANCEL)
		return;

	if (dresult == IDOK) {
		// open file specified by dialog
		filename = filedialog->GetPathName();

		CFile *pFile = new CFile(LPCTSTR(filename), CFile::modeCreate |
CFile::modeReadWrite);

		// create archive for storing using pFile
		CArchive Ar(pFile, CArchive::store);

		// load mBaseMap with all Base frames
		Frame *tempf;
		mTreeMap->Lookup("Base", tempf);
		tempf->AddToFrameMap(mBaseMap);
		mBaseMap->Serialize(Ar);
		Ar.Close();
		pFile->Close();
		delete pFile;
		mBaseMap->RemoveAll();
	}

	delete filedialog;

}

void CThesisDoc::OnFileLoadbaseknowledge()
{
	CFileDialog *filedialog;
	int dresult;
	CString filename;

	// load file dialog
	filedialog = new CFileDialog(TRUE, ".kbs", NULL, OFN_HIDEREADONLY |
OFN_OVERWRITEPROMPT, szBaseFilter);
	dresult = filedialog->DoModal();  // OK or Cancel

	if (dresult == IDCANCEL)
		return;

	if (dresult == IDOK) {
		CMap<CString, LPCTSTR, CString, CString&> TempMap;
		POSITION pos;
		int i, count;
		CString rname, vname;
		// open file specified by dialog
		filename = filedialog->GetPathName();

		CFile *pFile = new CFile(LPCTSTR(filename), CFile::modeReadWrite);

		// create archive for loading using pFile
		CArchive Ar(pFile, CArchive::load);
```

```
                mBaseMap->Serialize(Ar);

                // copy all elements of mBaseMap and add into mTreeMap

                // just replace for now - dialog later
                // delete Base tree
                Frame *tempf;
                mTreeMap->Lookup("Base", tempf);
                tempf->ToCMap(&TempMap);
                mRoot->DeleteChild(tempf);
                pos = TempMap.GetStartPosition();
                count = TempMap.GetCount();
                for (i = 0; i < count; i++) {
                        TempMap.GetNextAssoc(pos, rname, vname);
                        mTreeMap->RemoveKey(rname);
                }


                // add new elements
                pos = mBaseMap->GetStartPosition();

                CString tname;
                count = mBaseMap->GetCount();
                for (i = 0; i < count; i++) {
                        mBaseMap->GetNextAssoc(pos, tname, tempf);
                        mTreeMap->SetAt(tname, tempf->Copy());
                }
                // reconstuct parent links
                Frame *ftempparent;
                CString tempparentname;
                pos = mBaseMap->GetStartPosition();
                for (i = 0; i < count; i++) {
                        mBaseMap->GetNextAssoc(pos, tname, tempf);
                        tempparentname = tempf->mParentName;
                        mTreeMap->Lookup(LPCTSTR(tname), tempf);
                        if (tname != "") {
                                if (mTreeMap->Lookup(LPCTSTR(tempparentname), ftempparent))
                                        ftempparent->AddChild(tempf);
                        }
                }

                Ar.Close();
                pFile->Close();
                delete pFile;

        }
        mTreeMap->Lookup("Base", mSelectedRoot);
        mRoot->CountLeaves();
        mRoot->CalcSlotUnreferenced();
        DisplayTreeRoot(mSelectedRoot);
        delete filedialog;
}

void CThesisDoc::OnKnowledgebaseRemoveslot()
{
	CDialogRemoveSlot dialog;
	int dresult;
	Frame *tempf;
	Slot *temps;

	dresult = dialog.DoModal();

	if (dresult == IDOK) {
		if ((dialog.m_Frame == "") || (dialog.m_Name == ""))
			return;
		if (!mTreeMap->Lookup(LPCTSTR(dialog.m_Frame), tempf)) {
			MessageBox(NULL, "Frame does not exist!", "Error", MB_APPLMODAL);
			return;
		}
		if (tempf->mSlotMap->Lookup(LPCTSTR(dialog.m_Name), temps)) {
			tempf->RemoveSlot(dialog.m_Name);
			mRoot->CalcSlotUnreferenced();
			DisplayFrame(tempf);
		}
		else
			MessageBox(NULL, "Slot does not exist!", "Error", MB_APPLMODAL);
	}
}


void CThesisDoc::OnKnowledgebaseMovesubtree()
{
	CDiaMoveSubtree dialog;
	int dresult;
	Frame *f, *f2;

	// call dialog
	dresult = dialog.DoModal();

	if (dresult == IDOK) {
		if ((mTreeMap->Lookup(LPCTSTR(dialog.m_Name), f)) && (mTreeMap-
>Lookup(LPCTSTR(dialog.m_NameTo), f2)))
			f->mpParent->MoveChild(f, f2);

		mRoot->CountLeaves();
		DisplayTreeRoot(mSelectedRoot);
	}
}

void CThesisDoc::OnKnowledgebaseSetslotvalue()
{
	CDSetSlotValue dialog;
	Frame *f, *fref;
	Slot *s;
	int dresult;

	dresult = dialog.DoModal();


	if (dresult == IDOK) {
```

```
                    // lookup FrameName
                    if (!mTreeMap->Lookup(LPCTSTR(dialog.m_FrameName), f)) {
                              MessageBox(NULL, "Frame not found!", "Error", MB_APPLMODAL);
                              return;
                    }
                    // lookup slot in frame
                    if (!f->mSlotMap->Lookup(LPCTSTR(dialog.m_SlotName), s)) {
                              MessageBox(NULL, "Slot not in Frame", "Error", MB_APPLMODAL);
                              return;
                    }
                    // set value
                    s->mValue = dialog.m_Value;
                    // lookup and set reference
                    if (mTreeMap->Lookup(LPCTSTR(dialog.m_Value), fref))
                              s->mReference = fref;
                    else
                              s->mReference = NULL;

                    f->CalcSlotUnreferenced();
                    DisplayFrame(f);
          }

}
```