1993

Improving explanation facilities in expert systems.

Hien My. Bui
University of Windsor

IMPROVING EXPLANATION FACILITIES IN EXPERT SYSTEMS

by

Hien My Bui

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree of
Master of Science at the
University of Windsor

Windsor, Ontario, Canada
September, 1993

ISBN 0-315-87376-0

[1] Chapter 2 and 3 and Appendix A and B are surveys. Substantial material in these surveys was directly taken from source references and summaries. No right is assumed in this case.

Abstract

One of the important features of expert systems is to provide explanations suitable to the user's needs. Thus it is desirable to construct an expert system which can flexibly give explanations to different levels of users. Many existing expert systems are only designed for one type of user. In this work, using the Designer's Assistant system as the platform, we build up the user modelling and text retrieval components so that the system can adapt to different levels of the user's expertise. In the implementation, a model is built for each user using the system based on his/her expertise, goals, needs, etc. Text documentation is retrieved for a particular user based on his/her model and is used as explanations. The result is that different explanations are provided for different users, depending on their needs.

To my parents, my husband and baby.

Acknowledgments

I would like to acknowledge the guidance and support provided by Dr. S. Bandyopadhyay and Dr. W. Miller for their role as my supervisory committee member. Specifically, I am very grateful to Dr. J. Morrissey for taking the time and effort in guiding me through this work. She has made numerous valuable suggestions and comments on the thesis. Special thanks must go to Phu, Arunita, Tibor, and Steve for all their help throughout the various stages of this work. Last but not least, I would like to thank Dr. R. Frost for his help and encouragement throughout this work.

TABLE OF CONTENTS

Abstract iv

Acknowledgments vi

List of Figures xi

1 INTRODUCTION 1
1.1 Objective of Thesis 1
1.2 Motivation 1
1.3 Solution Outline 2
1.4 Thesis Organization 3

2 EXPLANATION FACILITIES IN EXPERT SYSTEMS 5
2.1 Introduction 5
2.2 Overview of Expert Systems and Explanation Facilities 5
2.2.1 What is an Expert System 5
2.2.2 Components of Expert Systems 6
2.2.3 What is an Explanation Facility 9
2.2.4 Why is an Explanation Facility Useful 9
2.2.5 Components of Explanation Facility 10
2.2.5.1 A Why explanation 10
2.2.5.2 A How explanation 10
2.2.5.3 A What explanation 10
2.2.6 Example 11
2.3 Aspects of Explanation Facilities 14
2.3.1 Content of explanations 14
2.3.2 User models 15
2.3.3 The human-computer interface 15
2.4 Analysis of Research 16
2.4.1 Content of explanations 16
2.4.1.1 Introspection — The First Approach . . . 16
2.4.1.2 Introspection — The Second Approach . 17
2.4.1.3 Introspection — The Third Approach . . . 18
2.4.1.4 Explanation by concocting 19

2.4.2 User models 19

2.4.3 The human-computer interface 20

2.5 Future Trends 21

2.6 Conclusion 22

3 USER MODELLING IN EXPLANATION FACILITIES IN EXPERT SYSTEMS 24

3.1 Introduction 24

3.2 Frameworks for User Modelling 24

3.2.1 User modelling classification 27

3.2.1.1 Carbonell's approach 27

3.2.1.1.1 Empirical quantitative models:. . . . 27

3.2.1.1.2 Analytical cognitive models: 28

3.2.1.2 Rich's approach 28

3.2.1.2.1 Canonical vs. individual model . . 29

3.2.1.2.2 Explicit vs. implicit 30

3.2.1.2.3 Long-term vs. short-term models . 31

3.3 Techniques for Building User Models 33

3.3.1 Inferring individual facts 34

3.3.2 Using stereotypes to infer many things at a time 36

3.4 Review of Current Systems 37

3.4.1 Representing User Models 37

3.4.2 Scalar Models 39

3.4.3 Ad Hoc Modelling Systems 40

3.4.4 Profile Models 41

3.4.5 Overlay Models 42

3.4.6 Process Models 44

3.5 Future Work 45

3.6 Conclusion 46

4 IMPLEMENTATION 48
4.1 Introduction 48
4.2 Implementation Context 49
4.2.1 What is the Designer's Assistant? 49
4.2.2 Why Choose the Designer's Assistant as an Experimental Platform? 50
4.3 User Modelling 51
4.3.1 The Ideal User Model 51
4.3.2 The Design's Assistant User Model 51
4.3.3 How do we build the model 52
4.3.4 How the model is maintained 54
4.4 Information Retrieval Techniques 57
4.4.1 How Keywords are Acquired and Used 57
4.4.2 Types of IR Techniques Used 59
4.5 User Interface 59
4.6 Conclusions 60

5 CONCLUSIONS AND FUTURE WORK 62
5.1 Summary and Conclusions 62
5.2 Future work 63

6 65
. . . . 65
A APPENDIX A — INFORMATION RETRIEVAL 73
A.1 Introduction 73
A.2 Information Retrieval Systems 73
A.2.1 Data base management systems 74
A.2.2 Question-Answering System 75
A.2.3 Information Retrieval Systems 75
A.3 Activities in Information Retrieval Systems 76
A.3.1 Information Analysis 76
A.3.1.1 The inverse document frequency weight. 80
A.3.1.2 The signal-noise ratio 80
A.3.1.3 The term discrimination value 82
A.3.1.4 Term Thesaurus 87

A.3.2 Information Organization and Search 89
A.3.3 Query Formulations 91
A.3.3.1 Boolean query formation and search . . . 91
A.3.3.2 Matching Functions and Serial Search . 94
A.3.3.3 Probabilistic Information Retrieval 96
A.3.4 Information Retrieval and Dissemination 98
A.4 Conclusion 100
B APPENDIX B — USER INTERFACE FOR EXPERT SYSTEMS 102
B.1 Introduction 102
B.2 The Requirements for User Interfaces to Expert Systems 103
B.2.1 The natural idiom 104
B.2.2 Immediate feedback 105
B.2.3 Recoverability 105
B.2.4 The appropriate interaction styles 106
B.2.5 Multiple interfaces to the same knowledge . . 106
B.3 Aspects of the User Interface 107
B.3.1 Dialogue control 107
B.3.2 Natural language processing 108
B.3.3 New technology 108
B.4 Review of Current Building Tools to Support User Interfaces 109
B.4.1 The KEE interface 110
B.4.2 The ActiveImages package 110
B.4.3 Slotgraph 111
B.4.4 SimKit interface tools 111
B.4.5 Common Windows 111
B.4.6 KEEpictures Toolkit 112
B.5 A General Approach to Implementing Interface Tools 112
B.6 Conclusion 114
C PROGRAM LISTING 116
D VITA AUCTORIS 200

List of Figures

Figure 1	Diagram of Components of Expert Systems	7
Figure 2	An overlay model	43
Figure 3	Excerpt from Typical Stop List	84
Figure 4	Typical term deletion algorithm (data for 1,033 documents in medicine)	86
Figure 5	Typical Thesaurus Excerpt	88
Figure 6	A sequence of refines query statement in the Stair system.	100

Chapter 1 INTRODUCTION

1.1 Objective of Thesis

The objective of this thesis is to investigate how the quality of explanations given in expert systems can be improved by the use of user modelling techniques, information retrieval techniques and user interface techniques.

To accomplish the objective, two main tasks were undertaken: a survey of related areas and an implementation. Areas surveyed include: explanation facilities in expert systems, user modelling in explanation facilities in expert systems, user interfaces for expert systems and finally information retrieval systems. To test our ideas, we have implemented different user models in an existing expert system for FIR filter design [Sar 90] which was limited to only one type of user.

1.2 Motivation

A major factor which can affect the ease with which people use an expert system is the ability of the system to tailor its behavior to the specific needs of an individual user. Many expert systems are only designed for one type of user. They require the user to be familiar with the concepts and terms of a domain. The most common form of explanation facility in use is non-interactive and consists of printing out a trace of the rules being used or displaying canned text explanations. Although rule traces can be useful for debugging, they rarely provide an acceptable explanation for the user. Canned text, another form of explanation, involves anticipating user questions in advance and storing appropriate answers in English text. The major problem with the use of canned text is that it is unlikely that all user questions will be anticipated in advance and this makes it difficult to provide more advanced forms of explanation.

In short, current explanation facilities are not adequate and are difficult to use for naive users. We suggest that they can be improved by using techniques from user modelling, information retrieval and user interfaces.

1.3 Solution Outline

The ability of an expert system to provide explanations is linked to the performance of three main techniques used in the expert system: user modelling, information retrieval and user interface. User modelling techniques can assist in finding relevant and pertinent help for different types of users. Information retrieval techniques can be used to retrieve meaningful passages of text which are the basis of explanations. User interface techniques can be used to improve the interaction between the user and the system..

In our work, we make use of these three techniques to provide a better explanation for the user of expert systems.

First of all, user models are built by combining explicit and implicit approaches. In this work, we consider two stereotypes of users: expert and naive.

At the initial stage the stereotype is formed. Then an individual model is developed as the user uses the expert system. During the design process the user is questioned about his/her goal, level of expertise, type of explanations preferred, etc. and his/her behavior is observed.

When a model is built for a specific user, it is used to select a set of suitable keywords which are associated with rules in the knowledge base. Based on these keywords, simple information retrieval information techniques are applied to extract relevant text which is used as the content of explanations. In this way, different explanations are provided for different users, depending on their needs.

Finally, user interface techniques are employed to allow the user to be interactively involved in the designing process. It helps the user to use the system more efficiently and effectively.

Therefore, with our solution, the user of an expert system can obtain meaningful explanations suitable to his/her background. Moreover, the employment of user interface techniques make the system more user-friendly and easier to use.

1.4 Thesis Organization

In chapter two we provide background information on explanation facilities in expert systems. It contains a brief description of an expert system, its components, the usefulness of explanation facilities and aspects of explanation facilities. Chapter three describes user modelling in explanation facilities in expert systems. Frame works for user modelling and classification of user modelling are presented along with techniques for building user models and reviewing of current systems. Chapter four provides the description of the actual implementation of the system. We also provide reviews of information retrieval systems and user interfaces of expert systems in the appendixes A and B respectively. Appendix C contains a listing of the program.

Chapter 2 EXPLANATION FACILITIES IN EXPERT SYSTEMS

2.1 Introduction

In the Artificial Intelligence area, researchers have considered the development of explanation facilities to be one of the most valuable contributions towards developing a methodology for building expert systems. Most of the research activity is concerned with improving the quality of explanation produced.

The ability to explain its reasoning is usually considered an important component of any expert system. An explanation facility is useful on several levels: it can help developers to test, monitor and debug the system during program development; it can assure the domain expert that the system's knowledge and reasoning process is appropriate; and it can instruct the naive user or student about the knowledge in the system.

The problem of producing explanations can be viewed in a framework of three major considerations: the content of explanations, user modeling and the human-computer interface. In the following sections a discussion and review of work carried out in each area is presented.

2.2 Overview of Expert Systems and Explanation Facilities

2.2.1 What is an Expert System

In recent years, *expert systems* have been produced in many different areas of expertise such as diagnosis in various areas of medicine, chemistry, electronics, geology, management, oil and mineral exploration and income tax. An expert system is an intelligent computer program which uses knowledge and inference procedures to solve problems that usually require significant human expertise for their solution. The knowledge necessary to perform at such a level, and the inference procedures used, can be thought of as a model of the expertise of the best practitioners in the field.

2.2.2 Components of Expert Systems

An expert system usually consists of the following separate components:

. *Knowledge base:* This is the body of facts, rules and heuristics which forms the basis of a knowledge system.

. *Inference engine:* This is the part of a knowledge based system that contains the procedures for reaching a conclusion.

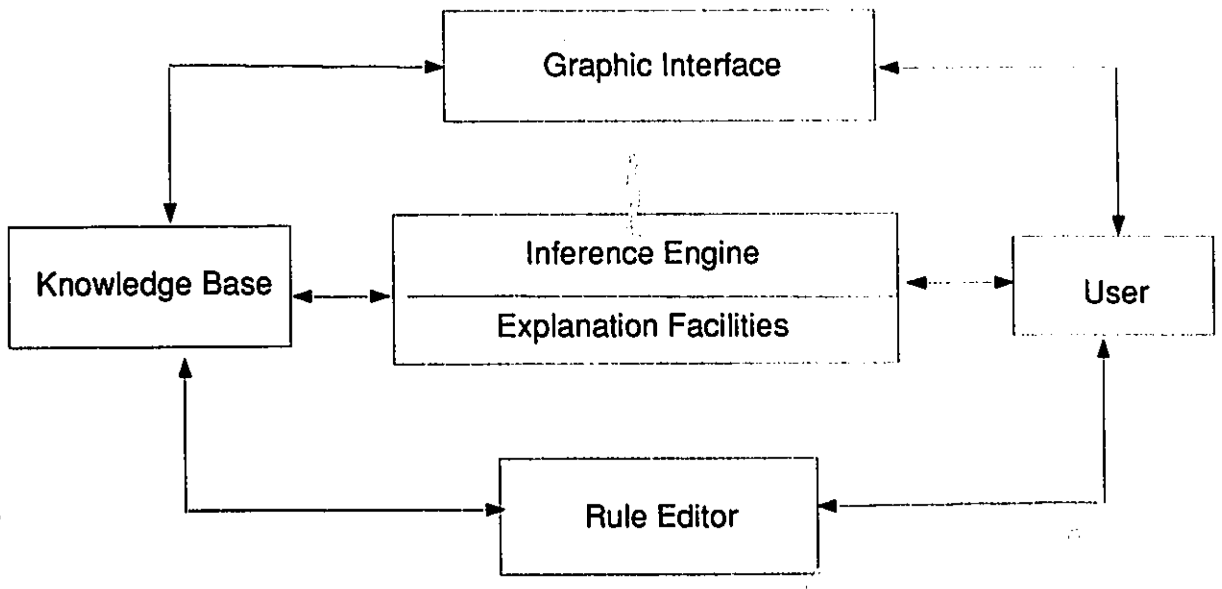
. *Explanation facilities:* This component provides explanations to a user. It explains how solutions were reached and justifies the steps used to reach them. It allows users to ask "Why", "How" and "What" explanations of the inference engine.

. *Rule editor:* This component allows users to add, delete and modify facts or rules.

. *Graphic interface:* This is included as part of the explanation facilities of some expert systems. For example, it might allow users to view an expert demonstration of the operation of a computer-aided design (usually for designing digital logic circuits).

The following diagram shows a hierarchical view of an expert system where *User* can be end users, knowledge engineers or domain experts. Information can be passed from one entity to the other along the lines in the direction of the arrows.

Figure 1 Diagram of Components of Expert Systems



Terminology

Some frequently used terms are:

Goal: this is a specific task, strategy, or fact which needs to be reached or satisfied.

Rule: this is a formal way of specifying a recommendation, directive, or strategy. Rules are expressed as:

- IF <*premise*> THEN <*conclusion*> or
- IF <*condition*> THEN <*action*>.

Backward chaining: this is an inference method where the system starts with what it wants to prove, for example G, and tries to establish the facts it needs to prove G. It seeks to satisfy a stated goal by seeking rules in which the THEN portion matches the goal, then seeking other rules whose THEN portions match the IF portion of the rule which satisfies the goal.

Forward chaining: this is another inference method where the rules are matched against the facts to establish new facts. It seeks to identify all rules whose IF portions are true, then uses the THEN portions of those rules to find other rules which are also true.

Tracing: this facility provides the user with a trace or display of the system operation, usually by listing the names (values of variables used) for all rules fired, or showing the names of all subroutines called.

Slot: this is an attribute associated with a node in a frame system. The node may stand for an object, concept, or an event. For example, a node representing the object *employee* may have a slot for the attribute *name* and one for the attribute *address*. These slots would be filled with the employee's actual name and address.

2.2.3 What is an Explanation Facility

An explanation refers to information that is presented to justify a particular course of reasoning or action.

In expert systems an explanation facility typically refers to a number of techniques that help a user understand what a system is doing and why. Many knowledge systems allow a user to ask "Why", "How", or "What" queries. In each case the system responds by revealing something about its assumptions or its inner reasoning.

2.2.4 Why is an Explanation Facility Useful

The explanation facility is an important component of any expert system. It helps users understand and trust the system and helps decide the best way to exploit the system's capabilities. In many consulting applications, explanation facilities need the ability to justify and explain their advice to the user. This is necessary for the following reasons:

- Because the knowledge base is limited, the user may want to know if the system makes use of all the knowledge that the user considers relevant.

- The user may want to know if the strategies adopted by the system for solving the problem are satisfactory.

. The user may wish to know if all the relevant data describing the problem state are being considered [Hen88].

2.2.5 Components of Explanation Facility

The function of the explanation component is to give the reason for questions asked by the inference component (Why explanation); and to justify the deduced facts (How explanation) or help users to understand a technical term or rule (What explanation).

2.2.5.1 A Why explanation

A Why explanation is an extract from the argumentation chain (that is, in general, the rules used). Basic components of an Why explanation include:

- the current goal
- the current rule under investigation
- the current satisfied premises.

2.2.5.2 A How explanation

The basic components of a How explanation include:

- the knowledge source employed
- The context of acquisition which may consist of supporting facts.

2.2.5.3 A What explanation

A What explanation provides additional information of definition or details of terms and rules.

2.2.6 Example

The following example taken from MYCIN's expert system to illustrate its capability to explain the reasoning to the user [Sho76]. This means that it can describe its reasoning steps; how a request for data is related to a goal; how one goal leads to another; and how a goal is achieved. The example shows a consultation excerpt in which the program demonstrates a line of reasoning (chained goals) that lead backwards from the current question:

Is the patient a compromised host (rule 343) -> is the patient immunosuppressed? (rule 395) —> is the patient receiving steroids? (rule 543) —> is e.colis causing the meningitis?

In the questioning session after the consultation, one can inquire further about the program's intermediate reasoning steps, including why it didn't ask about something. User response is shown in bold.

23) *Is J. Smith a compromised host?*

*****Why**

[i.e. WHY is it important to determine whether J. Smith is a compromised host?]

[3.0] This will aid in determining whether J. Smith is immunosuppressed.

```
If
  [3.1] J. Smith is not a compromised host
```

then

it is definite (1.0) that J. Smith is not immunosuppressed

[RULE343]

[back to question 23...]

*****Why**

[i.e. WHY is it important to determine whether J. Smith is immunosuppressed?]

[4.0]... in order to determine whether J. Smith has received corticosteroids.

If

[4.1] J. Smith is not immunosuppressed

then

it is definite (1.0) that J. Smith has not received

corticosteroids.

[RULE395]

[back to question 23...]

*****Why**

[i.e. WHY is it important to determine whether J. Smith has received corticosteroids?]

[5.0]... in order to determine the organisms causing the infection.

It has already been established that

[5.1] the infection which requires therapy is meningitis, and

[5.2] only circumstantial evidence is available for this case, and

[5.3] the type of meningitis is bacterial.

Therefore,

If

[5.4] J. Smith is receiving corticosteroids

then

there is evidence that the organisms causing the infection

are:

e.coli (.4), klebsiella-pneumoniae (.2), or

pseudomonas-aeruginosa (.1)

[RULE543]

[back to question 23...]

*****No**

Now the user can ask for other explanations:

*****Did you consider whether the patient was receiving steroids?**

RULE543 could have used:

whether J. Smith has received corticosteroids.

Clause 4 of RULE543 {the patient has received corticosteroids} was already known to be false.

*****How did you know that the patient had not received steroids?**

RULE395 was used to conclude that J. Smith has not received corticosteroids. The last question asked before the conclusion was made was 23.

*****What was question 23?**

Question 23 was asked in order to find out whether J. Smith is a compromised host in an effort to execute RULE343.

2.3 Aspects of Explanation Facilities

The explanation facility of expert systems helps increase user confidence in and patience with systems by revealing internal rules leading to system conclusions. Applying system knowledge can help users understand how the system reached its conclusions, help debug, test or monitor the knowledge base and problem-solving behavior and convince users that the system's conclusions are reasonable.

In knowledge-based systems, there are three top-level functions to generate the explanation.

- content of explanations
- user models
- the human-computer interface.

2.3.1 Content of explanations

If the user queries some aspect of the system's decision making, we need to generate an information structure containing elements that make up an explanation.

In constructing this explanation we must consider the current problem solving task. Explanation content can be put together in two ways

–By introspecting: that is, picking appropriate traces for the user query or retrieving knowledge base portions used in making the decision.

–By concocting: that is, producing a justification that does not relate to how the decision was actually made, but that independently makes the decision plausible.

2.3.2 User models

It may not be necessary to communicate all available explanation content to users. By applying user goals, states of knowledge, and the dialog structure, systems filter, shape, and organize process output so that explanations respond to user needs. This often requires user modeling, and tailoring explanations for the user [MWM85].

2.3.3 The human-computer interface

The content of explanation and user modelling functions produce all the information needed conceptually and logically for the required explanation. But how to present it to users is still an issue to be considered. Some considerations are What is an appropriate human-computer interface for effective display? What is the best form of presentation; natural language or in graphical form?

In general, in the process of generating explanations, the content of explanation function is more important than the other two. Poor explanations will be presented if the explanation content is inadequate or inappropriate even if the theories for user modeling and the interface functions are good.

2.4 Analysis of Research

In the following we consider in more detail approaches which are widely employed in the content of explanations, user modelling and user interface.

2.4.1 Content of explanations

2.4.1.1 Introspection — The First Approach

Davis, Shortliffe and the other originators of MYCIN [Sho76] contributed `the first approach` to knowledge systems explanation. Their essential concept was that a trace of the problem-solving activity at the implementation level (for MYCIN, this would be its rule architecture) can give explanations about what the system did. During the execution of the program, MYCIN's explanation facility answered user questions about *how* and *why* certain conclusions were reached by combining a declarative reading of production rules with a simple natural language translation system to generate an augmented trace of the system reasoning. MYCIN expressed its explanation entirely in terms of rules and goals.

2.4.1.2 Introspection — The Second Approach

In the first approach, traces of rule activations may describe program behavior, but they can not justify it, because the justification is part of the knowledge used to design and implement the program. This knowledge is not represented explicitly in the code. The principles of reasoning in the domain are typically confused with the model of the domain that the system reasons with. With his Xplain system [Swa83], Swartout introduced the second approach. He suggested that one way to design an expert consultation program is to specify the domain model and the domain principles, and then invoke an automatic programmer upon this specification to generate the performance program. The process of integrating the prescriptive and descriptive aspects of the specification into the final system is recorded and used to produce explanations of the system's behavior. A knowledge based system has task-specific goals and problem-solving knowledge that we can view as compiled from more general domain knowledge. If the system remembers a trace of the compilation, it can justify system rules in terms of deeper knowledge. Xplain uses deep[2] knowledge (the "domain model" contains facts about the domain of application as causal paths and taxonomies) and a representation of problem-solving control strategies ("domain principles" include

[2] "deep" knowledge refers to multi-levels of abstraction of knowledge. For example, first level of knowledge is the domain knowledge of application. Second level of knowledge is the domain knowledge of the domain knowledge of application.

methods and heuristics, which are usually either hard-coded into the interpreter or given to the interpreter as meta-rules) to compile a knowledge-based system. Therefore, the system can examine the control strategy to analyze system behavior and can use the deep model to justify system rules.

2.4.1.3 Introspection — The Third Approach

The third important approach to explanation can be seen in NEOMYCIN [CL81]. Clancey indicated that knowledge-based systems typically perform tasks best described at a higher level than a rule base's goal-subgoal level. But MYCIN had explicit representation of the rules only, and not of problem-solving strategies that may be encoded implicitly in rule formalisms by system designers; therefore, it could not answer "why" questions that needed to be interpreted strategically. However, Clancey points out that if system behavior is represented at the task level, it can produce explanations at the task level. NEOMYCIN solves the same diagnosis problem as MYCIN, but explicitly represents the diagnostic task (a domain-independent strategy for doing diagnosis). It contains diagnostic operators including "establish hypothesis space" and "explore and refine" that represent the diagnostic strategy and in terms of which it can explain its problem-solving activity, that is, forward reasoning from data, associations which trigger new hypotheses, and a "working memory" of alternative hypotheses that it explores according to a "group and differentiate" strategy. Thus, NEOMYCIN can give strategic explanations describing its higher level goals.

2.4.1.4 Explanation by concocting

Explanations are generated by concocting when problem solvers have no access to records of their own problem solving, or when information contained in those records is unnecessary or incomprehensible to users [CTJ89]. The explanation may argue convincingly that the answer is correct without actually referring to the derivation process, just as mathematical proof persuades without representing the process by which mathematicians derived the theorem.

2.4.2 User models

A model of the user can be used as a step in determining what needs to be explained to a particular user. The ability to adapt to a specific user is an important aspect of explanation.

The intended user will influence the explanation content. Consequently, much explanation research focuses on determining the level and content for a given user [WS89]. For example user modelling helps systems to construct internal representations of user knowledge, goals, and plans. McKeown et al. have implemented user models to *tailor explanations* for specific users [MWM85] — analyzing users to discovers the goals of their system interaction and using these goals to direct responses to queries and thereby providing the most relevant information possible.

Other research concentrates on what information and detail level systems should present to users during explanation. In ONCOCIN [LS83], Langlotz and Shortliffe are able to highlight significant differences between the user's and system's solutions by first asking the user to solve the problem — this is a common approach in Intelligent Tutoring Systems. In BUGGY [BB80], Brown and Burton compile an exhaustive representation of errors in arithmetic to identify a student's addition and subtraction *"bugs"*. Genesereth [Gen82] takes the approach of constructing a user plan in the course of an interaction to determine a user's assumptions about a complex consultation program.

In other user-modelling research user goals are employed to achieve explanations satisfying the implicit intent of user queries [HRWLe83]. Wallis and Shortliffe use numeric markers to distinguish *"complexity levels"* and the importance of rules [WS82]. Explanation levels vary according to how many problem-solving steps systems explicitly list for users. In the other words, based on detail and user expertise, the system omits highly complex and minimally important steps (those with high complexity ratings and low importance ratings) from explanations.

2.4.3 The human-computer interface

Questions to be asked are: Can the system and its users employ queries in an interactive discussion or should a query be provided as a simple probe? How can an appropriate human-computer interface effectively display and present information to users? These aspects significantly influence the generated explanation's form and content; interactive explanations are more difficult to produce than non-interactive explanations [WS89]. It is difficult to decide which explanations are best presented in natural language and which in graphical form.

KEE [Tek85], Intellicorp's knowledge engineering environment, provides many features including frame inheritance, advanced graphical representations, menu-driven commands, and several programming paradigms. It also provides an explanation facility based on the Why/How query. KEE uses a backward-chaining explanation window to display answers regarding 'why' and 'how' queries in network form and allows users to implement their own explanation facility.

2.5 Future Trends

The contribution of expert system researchers has been to place a high priority on the accountability of consultation programs, and to show how explanations of program behavior can be systematically related to the chains of reasoning used by rule-based systems. These contributions include attempts to separate out the different kinds of structural and strategic knowledge implicit in expert performance [Cla83], and attempts to make explicit and accessible the design purposes of generating better explanations and automatic programming [Swa83]. That is, different ways of structuring knowledge system for different strategical purposes. The effect of the structural knowledge is to provide a handle for separating out what the knowledge system is from when it is to be applied, and whether or not the strategy for invoking the knowledge system is explicit or encoded indirectly.

A reasoning system must contain, and be able to explain not only knowledge about a domain, but also knowledge about knowledge about a domain (meta-level knowledge) [Dav80]. In NEOMYCIN, the strategies are made abstract by making metarules and tasks domain-independent. It is possible to direct a consultation using this general problem-solving approach so that resulting explanations of strategy are able to convey this strategy. In future, discourse rules will be developed for determining a reasonable level of detail for a given user.

Another important area is constructing a user interface that is able to generate natural language explanations in response to user questions and more specifically to their needs.

2.6 Conclusion

In the artificial intelligent area, explanation is a complex topic. We have introduced the main concerns and activities of researchers who are interested in both developing a methodology for building expert systems and improving the quality of explanation they produce.

Expert systems must be able to explain what they do and why, but traditional approaches fail to provide adequate explanations and justifications.

Current systems have limited explanatory capabilities and present maintenance problems because of a failure to explicitly represent the knowledge and reasoning that went into their design.

Chapter 3 USER MODELLING IN EXPLANATION FACILITIES IN EXPERT SYSTEMS

3.1 Introduction

Within an expert system, it may not be necessary to communicate all available explanation contents to users. Since the knowledge base contains many different objects with regard to different areas of the knowledge, most users will be neither fully naive nor truly expert with respect to the knowledge base. *"Naive"* and *"expert"* are the extremes of a knowledge spectrum and most users will be at intermediate points in the spectrum. Such users may have *local expertise* about some objects in the knowledge base and not others. Hence, it is not sufficient for the system to indicate whether a user is naive or expert. Systems should apply user goals, states of knowledge, and the dialogue structure to filter, shape, and organize process output so that explanations respond to user needs. This often requires user modeling, and tailoring explanations for the user [MWM85]. User modeling, techniques for building user models and review of user models in current systems are presented in following sections.

3.2 Frameworks for User Modelling

In this section we will define some terms which are used throughout this thesis to understand what is an user model and how it is classified.

An ***expert*** user is one that is knowledgeable about the domain of the knowledge base, generally spending a lot of time using the system.

A ***naive*** user is not knowledgeable about the domain of the knowledge base, generally using the computer and/or package infrequently.

The requirements of these two classes of user differ substantially. The expert is concerned with speed and flexibility of input, direct control of the operation of each module of the system and access to all the resultant output in a structured and flexible manner. The naive user wants a clear and coherent interface where the system provides guidance as to the current options and their implications, error trapping and easy error recovery, and a concise summary of the results in an easily understood manner.

A ***stereotype*** consists of a set of facts and rules that are believed to apply to a class of users. Stereotyping is a major technique which researchers use to build models of users.

An ***individual*** has exactly one stereotype with which it is currently associated and a collection of definite ground facts which are true.

A ***user modelling component*** is the part of a dialog system whose function is to incrementally construct a user model, store, update, delete entries, maintain the consistency of the model, and supply other components of the system with assumptions about the user.

The term "*user model*" can be used to describe a wide variety of knowledge about a user, either explicitly or implicitly encoded, which is used by the system to improve the level of man-machine interaction. In "Learning and Teaching with Computer" O'Shea and Self [OS83] gave a very good broad definition of a user model, which encompasses a wide variety of modelling schemes:

> *"Any information which a program has which is specific to the particular student being taught. The information itself could range from a simple count of how many incorrect answers have been given, to some complicated data structure which purports to represent a relevant part of the student's knowledge of the subject."*

The issue of user modelling has been raised in many areas of artificial intelligence, from man-machine interfaces to expert systems and intelligent tutoring systems [Sle85]. This has led to a proliferation of different approaches and techniques. Several classification schemes have been proposed, which can be employed in order to provide an analytical understanding of user modelling. These schemes are helpful and are used to clarify how user modelling can be taken into consideration in the design of expert systems.

In order to provide a more analytical insight into the nature of user models, we will first examine a classification of them, then describe different techniques for inferring users knowledge of individual concepts from their observed behaviors to build user models. Several user-modelling systems will also be briefly reviewed.

3.2.1 User modelling classification

In the following, we will survey two major approaches, one proposed by Carbonell [Car83] and one proposed by Rich [Ric83].

3.2.1.1 Carbonell's approach

The scheme proposed by Carbonell identifies two categories: empirical quantitative models and analytical cognitive models.

Empirical quantitative models:

Empirical quantitative models entail information derived from an abstract formalization of general classes of users. The model is defined through parameters compiled from empirical data, encoding quantitative relations between primitive operations carried out by the user during the interaction with the system in solving a specific task, and a measurement of the performance shown by the user in solving such a task. Classical examples of this class are the keystroke model developed by Card, Moran, and Newell [CMN83] and the ZOG system developed by Robertson, Newell and Ramakrishna [RNR81]. ZOG is a frame-based system that facilitates user-computer communication. Its design was influenced by such factors as the response speed necessary to prevent user frustration. These models contain only surface knowledge about the user (or a specific class of users), and no internal reasoning takes place. Therefore, the resulting system does not contain any separate knowledge base devoted to represent user modelling information.

Analytical cognitive models:

Analytical cognitive models are aimed at simulating aspects of user cognitive processes taking place during interaction with the system. These models are based on explicit representation of user knowledge, of a qualitative nature. Implementation utilizes artificial intelligence techniques. The consideration of a knowledge base devoted to store user modelling information allows the specific traits of each single user in a given class to be followed. For example, UMFE [Sle85] is a front end system for the explanation component of expert systems. It determines the user's level of sophistication by asking as few questions as possible, and then presents a response in terms of concepts which UMFE believes the user understands. It uses a series of implementor-defined inference rules to decide the concepts which the user is likely to know or may not know, given the concepts the user indicated he or she knew in earlier questioning.

3.2.1.2 Rich's approach

Rich [Ric83] gives three *"dimensions"* for classifying user models:

- ☐ A model of a single stereotype-user versus a collection of models of individual users.

☐ Models specified by the user or systems designer versus models inferred by the system based on the user's behavior.

☐ Models of long-term user characteristics versus models of the current task.

Canonical vs. individual model The first dimension is one single model of a typical user versus a collection of many individual models. The first category proposed by Carbonell [Car83] conforms to the canonical model approach, whereas any system that has to be capable to tailor its behavior to a heterogeneous variety of users has to conform to the individual model paradigm.

Canonical models characterize an abstract *"typical"* user, while individual ones have potential to accommodate each user's particular needs. A variety of computer systems have been designed around a canonical user model. For example, ZOG [RNR81] as previously described. Another example of a system built around a model of a canonical user is Genersereth's automated consultant for MACSYMA, a symbolic mathematical package [Gen78]. The consultant exploits an explicit model of the problem solving strategy used by MACSYMA users. But there is a limit to the usefulness of the canonical models to a system with a heterogeneous user community.

Individual models can enable such systems to provide each user an interface more appropriate to his needs than could be provided using a canonical model. There exist techniques for implementing such models so that they actually do improve the performance of the system. The decision to exploit individual user models has a profound impact on the other aspects of user modelling. If a system possesses only a single model of a canonical user, that model can be designed once and then directly incorporated into the overall system structure. If, on the other hand, the system is ultimately to possess a large array of models corresponding to each of its users, the question of how and by whom those models are to be constructed arises and this leads to the second dimension of user models.

Explicit vs. implicit There are two ways to make systems different for different users. Explicit models are based on information which is provided by the user to describe himself/herself or his/her needs (either under user control or system control), whereas implicit ones are constructed by observing his/her behavior (the models are built up by the system).

From the point of view of an increase of the level of user interaction and of the usability of the system, the latter approach is much more desirable. The model can be built without user intervention, and the knowledge included in the model, though uncertain because it results from an inference process, is generally more reliable than that directly provided by the user, who is usually a bad source of information about himself/herself [NW77].

In the explicit model, the user is asked to rate his/her level of expertise with the system, then uses that level to determine how much information to provide in explanation or error messages. The system essentially contains models of how much information people at each level already possess.

Beside the lack of accuracy inherent in explicit models, there is yet another consideration for allowing the system to build its user models itself. Users do not want to stop and answer a large number of questions before they can get on with whatever they are trying to use the system to do. This is particularly true of people who intend to use the system only a few times, and for only brief periods. In this case, the system should form an initial model and let the user immediately begin to use the system. This initial model can be based on the known characteristics of the system's overall user community. As the person interacts with the system, he/she provides it with additional information about himself/herself. As it acquires this information, the system can gradually update its model of the user until eventually it comes to be a model of that individual as distinct from the canonical user. Using this approach, the greatest effort will be expended on the construction of models of frequent users, while much less effort will be expended on models of infrequent users.

Long-term vs. short-term models The last dimension for considering the classification of user models is short-term versus long-term modelling. The former focuses on information that changes in the short term (e.g., during a single session) and the latter on characteristics that changes more slowly, possibly over a long period of time (e.g., a whole series of sessions). This last dimension is useful for pointing out some of the differences between user modelling techniques adopted in intelligent tutoring systems and those utilized in man-machine interfaces. In fact, the rapid change of user characteristics during a session is a distinguishing feature of tutoring systems, which indeed have the goal of changing (possibly in the short term) the knowledge level of the student.

In order to interact reasonably with a user, a system must have access to a wide variety of information ranging in time from relatively long-term facts, like his/her level of mathematical sophistication, to quite short-term facts, like the subject of last sentence the user typed. Although all of this information can contribute to the habitability of a system, it is useful, at least at the beginning of an exploration into the topic of user modelling, to separate the problem of inferring long-term from that of inferring short-term model because different techniques may be appropriate for the solution of the two problems.

Short-term modelling is important in understanding natural language dialogue. Long-term models have more available evidence to base inferences on than short-term ones and so may appear to be more accurate. However, this is confounded by the fact that what is being modelled is continually changing. But many systems could usually exploit a large amount of much more stable knowledge about their users. These long-term models can be derived over the course of a series of interactions between the system and its users. The model can contain such information as the user's level of expertise with computer systems in general, his/her expertise with this system in particular, and his/her familiarity with the system's underlying task domain. In addition to these general things that could be of use in a wide variety of systems, the user models employed by a particular system will often need to contain specific information relevant to the system and its task domain.

3.3 Techniques for Building User Models

Some specific techniques that can be used for user modelling can be represented as follows

1. elicitation mode, that is, the way information is organized at acquisition time.
2. acquisition procedure, that is, the way information is actually collected or produced.

According to the first criterion, two basic modes of information elicitation can be mentioned:

- Acquisition of a single information item at a time, which represents a specific fact about the user currently interacting with the system.
- Acquisition of a cluster of information items in one shot, which represents a collection of facts about the current user.

According to the second criterion, three main procedures for information acquisition can be identified:

- Observation of free dialogues between the user and the system, and observation of the answers provided by the user to specific direct questions posed by the system.
- Inference from observed facts encompasses two steps:
  a. acquisition through observation of facts that are not appropriate to be directly inserted in the user model but can be used as raw data from which some useful information items can be derived.
  b. inference from the observed facts of the appropriate information items to be inserted in the model.
- Inference from known facts is the expansion and refining of the model without using new information items acquired from the user-system dialogue.

These techniques fall into two broad groups: methods for inferring single facts at a time and methods for inferring whole clusters of facts at once.

3.3.1 Inferring individual facts

One of the simple ways to derive information about a user is to look at the way he/she uses the system. Any user who begins a session with a series of advanced commands is probably an expert. Any user whose first few attempts to form commands are rejected by the system is probably a novice and needs some help. One way to implement user modelling based on this sort of information is to construct a dictionary of system commands, options, and so forth, and to associate with each item an indication of what information the use of that item provides about its user.

Another way that users provide information about themselves is via the pattern of their commands. Consider the situation when a user asks a system for a particular piece of information. If he/she gets what he/she wants, he/she will either leave or continue with his/her next request. But if he/she does not satisfy, he/she is likely to try to restructure his/her request in another attempt to get what he/she wanted. This indicate to the system that it does not satisfy the user's need with its first response.

The model of the user can also be modified by observing the patterns of the user's questions. If the system misjudges the user's level and answers his/her first question by referring to a system parameter that means nothing to the user, the user's next question will refer to that parameter to figure out what it means. In this case, the system can conclude that its model is wrong and then modifies it until the user is satisfied with the level of explanation. Similarly, if the system underestimates the user's knowledge, he/she will ask for more specific information, and the system can then update its model.

3.3.2 Using stereotypes to infer many things at a time

The techniques that have been discussed so far enable a system to infer individual facts about a user. But if a user model is to be very complex, the question of how to collect all the required information within a reasonable period of time arises. If a user will have only a few interchanges with a system, then user modelling that requires many interactions to build an initial model will be of little use. However, in many situations it is possible to observe one or a small number of facts and from them to infer, with a fair degree of accuracy, a set of additional facts.

Stereotypes are useful for doing the kind of reasoning about people required to build user models. In particular, they provide a way of forming plausible inferences about yet unseen things on the basis of the things that have been observed.

A stereotype represents a collection of traits. It can be represented as a collection of attribute-value pairs. Each such attribute will be called a *facet*. A model of an individual user can also be represented as set of facets filled with values. The facets of the stereotypes used by a system should correspond to the facets of the user models built by the system.

Some traits may be easily observable. They serve as *triggers* that cause the activation of the entire stereotype. Since the presence of a trait may only be suggestive of a particular stereotype rather than absolute evidence for it, each trigger has associated with it a rating that is a rough measure of the probability that the stereotype is appropriate given that the trigger was observed.

Stereotypes represent structure among traits. There is often additional structure that can be captured by representing a collection of stereotypes as a hierarchy. Information in very general stereotypes can be used unless conflicting information is suggested by more specific stereotypes. The most general stereotype available to a system can represent a model of a canonical user. Thus even without much information, a system built on stereotypes will do no worse than one built on the traditional built-in model of a canonical user.

3.4 Review of Current Systems

In user modelling, the representation of user modelling is usually determined by the application purpose of the system. In this section we will review how user models are represented and the different types of models inferred.

3.4.1 Representing User Models

User models are often represented in one of three forms: a parametric form in which a small set of values is identified to characterize the user for a particular task; a discrete-event form in which the command or keystroke sequence is massaged into a finite-state or finite-context model; or a frame-like form in which domain knowledge is used to identify explicitly his performance with each concept.

An example of a parametric model is provided by Rowe [Row84]. Observing that databases are often used repeatedly for very similar tasks, he identified a theoretical model of these tasks which enabled his system to infer parameters for the model based implicitly on the user's recent queries to the database. The parameters are incorporated into the task model to predict which database items are of most interest. Many computer-aided instruction programs represent user models in parametrized terms. For example, branching systems developed from Skinnerian stimulus-response principles present course material and then quiz the student. The history of correct and incorrect responses drives the selection of subsequent instructional material. In these systems the courseware author must anticipate every wrong response, and prespecify branches to appropriate remedial material based on his/her ideas about what the underlying misconception might be [BF82].

Discrete-event models arise out of a system-theoretic approach to behavior-to-structure transformation. The commands invoked by the user are treated as an abstract sequence to be modelled in a way which supports analysis of the behavior and prediction of future inputs. The "reactive keyboard" [WCD82], which forms an implicit model of a user's keyboard input, is an example of this technique. It intercepts the keystrokes made by a physically handicapped user, storing each one along with its context, and expedites text entry by predicting sequences of characters which will be typed next.

Frame-like models are based on expert-system knowledge representation techniques and use ad hoc structures to store information relevant to selected facets of a user's performance. These are updated by rules which rely on knowledge of the task domain. Explicitly representing a student's perceived understanding of the domain allows flexible and modular construction of assistance programs. For example, it allows separation of teaching strategy from the subject expertise to be taught.

Parametric models are useful for well-defined tasks, but lack the flexibility required to support computer tutoring. Discrete-event models pose difficulties of communication with the user, since they work in an alien and very low-level language of sequences of commands and abstract systemic models of them. Frame-like models are imprecise since it is not possible in general to infer users' knowledge accurately from their behavior in complex domains. But they offer a great deal of structure and flexibility and, since they can provide individual estimations of a user's knowledge of each concept in the domain.

3.4.2 Scalar Models

In scalar models, the system asks an user to select his/her level of sophistication which represented by a scalar value; based on that value, the expert system will give explanations of a causal reasoning chain to the user or prompt him/her with error messages and possible suggestions.

The earliest adaptive CAI programs often used this method to represent the student's level of sophistication by a scalar value. Woods and Hartley [WH71] report that the first version of the Leeds arithmetic system kept a single number to represent the overall complexity of the tasks which the individual student was able to tackle. Loosely, this model asserts that the number of keystrokes determines how productive a user can be at a not very demanding task, such as searching for information or performing a well defined text editing procedure. Wallis and Shortliffe [WS82] assumed that the user can be represented by an integer difficulty level. The purpose of their system is to explain a causal reasoning chain to the user of an expert system; the concepts used in the explanation are essentially determined by the user- selected difficulty level.

The scalar models are simple for implementing, but are not flexible and are imprecise in reflecting the true level of users difficulty in case they misjudged that level themselves.

3.4.3 Ad Hoc Modelling Systems

In ad hoc modelling systems, different models are built based on different applications, these models like frame-like models which are based on expert-system knowledge representation techniques and use ad hoc structures to store information relevant to selected facets of a user's performance. Their characteristics can be referred to frame-like models' characteristics described above.

Hartley et al. [AE72] reported a system which captured many of the aspects of medical diagnosis. The student doctor was presented with a "patient" which consisted of a pattern of Zs, Os and *s and was asked to classify the pattern as one of a limited number of types. On request the system would generate characteristic examples of each "disease". The system thus had information about the task seen by the student and his or her response. Additionally, this particular system executed a series of models on the task given the student, and compared these results with the student's. Two sets of models were used, one based on stereotype matching and the other on a probabilistic model (for each set of models, one model corresponded to unweighted attributes, the second to a slight weighting and the third to an increased weighting; thus the unweighted probabilistic model corresponded to a (pure) Bayesian model).

3.4.4 Profile Models

In GRUNDY [Ric79], users were characterized in terms of a series of weighted attributes, such as romantic, suspense loving etc. As GRUNDY's task was to select an appropriate book for the user, Rich had similarly characterized each book; GRUNDY then proceeded to do a match between the stylized characteristics of the user and the books. To do this, it exploits two collections of data:

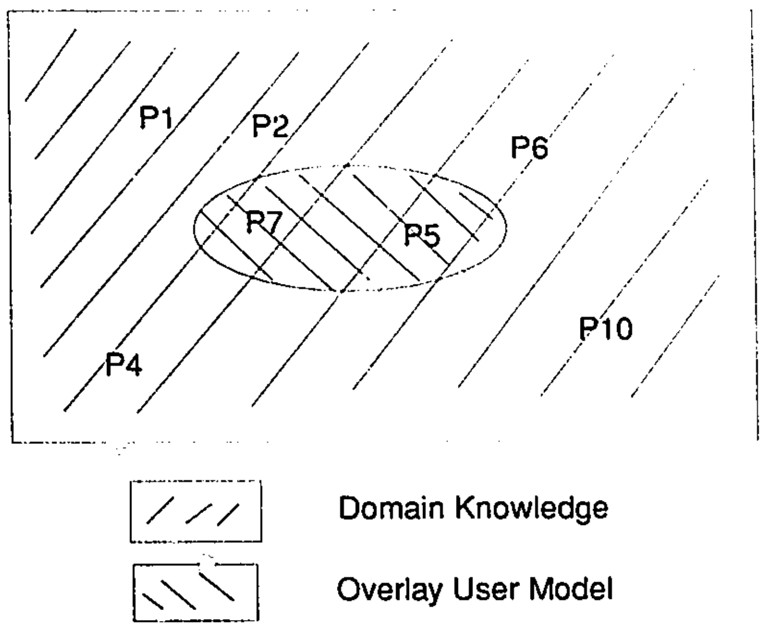
- descriptions of individual books. Each description is a set of facets filled with appropriate values.
- stereotypes that contain facets that relate to people's taste in books. Associated with each stereotype is a collection of triggers.

GRUNDY exploits user modelling techniques more extensively. Its main objective is to tailor interaction to the individual user. The system utilizes a hierarchy of frames comprised of several slots, called facets. At the beginning of the interaction, GRUNDY prompts the user with some specific questions and, from the answers obtained, it collects the information needed to select the stereotypes appropriate for that user. These will constitute the initial user model. The model contains long-term knowledge preserved from session to session, which includes the user's background and characteristics, and a record of past interactions with the system. An (implicit) refinement of the model is carried out by GRUNDY when the user expresses his/her judgment on the quality of the results produced so far by the system: the refinement is performed by changing (increasing or lowering) confidence factors attached to the facets of the stereotypes belonging to the current user model; it is supported by information gathered through a bounded scope dialogue driven by the system.

3.4.5 Overlay Models

In overlay model, the knowledge of the domain is represented as a network and each node has a numerical value associated with it indicating the likelihood that a particular user knows the knowledge associated with the nodes. An overlay model is illustrated below.

Figure 2 An overlay model



In SCHOLAR, Carbonell [Car70] represented knowledge of his task domain, the geography of South America, as a hierarchical semantic network. Nodes which corresponded to topics the student user knew/did not know were appropriately marked.

The overlay model, proposed by Carr and Goldstein [CG77], is the simplest of the modelling techniques. Overlay modelling assumes that all differences between the student's behavior and the behavior of the expert model can be explained by the lack of a skill or skills on the part of the student. Thus the knowledge of the student is viewed as a subset of the expert's knowledge. Since overlay models do not require information beyond the contents of the expert model, they are relatively easy to implement.

Overlay modelling has serious drawbacks. An overlay model has no provision for dealing with knowledge or beliefs of the student that differ from the expert's.

The significant difference between an overlay model and a profile model is that the former uses topics which the user is intending to acquire, as opposed to the more general descriptors used with profile models.

3.4.6 Process Models

All these systems assumed that the user's knowledge was merely a subset of the expert's. Recent studies in Cognitive Science have shown this is frequently *not* a valid assumption, and so models which allow both the correct and incorrect knowledge to intermingle have been introduced.

Self [Sel74] was the first person to suggest explicitly that a student-user model should be executable, so that models could be used predictively, and in some sense capture the processes by which the user solves the task. The BUGGY project [BB78], was the first significant instance of the idea of a process model. Further, the concept of a process model was extended significantly to include incorrect or buggy sub-procedures so that the BUGGY system was able to replicate the performance of students with (consistent) errors.

Sleeman [Sle82] used a similar technique to capture the errors made by algebra students. PIXIE [Sle87] is an intelligent tutoring system for diagnosing and remedying student's errors in a particular domain, for example early linear algebra.

An important aspect of these latter models is that they are *process* models- and so can be executed by an appropriate interpreter- thus enabling them to be used predictively. Both BUGGY [BB78] and PIXIE address the issue of *inferring* models by observing the student's performance on a series of tasks. Technical issues addressed by these systems include how to make the search computationally tractable, and how to overcome spurious responses.

The essential difference between the *ad hoc* modelling systems and the process models is that the former are highly dependent on specialized inference engines to interpret their user models. In the process model, the user model in its own right is suggestive about how the user would perform the task, and these models can be executed by a more standardized inference engine.

3.5 Future Work

Research on user modelling has been very active in the last few years. A number of interesting proposals have been made concerning representation systems for user models, methods for generating assumptions about the user from his/her natural-language input, and strategies for the planning of the system's dialog contributions on the basis of its model of the user. Several prototype systems have been implemented in order to exemplify and further investigate these proposals. However, the development of flexible user modelling components is rather costly and user modelling tasks will require a much richer knowledge base, a series of representational schema to capture the user's knowledge, and an extensive range of inference techniques. Future modelling systems should use their models predictively and report to the investigator situations where the models persistently fail to predict subsequent user behavior. Alternatively, future systems might use additional knowledge to resolve the conflict.

3.6 Conclusion

A user model is simply the model a system keeps of the individual currently using it. User models are generally needed to aid in the interaction between the system and a user: to help in understanding what the user wants the system to do, or to enable the system to provide its responses to the particular user and his needs. Thus user modelling has concentrated on modelling the user's goals, plans, and beliefs.

Many interactive systems have a strong need to maintain models of individual users. The stereotype can contain definite facts and define rules of inference as well as default information and rules. The rules can be used to derive new information, both definite and assumed, from the currently believed information about the user.

It is important that a user modeler not rely too heavily on answers to specific questions in building models of individuals. The most obvious way to build a user model without asking many questions is to make direct inferences from a user's behavior to a model of him/her.

Chapter 4 IMPLEMENTATION

4.1 Introduction

In this chapter, we describe the implementation used to test our ideas for improving explanations in expert systems. The platform used is the ***Designer's Assistant*** system [Sar 90] which is a knowledge base system for designing high speed digital signal processing applications. To this system we have added a user modelling component and a text retrieval component. The idea is to use the information retrieval component to select relevant passages of text for an explanation and to use the user modelling component to decide what should be retrieved and how it should be presented.

We organize this chapter as follows: section 4.2 provides a brief description of the ***Designer's Assistant*** system and the reasons for choosing it as an experimental platform. In section 4.3 we discuss in detail how to build and maintain the user model. In section 4.4 we discuss how keywords are attached to rules and used with simple information retrieval techniques to retrieve relevant text as a basis for an explanation. Section 4.5 describes the user interface. It includes the specification of options which a user can have during the design process. Finally, section 4.6 contains the conclusions of the chapter.

4.2 Implementation Context

A knowledge base system for high speed digital signal processing application, in particular the design of digital FIR filters using a special purpose systolic architecture has been developed [Sar 90]. The knowledge base is just one component of the ***Designer's Assistant***, an expert system, which is part of a systolic compiler system. The main aim of the ***systolic compiler*** project was to develop a system capable of generating a VLSI chip layout from user inputs. We use the ***Designer's Assistant*** as our experimental platform. A brief description of what the system is and why it is chosen is given below.

4.2.1 What is the Designer's Assistant?

The Designer's Assistant is a knowledge base system which allows users to interactively design digital signal processing systems. It consists of five main components:

1. The knowledge base,
2. The inference engine,
3. The explanation facilities,
4. The user interface, and
5. The abort facility.

The Designer's Assistant is based on an expert system shell developed using Quintus Prolog. It allows interfaces to subroutine libraries in standard programming languages such as C or Fortran. IF-THEN rules are employed to store knowledge. The inference engine, whose inferencing scheme is similar to that of Prolog, is capable of making inferences using the rules in the knowledge base. The explanation facility component enables the system to explain its reasoning. The user interface makes use of the windowing facilities of the Prowindows system to provide the user with the window-based input/output. The user is led through the design process by answering a number of queries posed by the system. Finally, the abort/ restart facility gives the user freedom to discontinue a session in the middle of design process.

4.2.2 Why Choose the Designer's Assistant as an Experimental Platform?

The Designer's Assistant is chosen as our experimental platform because we need an expert system environment which can accommodate different levels of users; besides the system is conveniently available. Furthermore, the Designer's Assistant is based on the declarative programming paradigm that makes the program more readable and easy to modify. The other advantage of the system is that its user interface component provides a flexible and friendly environment to both expert and novice users.

4.3 User Modelling

4.3.1 The Ideal User Model

There are various different types of information about users which the ideal user model might contain. These include knowledge about a user's level of competence with a system including a history log of previous interactions, knowledge about the user's level of domain expertise, knowledge about the user's interests, values, aptitudes, goals, expectations and assumptions, knowledge about the preferred method of interaction and finally knowledge about the user's model of how the system works. The development of flexible and real user models is very complicated and costly. Building a full user modelling system is beyond the scope of this thesis. In this thesis we developed the "models" for two users — one expert and one naive, really a simulation of what the system might build if we had the resources available.

4.3.2 The Design's Assistant User Model

In the context of the Design's Assistant, we start off with two stereotypes but evolve towards individual user models. Facets include knowledge domain, frequency of system use, quantity of information, quality of explanation, goal, user interface preferred. Expert users require the flexibility of input, speed, and direct control of the process. Naive users usually require a clear and coherent interface, system-lead user, error trapping and easy error recovery, and clear presentation of results.

A user model is established for each user based on explicit information obtained from the user and from the observed behavior of the user. With the established user model, a list of keywords is selected. These keywords form an input for the information retrieval component to retrieve relevant texts which are used as explanations. In this way, explanations generated are more likely to be suitable to the user's needs. This is an improvement over traditional explanation facilities. In particular, for the Designer's Assistant this use of user modelling techniques greatly improves its performance because it is easier to use and leads to quicker designs.

4.3.3 How do we build the model

User modelling component was developed using Quintus Prolog [Sys88a] and Quintus Prowindows [Sys88b]. To build the user model, we start with two stereotypes of users: expert and naive uses. For a frequent user, the system explicitly develops an individual model by restoring his/her previous information from the database. For an infrequent user, it explicitly asks the user to answer basic questions in order to form his/her initial model. These questions are based on five categories as explained below:

1. ***Level of expertise.*** The user is asked to rank his/her level of expertise on a scale from 1 to 5 for which 1 is the least expert and 5 is the most.

2. ***Frequency of use of system.*** The user is asked whether he/she uses the system only *once, daily, monthly,* or *sometimes.* If the user only uses the system once, all his/her information will be discarded after the session is finished. For the daily user, his/her model is more likely to stay the same. The situation is different for the monthly and sometimes users. For the first type of these users, the system restores his/her information of the model which can be changed implicitly, while for the later type the system asks whether he/she wants to store this model into the database or not.

3. ***Quantity of information.*** The user is asked to rank his/her preferred quantity of information and explanations from short to long on a scale from 1 to 5.

4. ***Quality of explanation.*** The user is asked how he likes the explanations presented from a technical point of view. A scale from 1 to 5 is given which runs from the least to the most technical.

5. ***Goal.*** The user is asked whether his/her goal is either of a student or of an expert.

Beside explicitly asking questions, the system can implicitly change the parameters assigned to each user at initial stage. The changes are made based on the number of times that the user requires explanations, the user's wrong input values, or information that he/she wants to change when having that option.

After this process a user model is established. Whenever the use requires explanations from the system a list of keywords is generated based on his established user model. This list is then sent to the information retrieval component to retrieve a full text (contained keywords) which is considered as a content of explanations. Hence, the content of explanations varies depending on given keywords. To illustrate the above point we will consider the default cases. If the user is a naive one then his level of expertise is the lowest and the quality of explanation is the least technical. However, if the user is an expert then his level of expertise is the highest and the quality of explanation given is the most technical. The list of keywords selected is based on these criteria. In general, keywords are generated differently at a particular rule according to different levels of expertise or different quality of explanations requested. We also note that the quantity of explanations has only an effect on the size of the content of explanations. Changing the quantity of explanations results in a display of part or full text documentation from the information retrieval component.

4.3.4 How the model is maintained

Each user is assumed to have a unique name. After each session, if the process terminates successfully all updated information about the user is stored into a file. Otherwise the user has the option of saving information and it may or may not be used at a later date. If the user saves his/her model's information, his/her own model will be retrieved when he/she uses the system; otherwise the system will start a fresh from the beginning to built his/her model.

In database, a user model contains the information as follows:

User name.

Explanation or not

Type of explanation.

Frequent use of the system.

Goal.

Level of expertise.

Quality of explanation.

Quantity of information.

The data structure for a user model consists of eight different fields. Each field is described as below:

1. **Name:** Character field contains unique user's name used as Primary Key.
2. **Explanation:** Character field contains the value *yes* or *no* showing whether the user wants explanations or not.
3. **Type:** Character field with the value *designer* or *end-user* indicating what type of explanation is provided.
4. **Frequency:** Character field with the value *once*, *daily*, *monthly*, or *sometimes*. This is used to update user models.

5. **Goal:** Character field indicating user's goal with the value *student* or *expert.*
6. **Expertise:** Numerical field containing the value from 1 to 5 to indicate the level of expertise of user.
7. **Quality:** Numerical field with the value from 1 to 5 indicating how technical the explanation provided should be.
8. **Quantity:** Numerical field containing a value from 1 to 5. This is used to decide how much information the user wants.

The following example shows how users' information is stored in the database.

```
user_list('.'('Michael Smith','.'('Jenny Price','.'  ('John Norrie','.'(
'Philip Poon','.'('Carmen Lee','.'  ('Bill Yuille','.'('Jim Farrell','.'(
'Steve Owens,'.'  ('Dave White',[])))))))))).
user_data('Michael Smith',yes,end_user,daily,student,1,3,3).
user_data('Jenny Price',yes,end_user,daily,student,1,3,5).
user_data('John Norrie',yes,designer,daily,expert,5,5,5).
user_data('Philip Poon',no,designer,monthly,student,3,3,3).
user_data('Carmen Lee',yes,end_user,daily,student,1,1,1).
user_data('Bill Yuille',no,designer,daily,expert,3,3,5).
user_data('Jim Farrell',yes,end_user,monthly,expert,5,3,5).
user_data('Steve Owens',no,end_user,mothly,expert,1,2,3).
user_data('Dave White',yes,designer,daily,student,4,4,3).
```

In this example, consider the user "Bill Yuille". He is an expert, uses the system every day; his level of expertise is 3, quality of explanation is 3 and quantity of information is 5. At the moment, he doesn't want explanations from the system. But he wants the system to store his information which may be used later on.

4.4 Information Retrieval Techniques

In this section we discuss how information retrieval techniques are used to construct explanations.

4.4.1 How Keywords are Acquired and Used

The domain knowledge, FIR filters in this case, is represented as a set of IF-THEN rules. We have attached a list of keywords to each rule. To retrieve relevant information for a particular user, the user model is used to select a set of keywords. In the text retrieval component, simple information retrieval search techniques are employed to search for a textual explanation relevant to what is going on in the design process. We note that the user doesn't have to be an expert in the domain area or on information retrieval search techniques.

Each rule has keywords attached to it. The order runs from the least to the most technical. The list of keywords attached to each rule can be illustrated as below:

Rule1:

1–1kw1 1–1kw2 ... 1–1kw5

1–2kw1 1–2kw2 ... 1–2kw5

...

...

1–5kw1 1–5kw2 ... 1–5kw5

Rule2:

2–1kw1 2–1kw2 ... 2–1kw5

2–2kw1 2–2kw2 ... 2–2kw5

...

...

2–5kw1 2–5kw2 ... 2–5kw5

Keywords are selected depending on the level of expertise and the quality of explanation required by the users. For example, if the level of expertise is 2 and the quality of explanation is 3 then keyword ***2–2kw3*** is selected for Rule 2. The number of keywords generated depends on the number of rules involved in the process.

At each stage, the user should be able to do a number of things as follows

☐ Replace keyword with a broader, narrower, more or less technical keyword

☐ Assign weights to keywords

☐ Add keywords

☐ Delete keywords

☐ Be able to ask the system to modify provided explanations for more technical or less technical, longer explanation or shorter explanation.

4.4.2 Types of IR Techniques Used

Depending on the quality of explanation and the level of expertise, option 1 or option 2 can be used. In option 1, the selected keywords as described above are used as the input for information retrieval component to retrieve the relevant text. In option 2, the selected keywords are used with a thesaurus in the information retrieval component to generated the relevant text. Option 2 is used under following circumstance only:

1. When the users are *truly* novices, that is at the lowest level of expertise and the least quality of explanation, and require less technical explanation.
2. When the users are *truly* experts, that is at the highest level of expertise and the most quality of explanation, and require more technical explanation.

4.5 User Interface

The user interface in the expert system is performed through a window-based input/output which makes use of the windowing facilities of the Prowindows system [Sys88b]. A mouse-and-menu style is employed for the user's choices and the choices are displayed in the form of a set of options.

At any stage the user has the option to change any of his information. Furthermore, the system asks the user whether he/she wants explanations to be provided or not. If the *"no"* is selected then the system provides no explanations. While the choice of *"yes"* leads to two different types for explanations, one for the end user and one for the design user. When the design user is requested, the system will display a list of rule trace along with an canned text attached to each rule for explanation. For the choice of end user, the system provides two different interfaces corresponding to two different types of end users: expert users and student users. It will ask the user a series of questions in order to form an initial model. When the user is an expert the system will give him/her more options to modify his/her list of keywords than a student user.

4.6 Conclusions

In this chapter we have described the functionality of user modelling and information retrieval components which have been added to the Designer's Assistant, an expert system for the design of FIR filters. Two stereotypes have been implemented: expert and novice. Initially a user is assigned to a stereotype but as he/she uses the system the user model is developed through a variety of techniques. When an explanation is required, the user's model is used to select a set of keywords. These keywords are then passed to the information retrieval component which uses these keywords to select relevant passages of text as a basis for explanation. The user model is used to tailor the exact quantity and quality of explanation given.

Chapter 5 CONCLUSIONS AND FUTURE WORK

In this chapter we summarize the results of the thesis and suggest some directions for future work.

5.1 Summary and Conclusions

We have introduced the main concerns and activities of researchers who are interested in both developing a methodology for building expert systems and improving the quality of explanation they produce.

Expert systems must be able to explain what they do and why, but traditional approaches fail to provide adequate explanations and justifications. Ideally, the quality of the explanations could be improved by having the system model what it believes the user knows, developing tutorial strategies giving the system a more global view of its interaction with the user and allowing it to take part in directing it, and improving the system's understanding of its own explanatory capabilities and the user's question so that it can reformulate the user's request into what it can deliver. However, an explanation system should do more than just read back reasoning steps or give canned responses as many currently do.

We suggest that current explanation facilities can be improved by using techniques from user modelling, information retrieval and user interfaces. We have added user models and information retrieval components to the Designer's

Assistant, an expert system for the design of FIR filters. Two stereotypes have been implemented: expert and novice. Initially a user is assigned to a stereotype but as he/she uses the system the user model is developed through a variety of techniques. When an explanation is required, the user's model is used to select a set of keywords. These keywords are then passed to the information retrieval component which uses these keywords to select relevant passages of text as a basis for explanation. The user model is used to tailor the exact quantity and quality of explanation given. Every user gets a different explanation as needed. This is an improvement over traditional explanation facilities.

5.2 Future work

Modelling is a very open-ended task because people have a great deal of diverse knowledge which is structured in many different ways. User models can be developed for every individual user. This is most desirable where one particular system is to be used by people with substantially different background. The systems should know who their users are, the context in which they are trying to work. It is also aimed at improving the performance of the system, both the system external behavior to the interaction (eg., dialogue with the user, information displayed on explanation given, corrections of user's errors and possible suggestions) and adjustment of system internal operation to user's characteristic.

Research on user modeling has been very active in the last few years. A number of interesting proposals have been made concerning representation systems for user models, methods for generating assumptions about the user from his/her natural-language input, and strategies for the planning of the system's dialog contributions on the basis of its model of the user. Also, several prototype systems have been implemented in order to exemplify and further investigate these proposals. However, a great number of fundamental problems still remain unsolved and the development of flexible user modeling components is rather costly.

It seems clear that these more open-ended modelling tasks will require a much richer knowledge base, a series of representational schema to capture the user's knowledge, and an extensive range of inference techniques. Future modelling systems should use their models predictively and report to the investigator situations where the models persistently fail to predict subsequent user behavior. Alternatively, future systems might use additional knowledge to resolve the conflict.

REFERENCES

[AE59]P.H. Ault and E. Emery. *Reporting the news*. Dodd,Mead and Co., New York, 1959.

[BB78]J. S. Brown and R. R. Burton. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 2, 1978.

[BB80]J. S. Brown and R. R. Burton. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 2, 1980.

[BF82]A. Barr and E. A. Feigenbaum. *The Handbook of Artificial Intelligence*. William Kaufman, Menlo Park, CA, 1982.

[BGT87]G. Brajnik, G. Guida, and C. Tasso. User modeling in intelligent information retrieval. *Information Processing and Management*, 23(4), 1987.

[BK81]D. A. Buell and D. H. Kraft. Threshold values and boolean retrieval systems. *Information Processing and Management*, 17(3), 1981.

[Boo80]A. Bookstein. Fuzzy requests: An approach to weighted boolean searches. *Journal of the ASIS*, 31(4), July 1980.

[BW88]R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[Car70]J. R. Carbonell. Ai in cai: An artificial intelligence approach to computer-aided instruction. *IEEE Transactions on Man-Machine Systems*, 11, 1970.

[Car83]J.G. Carbonell. The role of user modelling in natural language interface design. Technical report, Carnegie Mellon University, Department of Computer Science, Pittsburgh, 1983.

[CG77]B. Carr and I. Goldstein. Overlays: A theory of modeling for computer aided instruction. *Massachusetts Institute of Technology*, AI Memo 406, February 1977.

[CL81]W.J. Clancey and R. Letsinger. Neomycin: Reconfiguring a rule-based expert system for application to teaching. In *Proceedings of the International Joint Conference on AI*, 1981.

[Cla83]W. J. Clancey. The epistemology of a rule-based expert system: A framework for explanation. *AI*, May 1983.

[CM84a]K. L. Clark and F. G. McCabe. *micro-PROLOG: Programming in Logic.* Prentice-Hall International, 1984.

[CM84b]W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, second edition, 1984.

[CMN83]S.K. Card, T.P. Moran, and A. Newell. *Thepsychology of Human-Computer Interaction.* Lawrence Erlbaum, Hillsdale, NJ, U.S.A, 1983.

[CNV88]M. A. Covington, D. Nute, and A. Vellino. *Prolog Programming in Depth.*

Compute Books, Scott, Foresman and Company, 1988.

[CTJ89]B. Chandrasekaran, Michael C. Tanner, and John R. Josephson. Explaining control strategies in problem solving. *IEEE Expert*, Spring 1989.

[Dav80]R. David. Metarule: reasoning about control. *AI*, 15, 1980.

[dB83]J. deKleer and J. S. Brown. Assumptions and ambiguities in mechanistic mental models. In D. Genter, A. L. Stevens, and Eds., editors, *Mental Models*. Erlbaum, Hillsdale, NJ, 1983.

[Den67]S. F. Dennis. The design and testing of a fully automatic indexing-searching system for documents consisting of expository text. In G. Schecter, editor, *Information Retrieval: A Critical Review*. Thompson Book Co., Washington, D.C., 1967.

[DST86]E. Drascher, D. Sharpe, and K. Tidwell. *Common Windows Manual*. Intellicorp, MountainView, CA, 1986.

[FH88]A. J. Field and P.G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[Gen82]M. R. Genesereth. The role of plans in intelligent teaching systems. In D. Sleeman, J. Brown, and eds, editors, *Intelligent Tutoring Systems*. London: Academic Press, 1982.

[GSE83]P. Genter, A. L. Stevens, and Eds. *Mental Models*. Lawrence Erlbaum, Hillsdale, NJ, 1983.

[HE80]J. Hobbs and D. Evans. Conversation as planned behavior. *Cognitive Science*, 4, 1980.

[Hen88]James A. Hendler. *Expert Systems: The User Interface*. Ablex, New Jersey, 1988.

[HHW84]J. D. Hollan, E. L. Hutchins, and L. Weitzman. Steamer: An interactive inspectable simulation-based training system. *AI magazine*, 1984.

[HRWLe83]F. Hayes-Roth, D. A. Waterman, D. B. Lenat, and eds. *Building Expert Systems*. Addison-Wesley, Reading, Mass., 1983.

[Jon72]J. SparcK Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1), March 1972.

[LS83]C. Langlotz and E. H. Shortliffe. Adapting a consultation system to critique user plans. Technical report, Stanford University HPP-83-2, April 1983.

[LS87]W. Leigh and A. Smith. *Prolog to Expert Systems*. Mitchell, 1987.

[Luh58]H. P. Luhn. The automatic creation of literature abstracts. *IBM Journal of Research and Development*, 2(2), April 1958.

[MWM85]K. R. McKeown, M. Wish, and K. Matthews. Tailoring explanations for the user. In *Proc. Ninth IJCAI*, Morgan Kaufmann,Los Altos, Ca., Aug. 1985.

[NW77]R.E. Nisbett and T.D. Wilson. Telling more than we can know: Verbal reports on mental processes. *Psychological Review*, 84, 1977.

[OS83]T. O'Shea and J. Self. *Learning and Teaching with Computers: Artificial Intelligence in Education*. Prentice-Hall Inc., Englewood Cliffs, NJ, U.S.A, 1983.

[Pol81]A. S. Pollitt. An expert system as an online search intermediary. *5th International Online Information Meetings*, December 1981.

[Rad76]T. Radecki. Mathematical model of information retrieval based on a concept of a fuzzy thesaurus. *Information Processing and Management*, 12(5), 1976.

[RC85]M. H. Richer and W. J. Clancey. Guidon-watch: A graphic interface for viewing a knowledge-based system. *IEEE Computer Graphics and Applications*, 5(11), 1985.

[Ric79]E. Rich. Building and exploiting user models. Technical report, Carnegie Mellon University, Department of Computer Science, Pittsburgh, 1979.

[Ric82]E. Rich. Stereotpes and user modeling. In A. Kobsa and W. Wahlister, editors, *User Models in Dialog Systems*. Springer-Verlag, 1982.

[Ric83]E.A. Rich. Users are individuals: Individualizing user models. *International Journal of Man-Machine Studies*, 18, 1983.

[RNR81]G. Robertson, A. Newell, and D. Ramakrishna. Zog approach to man-machine communication. *International Journal of Man-Machine Studies*, 14(4), 1981.

[Row84]N. C. Rowe. Modelling degrees of item interest for a general database query system. *International Journal of Man-Machine Studies*, 20, 1984.

[Sar90]A. Sarkar. Implementation of a knowledge base for fir filter design. Technical report, University of Windsor, Department of Computer Science, Windsor, 1990.

[Sch82]B. Schneiderman. The future of interactive systems and the emergence of direct manipulation. *Behavior and information technology*, 1, 1982.

[Sel74]J. A. Self. Student models in computer-aided instruction. *International Journal of Man-Machine Studies*, 6, 1974.

[SFW90]G. Salton, E. A. Fox, and H. Wu. Extended boolean information retrieval. Technical report, Cornell University, Department of Computer Science, Windsor, Ithaca. New York, August 1990.

[Sha51]C. E. Shannon. Prediction and entropy of printed english. *Bell System Technical Journal*, 30(1), January 1951.

[Sho76]E. H. Shortliffe. *Computer Based Medical Consultations: MYCIN*. North-Holland, Amsterdam, 1976.

[Sle82]D. H. Sleeman. Assessing competence in basic algebra. In D. Sleeman, J. S. Brown, and Eds, editors, *Intelligent Tutoring Systems*, London, 1982. Academic Press.

[Sle85]D. Sleeman. Umfe: A user modelling front-end subsystem. *International Journal of Man-Machin Studies*, 23, 1985.

[Sle87]D. H. Sleeman. Pixie: A shell for developing intelligent tutoring systems. In

R. W. Lawler and M. Yazdani, editors. *Artificial Intelligence and Education Volume 1*, Norwood, NJ, 1987. Ablex.

[SM83]G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill Book Co., New York, 1983.

[SSB+81]E. H. Shortliffe, A. C. Scott, M. B. Bischoff, W. van Melle, and C. D. Jacobs. Oncoctin: An expert system for oncology protocol management. In *Proc. Seventh IJCAI*, Vancouver, BC, Aug. 1981.

[SW86]M. Stelzner and M. D. William. Specification by reformulation: An approach to knowledge-based interface design. 1986.

[Swa83]W. R. Swartout. Xplain: A system for creating and explaining expert consulting programs. *AI*, September 1983.

[Sys88a]Quintus Computer Systems. *Quintus Prolog Development Environment (User's Guide, Reference Manual, Library Manual)*. Mitchell, Inc. Mountain View, Ca., 1988.

[Sys88b]Quintus Computer Systems. *Quintus proWindows User's Guide*. Mitchell, Inc. Mountain View, Ca., 1988.

[Tek85]Teknowledge. *S.1 Product Description*. Palo Alto, California, 1985.

[Tor89]T. Toronyi. Explanation facilities for an expert system shell. Technical report, University of Windsor, Department of Computer Science, Windsor, 1989.

[vR79]C. J. van Rijsbergen. *Information Retrieval*. Butterworths, London, 1979.

[VS78]Stairs VS. A tool for the end user. *IBM Scientific and Cross Industry Center*, June 1978.

[WCD82]I. H. Witten, J. G. Cleary, and J. J. Darragh. The reactive keyboard: a new technology for text entry. In *Convergin Technologies: Proceedings of the Canadian Information Processing Society Conference*, Ottawa, Ontario, 1982.

[WH71]P. M. Woods and J. R. Hartley. Some learning models for arithmetic tasks and their use in computer-based learning. *British Journal of Educational Psychology*, 41, 1971.

[WK79]W. G. Waller and D. H. Kraft. A mathematical model for a weighted boolean retrieval system. *Information Processing and Management*, 15(5), 1979.

[WS82]J. W. Wallis and E. H. Shortliffe. *Explanatory Power for Medical Expert Systems: Studies in the Representation of Causal Relationships for Clinical Consultations, Methods of Information in Medicine*. 1982.

[WS89]Michael R. Wick and James R. Slagle. An explanation facility for today's expert systems. *IEEE Expert*, Spring, 1989.

APPENDIX A — INFORMATION RETRIEVAL

A.1 Introduction

Information retrieval (IR) deals with the representation, storage, organization, and accessing of information items and finally retrieving them in response to a user's query. In principle, we have no restriction on the type of information items dealt with in information retrieval. These information files can be personal records, part inventories, customer account information, business correspondence, document holding in libraries, patient records in hospitals and so on.

A.2 Information Retrieval Systems

There are various types of information system. The most important computer-based information systems today are data base management systems, automatic question-answering systems and information retrieval systems. It appears on the surface that the problems which arise in these three information systems are much the same, in the sense that in each case stored information files are processed in response to queries submitted by users, and that answers are generated to these queries. However, in practice, the problems are different. We shall briefly discuss these information systems in the following sections.

A.2.1 Data base management systems

Data base management systems are concerned with the storage, maintenance, and retrieval of data facts available in the systems in an explicit form. That is, they process specific data elements normally stored in two-dimensional tables. Each row of such a table may be used to represent a record included in the file, each column identifies some attributes whose values are then used to distinguish the records from each other. An example of such table is a personnel file whose rows identify the individual employees in an organization and whose columns contain employee numbers, address, job classifications, salary amount and the department numbers for various individuals.

The problems of concern for researchers in data base management then include storage and retrieval of data, the updating or deleting of data, formulations of database queries, the protection of data from unintentional or deliberate misuse or damage, and the transmission of data to remote users or other data management systems.

In data base processing, each search request must state the specific values of certain record identifiers such as age, job classification, salary category and so on for the records of interest. The retrieved information consists of all the records that match the stated search request exactly.

A.2.2 Question-Answering System

Question-answering systems provide access to factual information in a natural language setting. Since in question-answering, queries dealing with particular facts may have to be answered directly, the stored data base often consists of large numbers of facts relating to special areas of discourse, together with general world knowledge covering the context within which conversations between persons usually take place. Typical problems of concern in question-answering system therefore include constructing the knowledge-base to store the facts of interest in a given subject area, storing general world knowledge in a given question-answering situation, analyzing the user query by employing syntactic, semantic and inferencing techniques, comparing the analyzed query with stored knowledge, and generating a suitable response derivable from relevant facts already known.

A.2.3 Information Retrieval Systems

Information retrieval systems are concerned with the representation, storage, and access to documents or representatives of documents. The aim of information retrieval is to retrieve bibliographic references, that is, citations or abstracts to bibliographic items, or retrieve full text of all documents. The input information to the system includes the natural language text of the documents or of document excerpts and abstracts. The output in response to a search request consists of sets of references or the full text of documents. These references are intended to give the system user information about documents of potential interest.

A.3 Activities in Information Retrieval Systems

The task of designing and using an information retrieval system involves four major activities: information analysis, information organization and search, query formulation, and information retrieval and dissemination.

A.3.1 Information Analysis

In a document processing environment, this task is probably the most important and also the most difficult to carry out. The analysis operation, also known as indexing consists of the assignment to the document items of identifiers or index terms capable of representing document content. These index terms are designed to identify and represent stored items and hence they fulfill three related purposes as described by Salton and McGill [SM83].

1. *To allow the location of items dealing with topics of interest to the user*
2. *To relate items to each other, and thus relate the topic areas, by identifying distinct items dealing with similar, or related topic areas.*
3. *To predict the relevance of individual information items to specific information requirements through the use of index terms with well-defined scope and meaning.*

There are two types of identifiers in widespread use: objective attributes and subjective attributes. For example, in a personnel file a person's name, age, and job classification could be used as objective attributes to identify a personnel record. Similarly, in a library file a book's author, publisher, and date of publication could be used to identify the record of a book. In addition to using the values of certain objective attributes for information identification, it is also possible to utilize subjective attributes or content terms to describe stored items.

The automatic indexing process starts with the observation that the words occur in natural language text unevenly; hence, classes of words are distinguishable by their occurrence frequencies. H. P. Luhn [Luh58], one of the pioneers in automatic indexing, stated

"The justification of measuring word significance by use-frequency is based on the fact that a writer normally repeats certain words as he advances or varies his arguments and as he elaborates on an aspect of a subject. This means of emphasis is taken as an indicator of significance.

Based on the frequency characteristics of individual words in document texts, Luhn proposed an algorithm to derive index terms as follows:

1. Given n documents, calculate the frequency of each unique term k in each document i denoted by $FREQ_{ik}$
2. calculate $TOTFREQ_k$ by

$$\text{TOTFREQ}_k = \sum_{i=1}^{n} \text{FREQ}_{ik}$$

which is the total collection frequency for term k

3. Arrange the words in decreasing order using their collection frequency. Decide on some high threshold value, low threshold value and remove all words with collection frequencies above the high threshold and below the low threshold. The removal of words with a collection frequency above high threshold value eliminates high-frequency function words such as *the, of, and, to, a, in, that, is, was, he*. Similarly, the removal of words with a collection frequency below low threshold value deletes terms which occur so infrequently in the collection that their presence does not affect the retrieval performance in a significant way.

4. The remaining medium frequency words are now used as index terms to represent the document content.

This approach has certain disadvantages: the removal of all high frequency words might produce some losses in recall due to the fact that the use of broad, high frequency words is effective in retrieving large numbers of relevant items. The elimination of low-frequency terms might produce losses in precision. By recall and precision are meant the two parameters that measure the effectiveness of retrieval. Recall measures the proportion of relevant information actually retrieved in response to a search (that is, the number of relevant items actually obtained divided by the total number of relevant items contained in the collection). Precision measures the proportion of retrieved items actually relevant (that is, the number of relevant items actually obtained divided by the total number of retrieved items). The other disadvantage is the necessity to choose suitable thresholds in order to distinguish the useful medium-frequency terms from the remainder.

It is desirable that a useful index term must fulfill a dual function: it must represent the content of the document and distinguish the documents to which it is assigned from the remainder to prevent the indiscriminate retrieval of all items. Thus, the use of absolute frequency measures such as $FREQ_{ik}$ or $TOTFREQ_k$ for the identification of content indicators is put into question. This can be illustrated by the use of the term *computer* as a document indicator |"computer" is likely to occur in every collection item and cannot therefore be used to distinguish the items from each other. This consideration suggests the use of "relative frequency" measures to identify terms occurring with substantial frequencies in some individual documents of a collection. Such terms may then help in retrieving the items to which they are assigned, while also distinguishing then from the remainder of the collection. Several approaches have been proposed to derive term weighting functions based on the above consideration. Those include an inverse document frequency function, the signal-noise ratio and the term discrimination value.

A.3.1.1 The inverse document frequency weight.

In this method, it is assumed that term importance is proportional to the standard occurrence frequency of each term k in each document i and inversely proportional to the total number of documents to which each term is assigned. The inverse document frequency can be described as [Jon72]

$$log_2\left(\frac{n}{DOCFREQ_k}\right) + 1 = log_2(n) - log_2(DOCFREQ_k) + 1$$

where n is the number of documents on the collection and $DOCFREQ_k$ denotes the number of documents in which a term k occurs.

A possible weighting function which measures the importance or weight of term k in a given document i can be written as

$$Weight_{ik} = FREQ_{ik} * [log_2(n) - log_2(DOCFREQ_k) + 1]$$

As we can see in the above expression, the weight would increase as the frequency of the term in the document, $FREQ_{ik}$ increases but decrease as $DOCFREQ_k$ increases. Thus this function assigns a high degree of importance of terms occurring in only a few documents of a collection.

A.3.1.2 The signal-noise ratio

In this approach, the construction of measures of term importance is done based on the consideration of information theory. The information content of a message, or term can be measured as an inverse function of the probability of occurrence of the words in a given text. That is,

$$INFORMATION = -log_2(p)$$

where p denotes the probability of occurrence of the word and INFORMATION denotes the information measure of the word. Shannon [Sha51] gives a formula for average or expected information, when a document is characterized by t possible identifiers, or terms, each occurring with a probability p_k as follows

$$AVERAGE\ INFORMATION = -\sum_{k=1}^{t} p_k log(p_k)$$

Analogous to Shannon's information measure, the noise $NOISE_k$ of an index term k for a collection of n documents is

$$NOISE_k = \sum_{i=1}^{n} \frac{FREQ_{ik}}{TOTFREQ_k} log_2\left(\frac{TOTFREQ_k}{FREQ_{ik}}\right)$$

The measure of noise varies inversely with the concentration of a term in the document collection. The signal of term k is defined as follows [Den67]

$$SIGNAL_k = log_2(TOTFREQ_k) - NOISE_k$$

The weight or importance of term k in document i can now be calculated as

$$WEIGHT_{ik} = FREQ_{ik} * SIGNAL_k$$

taking into account both occurrence frequency and the signal.

A.3.1.3 The term discrimination value

In this method, the construction of the term weight or term importance is accomplished by the use of the *"discrimination value"* of a term. The discrimination value of a term measures the degree to which the use of the term will help to distinguish the documents from each other.

$$WEIGHT_{ik} = FREQ_{ik} * DISCVALUE_k$$

where DISVALUE$_k$ denotes the discrimination value of term k.

The derivation of the discrimination value is considered below.

Let D$_i$ and D$_j$ represent two documents each identified by a set of index terms. Define SIMILAR(D$_i$,D$_j$) to be a similarity measure that can be used to represent the similarity between the documents. The value of SIMILAR(D$_i$,D$_j$) runs form 0 to 1 representing no agreement among the assigned index term to perfect agreement. Let D' be the document in which the terms are assumed to exhibit average frequency characteristics, that is,

$$(AVERAGE\ FREQ)_k = \frac{1}{n}\sum_{i=1}^{n} FREQ_{ik}$$

The average similarity is then computed as

$$AVGSIM = constant \sum_{i=1}^{n} SIMILAR(D', D_i)$$

Let (AVGSIM)$_k$ represent the space density (average similarity) of the original document collection with term k being removed from all the documents. Then the discrimination value DISCVALUE$_k$ can be calculated for each term k as

$$DISCVALUE_k = (AVGSIM)_k - AVGSIM$$

It is noted that term with positive DISCVALUE gives good discriminators. Term with DISCVALUE close to zero leaves the similarity among documents unchanged when it is added or removed. Terms with negative DISCVALUE yield poor discriminators whose utilization renders the documents more similar.

A SIMPLE AUTOMATIC INDEXING PROCESS

In this part we will briefly describe a simple process for automatic indexing of a collection of documents. For bibliographic retrieval purposes, it is sufficient to use the titles of documents or abstracts for analysis. However, in many full-text retrieval systems, the full text of the documents is used for indexing purposes. This happens in specialized areas of discourses such as law or medicine. In these areas the vocabulary may be specialized and the presence of a particular term has specific annotations.

The first step in indexing process involves in the identification of all individual words that make-up the documents. Following this is the elimination of high frequency function words. These words comprise of 40 to 50 percent of the text words which are poor discriminators and can not be used to identify the document content. In English there are about 250 common words and these words are included in a so-called *stop-list*. An example of a stop-list is given in the table below

Figure 3 Excerpt from Typical Stop List

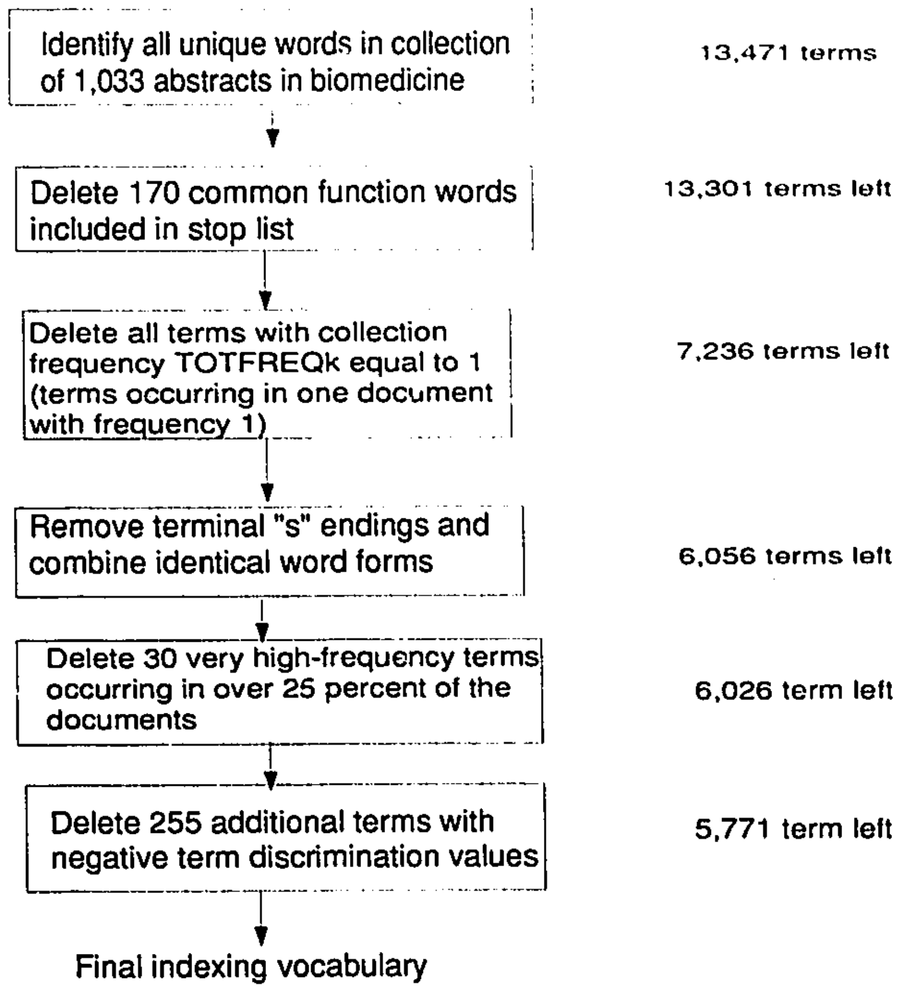
A	Amongst	Becomes
About	An	Becoming
Across	And	Been
After	Another	Before
Afterwards	Any	Beforehand
Again	Anyhow	Behind
Against	Anyone	Being
All	Anything	Below
Almost	Anywhere	Beside
Alone	Are	Besides
Along	Around	Between
Already	As	Beyond
Also	At	Both
Although	Be	But
Always	Became	By
Among	Because	Can
	Become	

The next step is to identify good index terms and to assign them to the documents of a collection. It is useful to use word stems by removing prefixes and suffixes. For example words such as analysis, analyzing, analyzer, analyzed and analyzing are reduced to a word stem "analy" which will have a higher frequency of occurrence in the document texts.

Once the word stems are generated, there is a need to recognize equivalent stems and to choose those stems to be used as index terms. The frequency-based technique considered previously can be employed to determine the potential usefulness of the remaining word stems. A term importance factor can be obtained by using the inverse document frequency $1/DOCFREQ_k$. $DISCVALUE_k$ or the $SIGNAL_k$ may also be used to emphasize precision. The word stems with sufficiently high term value factors can be assigned to the documents of the collection with or without a term weight.

The last step is to delete terms whose important factors are not high enough. An example of an indexing system is given in the simplified form of the flow chart below [Lov 68].

Figure 4 Typical term deletion algorithm (data for 1,033 documents in medicine)


Identify all unique words in collection of 1,033 abstracts in biomedicine
13,471 terms
Delete 170 common function words included in stop list
13,301 terms left
Delete all terms with collection frequency TOTFREQk equal to 1 (terms occurring in one document with frequency 1)
7,236 terms left
Remove terminal "s" endings and combine identical word forms
6,056 terms left
Delete 30 very high-frequency terms occurring in over 25 percent of the documents
6,026 term left
Delete 255 additional terms with negative term discrimination values
5,771 term left
Final indexing vocabulary

The elimination of some broad high-frequency terms may produce some losses in recall and the deletion of low-frequency terms may results in reduced retrieval recall and precision. Thus instead of deleting the high-frequency or low-frequency terms, it is preferable to improve these terms by transforming them into terms with better discrimination properties.

A.3.1.4 Term Thesaurus

The other method to construct index terms is to use context and term association. This can be done by creating a term thesaurus. A thesaurus then provides a classification of terms used in a given topic area into thesaurus classes. A thesaurus may broaden the vocabulary terms by addition of thesaurus class identifiers to the normal term lists, therefore, it enhances the recall performance in retrieval. The thesaurus class identifiers can also replace the original term entries to improve recall and provide vocabulary normalization. The table given below provides an excerpt of a thesaurus used in an automatic indexing environment for engineering documents [SM83].

Figure 5 Typical Thesaurus Excerpt

408	Dislocation
	Junction
	Mino[illegible]y-Carrier
	N-P-N
	P-N-P
	Point-Contact
	Recombine
	Transition
	Unijunction
409	Blast-Cooled
	Heat-Flow
	Heat-Transfer
410	Anneal
	Strain
411	Coercive
	Demagnetize
	Flux-Leakage
	Hysteresis
	Induct
	Insensitive
	Magnetoresistance
	Square-Loop
	Threshold
412	Longitudinal
	Transverse
413	capacitance
	Impedance-Matching
	impedance
	inductance
	Mutual-Impedance
	Mutual-Inductance
	Mutual
	Negative-resistance
	Positive-Gap
	Reactance
	Resist
	self-Impedance
	Self-Inductance
	Self
414	Antenna
	Klystron
	Pulses-Per-Beam
	Receiver
	Signal-to-receiver
	Transmitter
	Waveguide
415	Cryogenic
	Cryotron
	Persistent-Current
	Superconduct
	Super-Conduct
416	Relay

In this case the thesaurus class identifiers are represented by numbers. If the document has the term "anneal" then that term may be replaced by 410. When a document contains the term "anneal" and the query term is "strain", a term match would result through a thesaurus transformation without using the original word stems.

A.3.2 Information Organization and Search

Once the items have been indexed, the information file in which they will be placed must be organized appropriately. A search procedure is designed to locate items in the file by comparing the information requests with the stored document items. There are a variety of different search systems. The first possibility of file creation and search is a sequential scan. In this procedure, the information file is left unorganized and a sequential search is conducted for the whole file. In such a search, a query statement is compared with the identifiers of the first information item, then with those of the second item, and so on until the last item is reached. This approach is inexpensive; however, it can be applied only for small collections of documents, since a sequential scan of the whole file may not be performed fast enough to satisfy a user.

When only a single attribute controls the access to a file, it may be convenient to order the file by the values of that attribute. In this way, we can speed up a search for an information item by developing an sparse index to provide access to segments of a file. The search for a particular item now requires only a search of the sparse index and a search of the portion of the file specified by the index. The organization of a telephone book is an example — access to the file is controlled by a single attribute (subscriber name), is ordered by values of that attribute (position in an alphabetical listing of names), and is supplemented by an index (the names in the upper outer corner of each page) which restricted the file scan to a single page.

In practice, the use of an ordered file and index covering a single attribute type may be unduly restrictive. For example, one might want to access a file using variety of objective attributes, such as customer name, age, address, or bank balance. Furthermore, a single query could also include several types of content identifiers. In principle, several copies of a file could be designed. Each one would be ordered differently — by name, age, balance, and so on. As the number of attributes that are of interest grows, this solution becomes too expensive.

The alternative is to store a single copy of the main file and to build an index structure covering various attributes. This organization is an *inverted file*. In this way an inverted file provides for each acceptable query term a list of the document identifiers corresponding to that term. The documents responding to particular Boolean query statements can be rapidly identified in an inverted file system by using list intersections for terms related by Boolean *and* operators, or list unions for terms connected by Boolean *or* operator.

The main advantage of the inverted file search is that it ensures quick access to the information items because the index alone is examined in order to determine the items which satisfy the search request, rather than the actual file of items. Furthermore, the unsorted main file is relatively easy to maintain. However, its drawback lies in the fact that the size of the index structure may be substantial since pointer lists are maintained for all possible index terms. This is true especially for systems in which the terms attached to the items are weighted to reflect the term's importance, or in which information is included in the index to specify the location of each occurrence of a term in a document. In many practical inverted file systems, the index's size rivals that of the main file, and the need to maintain two large files presents substantial problems.

A.3.3 Query Formulations

A.3.3.1 Boolean query formation and search

We now consider the query formulation process. Most existing systems require an exact match between the query terms and the corresponding keywords attached to the store records or documents. Thus most retrieval systems require search formulation for the query terms. Various methods are available for expressing combinations of query terms. Boolean term combinations in which terms related by the Boolean operator *and*, *or* and *not* are the standard mechanisms. A typical query statement may be formulated as *A and B*, *A or B*, *A and not B* or *(A and B) or C*. These imply, respectively, that the documents to be retrieved must be identified by both term A and term B, by either term A or term B, by term A but not term B, or finally by either term C or the combination of terms A and B. In an inverted file arrangement, boolean queries can be answered by list unions for the *or* operator or by list intersection for the *and* operator.

In bibliographic and full text retrieval systems where individual text words are used as identifiers to the information items, we may formulate a query to request documents dealing with, for example

(FILE AND ORGANIZATION) OR (INFORMATION AND RETRIEVAL)

Most existing retrieval systems also allow the use of truncated terms. We could request documents about "INFORM* AND RETRIE*" where the asterisk represents truncation. The search would then be conducted for items identified by *"information"*, *"inform"*, *"informer"*, or *"informational"* as well as by *retrieve"*, *"retrieval"*, *"retrieving"* and so on.

An example of systems that uses traditional Boolean logic to formulate the search statement is the Lexis system [SM83], the system offered by Mead Data Central to provide a service specifically devoted to the manipulation of legal information. In the Lexis system, the full text of documents are stored and used for automatic indexing process, search and retrieval. The search terms consists of index terms and are connected by the Boolean logic to form search statements.

Common words such as *the*, *it* and *her* are excluded. Hyphenated words such as ANTI-TRUST are stored as two separate terms in the system. Lexis can recognize some special word endings and reduce them to common terms; for example, the terms CITY, CITIES, CITY'S and CITIES' all retrieve the same document set.

Rather than retrieving all documents containing all the given query terms, we may assign weight to each query term and be satisfied with retrieved documents for which a sum of weights of the matching query terms exceeds a given threshold. In this case, the retrieved documents are ranked in a decreasing order according to the sum of the matching term weights.

The use of Boolean queries has a number of disadvantages. For example, in the search responding to an *"or"* query such as A or B or C or D, a document containing all the query terms is equivalent to a document contain only one term. Likewise, for an *"and"* query such as (A and B and C and D) a document with all but one query term is as bad as a document with no query term at all. Thus there are various proposals to generalize and improve the Boolean query formulation process. Among these are fuzzy set models and extended Boolean query formulation. The fuzzy set models are compatible with the conventional Boolean logic and also facilitate the assignment of weights to document identifiers (but not the query terms) [Boo80], [Rad76], [WK79], [BK81]. Unfortunately, the fuzzy set models are subject to the same disadvantages as the conventional Boolean processing models for the "or" query statement and the "and" query statement.

Extended Boolean processing systems overcome these disadvantages by using a relaxed system of interpretation for the Boolean operators [SFW82]. In the extended system, the presence of more identifiers for a given document that match query terms is worth more than the presence of fewer identifiers. This can be accomplished by developing a "generalized distance function" based on Lp vector norm. This distance function is used to compare the Boolean queries with the document identifiers and the interpretation for the Boolean operators can be relaxed through a parameter P involved in the generalized distance function. If $P = \infty$ then the Boolean operators are as in the conventional Boolean logic. The operators *and* and *or* are interpreted less and less strict as P decreases from infinity to 1. For example if P = 10 then an "*and*" operator will favor most rather than all query terms in a document; likewise an "*or*" operator will favor the presence of some query terms in a document rather than the presence of one term when P = 10. The lowest limit of P is 1 and at this limit the "*and*" and "*or*" operators are undistinguished in the sense that the queries (A and B) and (A or B) are treated simply as the vector query (A,B).

A.3.3.2 Matching Functions and Serial Search

Search strategies [Rij79] can also be implemented by making use of a "*matching function*". This function provides a measure of association between a query and a document.

There exist many examples of matching functions in the literature. For example,

$$M = \frac{2|D \cap Q|}{|D| + |Q|}$$

where M is the matching function, D the set of identifiers representing the document, and Q is the set representing the query.

Another example of matching function is the cosine relation used by SMART project. If we let Q = $(q_1, q_2, \ldots, q_t)$ and D = $(d_1, d_2, \ldots, d_t)$ where q_i and d_i are numerical weights associated with the keyword *i*, then the matching function using cosine correlation is given by

$$r = \frac{\sum_{i=1}^{t} q_i d_i}{\left(\sum_{i=1}^{t} (q_i)^2 \sum_{i=1}^{t} (d_i)^2\right)^{\frac{1}{2}}}$$

The use of a matching function in search strategy can be illustrated in a by serial search. If we have a collection of N documents denoted by D_i for i=1 to N, then the serial search proceeds by computing at each document the value of the matching function $M(Q,D_i)$ for the same query. Having obtained N values of $M(Q,D_i)$, the set of documents to be retrieved is determined by two ways.

1. A suitable threshold is given for the matching function. Documents to be retrieved are those having matching function values above the threshold.

2. The documents are ranked in increasing order according to the matching function values. A rank position R is chosen and the set of document to be retrieved is $\{D_i | r(i) < R\}$ where r(i) is the rank position assigned to D_i.

The main problem with this approach is the assignment of the threshold or cut-off. It can be easy see that this process is arbitrary since we can not tell in advance what value of the threshold for each query will give the best retrieval.

A.3.3.3 Probabilistic Information Retrieval

From the above discussion, we see that a document is retrieved in response to a query whenever there is a match between the identifiers assigned to the document and the query keywords. That is, the document is assumed to be relevant to the corresponding query. Another retrieval process based on probability theory has been proposed. In this method, it is assumed that when the index and query terms are sufficiently similar, the corresponding probability of relevance is large enough to make it reasonable to retrieve the document in answer to the query. The rule for retrieval using probability model can be described as follows:

Let $P(Term_i|Rel)$ be the probability of $Term_i$ occurring in a document given that the document is relevant to a given query and let $P(Term_i|Notrel)$ be the probability of $Term_i$ occurring given that the document is not relevant to the query. Then a retrieval function P(Rel|Doc) which is the probability of relevance given a document having a document vector

Doc = <Term_1, Term_2, ..., Term_t>, can be calculated using Bayes' formulae

$$P(Rel|Doc) = \frac{P(Rel|Doc)P(Rel)}{P(Doc)}$$

$$and$$

$$P(Notrel|Doc) = \frac{P(Doc|Notrel)P(Notrel)}{P(Doc)}$$

where P(Rel) and P(Notrel) are the *a priori* probabilities of relevance and non relevance of a document respectively, and P(Doc) is given by

$$P(Doc) \; = \; P(Doc|Rel).P(Rel) + P(Doc|Notrel).P(Notrel)$$

with the above information, the retrieval rule calls for retrieval when

$$P(Rel|Doc) \geq P(Notrel|Doc)$$

$$or\ when$$

$$DISC \; = \; \frac{P(Rel|Doc)}{P(Notrel|Doc)} \; = \; \frac{P(Doc|Rel).P(Rel)}{P(Doc|Notrel)P(Notrel)} \geq 1$$

where DISC is the discriminant function.

The above rule indicates a relationship between the retrieval of the records and the occurrence characteristics of the terms in both the relevant and nonrelevant items. To use the rule we need to determine the probabilities of P(DoclRel) and P(DoclNotrel). If we assume that the terms are independent and the occurrence characteristics are obtained from a sample collection, then

$$P(Doc|Rel) \; = \; P(Term_1|Rel).P(Term_2|Rel)...P(Term_t|Rel)$$

We consider a collection of N documents and assume that R documents out of N are relevant to a given query Q and N-R documents are nonrelevant. The probabilities P(Term$_i$|Rel) and P(Term$_i$|Notrel) which represent the probabilities that a given Term$_i$ occurs in a document given the condition of relevance and nonrelevance, are calculated by

$$
\begin{aligned}
P(Term_i|Rel) &= \frac{r_i}{R} \\
P(Term_i|Notrel) &= \frac{n_i - r_i}{N - R}
\end{aligned}
$$

where r$_i$ denotes the number of relevant documents in which Term$_i$ occurs, and (n$_i$—r$_i$) is the number of nonrelevant documents which contain Term$_i$.

With this approach, the result of a search is an ordered list of documents that satisfy a query, ranked according to their probable relevance.

A.3.4 Information Retrieval and Dissemination

In response to a query statement, the retrieval process involves displaying the identifications of documents such as citations, titles, abstracts, etc. In most existing information retrieval systems, methods for query improvement or reformulation are provided. This process makes use of vocabulary displays which consist of terms related to those included in the original query statements. These related terms may then be used to expand query formulations. There also exist methods for the evaluation of search output. The operation often involves some sort of assessment of relevance by users. If enough relevant items are retrieved, the retrieval operation are considered satisfactory.

In most existing on-line information retrieval systems, the inverted file organization is used and the query formulation techniques such as Boolean connective, term truncation, term weights and so on are incorporated. Furthermore, the on-line system provides additional facilities for formulating better query statements to obtain better output. One of these facilities is the display of the system's search vocabulary. The user can obtain lists of all the terms in the inverted directories and list of synonyms or related terms, and use these terms to expand query statements or to generate alternative query statements.

Another facility is "browsing". This facility provides feedback for query refinement using the displayed abstract of a previously retrieved document, the user can obtain better query terms relevant to documents in interest.

In most on-line systems, it is also possible to save previous query statements. New statements can then be created by combining pieces of old query statements. For example, if query A is "information retrieval" and B is "file organization", then a new, more refined query can be created as "A and B" which covers the area " information retrieval and file organization". An example of on-line retrieval system which has some of the facilities for refining query statement is the well-known Stairs retrieval system [VS78]. The table below shows a typical sequence of refined query statements in the Stairs retrieval system

Figure 6 A sequence of refines query statement in the Stair system.

Query Number	Query Statement	Explanation
1	Health **and** resort **and** rheumatic	Use of boolean connections
2	Health **adjacent** resort **and** rheumat*	Use of term location information and of truncated terms
3	2 **and** hotel; **within sentence** quiet	Refinement of an earlier query statement
4	**Select** 3 **with** full board **less than** 75	Output restriction based on value of certain objective attributes

A.4 Conclusion

In the above we have outlined three types of information system as well as the four major activities involving in designing and using information retrieval systems. These include information analysis, information organization and search, query formation, and information retrieval and dissemination. Researches in information retrieval system are still going to improve upon existing methods for a better search and retrieval of information. Furthermore, advances in many secondary areas also help speed up operations and yield more effective output products. The new developments include the design of user-friendly interfaces and the application of expert systems acting as an intermediary in information retrieval systems [Pot81], [BGT87].

APPENDIX B — USER INTERFACE FOR EXPERT SYSTEMS

B.1 Introduction

Computers today are providing an expanding range of services to a rapidly growing pool of users. But they are not very good at communicating with their users. They often fail to understand what their users want them to do and then are unable to explain the nature of the misunderstanding to the user. In fact, it is the common experience of users of interactive systems, whether novice or experienced, infrequent or regular, that communication with their machines is a time-consuming and frustrating experience.

An expert system, unlike a traditional computer program, its not just a *tool* that implements a process, but rather it is a *representation* of that process. Further, many of these processes correspond to judgments that can have critical real world consequences. The user interface must often present not only conclusions, but an explication of the processes by which those conclusions are reached. The user interface requirements for expert systems have evolved considerably since the days when a consultation system first conducted a dialogue with the user. Larger-scale expert systems have also increased requirements for the user interface to help in managing complexity.

The following sections will represent the requirements for a user interface to expert system, the aspects of the user interface, building tools to support user interface and a general approach to implementing an interface tool.

B.2 The Requirements for User Interfaces to Expert Systems

The *user-system interface* (also called the user interface) is responsible for the communication of information between the applications and the user.

Requirements for interfaces to expert systems should be categorized into those supporting system developers and those supporting end users. The interface requirements for these two classes of users are not the same, although they may overlap. Major differences in these requirements stem from the different focus of the two classes of users

1. The major focus of the developers is on the *representation of the domain* and the reasoning processes.
2. The major focus of the end users is on the *domain* itself.

Developers' interface tools, therefore, will enable the developer to look at the underlying representation of the domain and allow tracing of the various reasoning processes. The end users' interface, on the other hand, should make their mental models of the domain explicit.

To assist both users and developers in managing complexity, major requirements for user interfaces to expert systems are identified as follows:

1. The interface should represent the domain in the user's natural idiom.
2. The interface should provide immediate feedback to the user in the effects of changes to system state by explicitly maintaining and displaying complex constraints and interrelationships.
3. The user must be able to recover easily from trying different alternatives.
4. The user interface must support the user at different interaction styles or levels of abstraction.
5. User interface must be implemented in such a way that it is possible to have multiple interfaces to the same knowledge.

B.2.1 The natural idiom

Mental models are systematic descriptions of objects and relationships between objects [GS83], and computational models are logical extensions when problems are too large or complex to be dealt with using a mental model. The interface needs to map directly onto this perception reflecting how the knowledge in the program maps onto the mental model. In other words, expert systems need to provide interactive displays of the user's model in the natural idiom (form), an idiom that is frequently graphic. Spreadsheets are an early example of using the natural idiom in a decision support environment, while Steamer [HHW84], a teaching system based on the concept of intelligent graphics that make visible to the student the operations of an otherwise abstract and complex steam generator system for large ship, is the canonical example in the AI community. The ONCOCIN project [Sho81] has attempted to provide this type of interface by graphically representing medical documents for doctors to enter patient information.

B.2.2 Immediate feedback

As well as displaying the domain in the natural graphic idiom, the displays must allow the user to act directly upon them, and then provide immediate feedback on the effects of the user's actions.

Immediate feedback on the system state frequently requires that the graphics reflect the running of the model through some sort of animations. Animation aids the *envisionment* process (described by de Kleer as " an inference process which, given the device's structure determines its functions") [KB83] and the user's understanding of system behaviors.

B.2.3 Recoverability

The end users of expert systems tend to be highly skilled knowledge workers, who may or may not be experts in the precise area addressed by the system. Aside from the fact that such users tend to be less patient with "unfriendly" systems, the success of the system is directly related to the users' ability to try out various actions on the system and then change their minds. They must be able to recover easily from situations they don't want to be in and to experiment with different responses without much penalty.

B.2.4 The appropriate interaction styles

As applications become larger and more complex, an overall requirement of the interface is to assist both developers and end users in dealing with complexity. This implies that interface design issues become more important, the larger the application is. To assist in dealing with complexity, the interface must assist the developer in decomposing the problem, and help the end user to understand the relationships between various levels of abstraction. To support these two requirements, the interface must allow both types of users to work at the appropriate interaction styles [Hob85], a style that changes frequently during a user session.

B.2.5 Multiple interfaces to the same knowledge

Multiple, overlapping problems using the same knowledge implies different interfaces for different types of problems, but integrated into the same system. The same knowledge, therefore, will be reflected in more than one interface or view [RC85]. The requirement for multiple views of the same knowledge has implications about the implementation of interfaces.

B.3 Aspects of the User Interface

B.3.1 Dialogue control

Dialogue control refers to the way in which the system and the user interact. The most common form of dialogue in expert systems today is a rigid style consultation with an exhaustive set of yes/no or menu style questions initiated by the system. The user generally has only a limited set of options available for interrupting this dialogue, for example, asking for how or why explanations or for clarification of a question. Options such as these, however, are typically treated by the system as temporary interruptions in its ongoing inquisition, rather than a smooth change of initiative into the user's hands. This lack of flexibility often leads to problems with user acceptance. The main difficulty is that the users have to answer a large number of questions, many of which appeared to be irrelevant. For example, in some areas users might just want to consult a system for one particular piece of information, rather than being locked into a full consultation. Similarly, an experienced user might not want advice on a particular remedy for a problem, but might want to find out answers to questions such as "why does a particular remedy work".

Expert systems must be good consultants as well as good problem solvers. When developing an expert system, designers need to ensure that the dialogue facilities of the system match the communication needs of the users and the constraints of the task environment. In a trouble-shooting situation users often want to volunteer information of their own choice, fairly rapidly and in an order which suits them. Without a natural language interface, however, providing an adequate volunteer facility poses a difficult man machine interface problem.

B.3.2 Natural language processing

The most natural and familiar of expression is for most human beings, their natural language. Unfortunately, natural language as a system is too vague and poorly understood to be used as a complete model for interactive computer languages. There is even the possibility that certain aspects of natural language are fundamentally incompatible with the requirements of man-machine interaction. However, none of these argues against the possibility that certain well-understood attributes of users' natural language can provide a useful starting point for human engineering of interactive command languages.

The field of natural language processing is even more at the research stage than is that of user models, particularly with regard to interfaces to expert systems. Full natural language interfaces are still more of an aspiration than a reality.

B.3.3 New technology

Many systems today offer "wide bandwidth" interfaces where the user is presented with a high resolution bit mapped screen with "windows". These windows allow the simultaneous display of, and interaction with, different parts of the software. Such systems are also generally equipped with a "mouse control" which can be used to identify and manipulate different parts of the display. These windowing and mouse control features are claimed by system developers to lead to greatly enhanced man-machine communication.

B.4 Review of Current Building Tools to Support User Interfaces

New user interface design approaches are using graphical and pictorial representation as an integral part of the dialogue, rather than as just an optional way of displaying data. The appropriate use of graphical presentation has the great advantage, since a human can extract, process, understand, and respond to the relevant information from a visual scenes which is much more fundamental than the ability to manipulate data verbally or arithmetically.

IntelliCorp markets knowledge-based programming machine tools appropriate for building a wide range of expert systems applications. The general approach to supporting the design and development of user interfaces to expert systems has been to develop a layered set of tools that facilitate design and implementation of highly interactive, graphic interfaces rather than textual interfaces. The goal is to free system developers from many of the details of building graphic interfaces so that they can focus their attention on matching the interfaces to the users' cognitive task.

B.4.1 The KEE interface

The Knowledge Engineering Environment (KEE) software development system is an integrated package of tools that allows programmers to develop knowledge-based systems without extensive training in AI technologies.

The first release, KEE 1, had early versions of the KEE interface featuring the unit display and the knowledge-base graph. These displays help the developer to examine knowledge-base structures at both the object level and the object hierarchy level. By the time that KEE 1.2 was released an "active rule graph" had been added. This graph animates the firing of rules, assisting the developer in debugging rules, and the end user in understanding the reasoning strategies the system is taking.

B.4.2 The ActiveImages package

ActiveImages is a package that provides facilities for building control panel interfaces to knowledge structures in knowledge-based applications [SW86]. The user interfaces built with ActiveImages explicitly mimic operator panels for many control systems. The ActiveImages package provides a set of predefined graphic objects, such as gauges, thermometers, bargraphs, and histograms, which can be used to display and control the state of a knowledge-based system.

Because ActiveImages can be created quickly and easily, system developers frequently build control panels to help them debug their applications. This is a case where an end user interface is also valuable for system developers.

B.4.3 Slotgraph

By the time that KEE 2.1 was released, the need for a general interface tool reflecting user-specified relationships between objects had been determined. Slotgraph creates graphs of user-specified relations, objects in slotgraph displays are active, and commands relevant to the object can be obtained by mousing on the object name.

B.4.4 SimKit interface tools

SimKit is a set of knowledge-based simulation and modeling tools. The SimKit Model Editor, provides a direct manipulation interface for building complex system models. Supporting the Model Editor is a Library Editor, which assists the system developer in establishing the relationships between model objects and relationships and their graphic representations. SimKit also supports the graphic specification of part/whole or composite/component relationships, a facility that assists the user in viewing a problem at multiple levels of abstraction.

B.4.5 Common Windows

Common Windows [DST86] a window system that is an extension to Common Lisp. Common Windows places the emphasis on system building tools by feature constructs and techniques that have proven useful in previous window systems such as Interlisp-D and Zetalip window systems. Many high-level facilities have been provided to handle the most common problems which interface builders face, such as scrolling and hotspotting (making a region of a window mouse-sensitive).

B.4.6 KEEpictures Toolkit

KEEpictures, object-oriented graphics toolkit, enables users of the KEE system to construct a variety of iconic interfaces. Its intended uses range from simple graphic displays to sophisticated animation techniques.

The KEEpictures environment provides a set of primitive standard picture classes, such as circle, axes, lines, and strings. Each of these primitives can be shaped, enlarge, shrunk, and altered in a variety of ways. Pictures can be combined together to form larger, composite pictures. As well as using the standard building blocks provided by KEEpictures, users can also draw their own picture using bitmaps. These bitmaps can then be added together or to other types of pictures, to form the final picture.

B.5 A General Approach to Implementing Interface Tools

KEEpictures, ActiveImages, and SimKit all implement their various interface primitives through object-oriented programming, using inheritance over object hierarchies of class/subclass and class/member relationships in KEE. Behavior of the graphic objects is implemented by sending the object a message to create a member of itself, delete, move or reshape itself. Specialization of class definitions provides modularity when defining object behaviors.

These tools have a combination of interaction styles, ranging from direct manipulation [Sch82] to mouse-and-menu to conversational. The choice among these styles depends on a number of factors as follows

1. A mouse-and-menu style is used when the user must make a choice, and the choices can be constrained to a limited set of options.
2. A conversational style is used when the choices to be made by the user cannot be constrained, as is typically the case when the user is naming objects or object attributes.
3. A direct manipulation style is used in selected situations when user options can be preconceived and sufficiently limited.

Flexibility is critical to the success of these tools also because of the evolving nature of interface requirements and special purpose requirements form specific domains. To relax the standard trade off between ease of use and flexibility, an approach to interface design called *specification by reformulation* [SW86] has been introduced. This approach involves making the results of a partial specification quickly visible, and then allowing the user to refine the specification incrementally and view the results. This approach requires a particular relationship between knowledge representation and interface design. The interface design features two distinct, but integrated interfaces:

1. An intelligent, interactive interface that enables the user to create quickly

approximate specifications, or prototypes.

2. A programmatic interface that enables the user to modify and reformulate the products of the interactive interface. The programmatic interface, while harder to use than the interactive interface, is optimized for expressiveness and completeness.

Reformulation is dependent on a declarative representation of the graphic images for communication between the two interfaces.

B.6 Conclusion

In the above we have reviewed the requirements for user interfaces to expert systems, its aspects and building tools to support user interfaces as well as a general approach to implementing interface tools. One of the clearest lessons learned from the early pioneering expert systems is that excellent decision making performance in itself will not guarantee user acceptability. Attention also needs to be paid to the user interface. Failure to recognize the man-machine interface needs of expert system users is probably the biggest reason for the disparity between the numerous expert systems which have been successfully developed in a laboratory and the small number which have actually made it into everyday field use. In the laboratory expert systems tend to be used by people who love them and are tolerant of their idiosyncracies. Outside the laboratory they will only be used if people find them useful and easy to work with.

Until fairly recently the user interface was not considered to be important enough to justify design effort separate from that required for the knowledge base, inference engine and other knowledge base specific software. It was simply something which was tagged on the end. Many organizations, however, are now realizing the importance of the user interface and are putting research effort in this direction.

PROGRAM LISTING

```
/* bring in needed libraries */

:- use_module(library(dialog)).
:- use_module(library(fromonto)).
:- use_module(library(strings)).
:- use_module(library(basics)).
:- use_module(library(files)).
:- use_module(library(messages)).
:- use_module(library(sets)).

/* make necessary procedures dynamic */

:- dynamic known/2, default/2.

executable_dir([parm, prop,known, explanation,
                store_value,append,write,nl,
                fail,!,read, user_def_val,
                user_def_fuzzy_val,type_of_parameter,
                append_tolerance_to_x, assert, retract,
                quantise,execute_all]).

/* cause unknown procedures to fail*/

:- prolog_flag(unknown, _, fail).

add_and_check(Name, Response, Query, Return, Dict_key) :-
   assert(user_response_database(
   Name, Response, Query, Dict_key)),
   !,
   check_value_to_response(Response, Return).
```

```
add_ans(Window, Dictionary, Answers) :-
   send(Window, clear),
   remove_dict_items(Dictionary),
   add_new_ans(Dictionary, Answers, 1),
   send(Window, dict, Dictionary).

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

add_new_ans(Dictionary, [Label | Rest], N) :-
   convert_to_string(N, Num_string),
   concat('(', Num_string, String),
   concat(String, ') ', String1),
   concat(String1, Label, Label1),
   send(Dictionary, append, dict_item(Label, Label1, 0)),
   Next is N + 1,
   add_new_ans(Dictionary, Rest, Next).

add_ques(Window, Question) :-
   send(Window, clear),
   get(Window, size, size(Width, _)),
   Length is Width//9,
   build_length_list(Question, Length, List),
   display_message_strings(Window, List),
   send(Window, line_no, 0),!.

blank_line(how) :-
   send(@how_display_window, print, ' ').

blank_line(whynot) :-
   send(@whynot_wind, print, ' ').

blank_line(_) :- nl.

build_length_list('end of string', _, []).
build_length_list(String, Length, [Line | Tail]) :-
```

```
   get_string_of_length(String, Length, Line, Remainder),
   build_length_list(Remainder, Length, Tail).

call_sda(_,_) :-
   clean,
   unix(shell(opensda1)).

change_previous_response(Dict_key, Object) :-
   user_response_database(Property,
      Previous_response, Query, Dict_key),
   determine_question(Query, Parm_prop, Dimension),
   save_current_display(Object),
   remove_response(Property),
   execute_previous_question(Parm_prop, Property,
                             Dimension, New_response),
   !,
   check_new_response(Previous_response, New_response),
   reset_previous_display(Object).

check_default(Parameter, Item) :-
   default(Parameter, Default_value),
   !,
   send(Item, selection, Default_value).

check_default(_, Item) :-
   send(Item, selection, '').

check_file_header(Stream) :-
   system_name([Line1, Line2, Line3, Line4]),
   read(Stream, Input1),
   Line1 = Input1,
   read(Stream, Input2),
   Line2 = Input2,
   read(Stream, Input3),
   Line3 = Input3,
   read(Stream, Input4),
   Line4 = Input4,
   !.

check_file_header(Stream) :-
   close(Stream).

check_his(_, History) :-
   nonvar(History),
   !.

check_his(Rule, [History]) :-
   convert_to_string(long, Rule, String),
   concat('I cannot prove ', String, History),
   !.

check_new_response(X, X) :- !.

check_new_response(_, _) :-
   (\+ change_in_database ->
      assert(change_in_database)
   ; otherwise -> true),
   reset_cursor(@user_log_window),
   display_message(@message_window,
           'Evaluating change made...please wait'),
   !,
   fail.

check_parameters(_, 0, _).

check_parameters(Rule, Number, New_rule) :-
   arg(Number, Rule, X),
   is_a_list(X),
   length(X, Length),
   (Length > 8 ->
          get_short_form(X, Short, Length)
    ; otherwise ->
```

```
        Short = X),
    arg(Number, New_rule, Short),
    Next is Number - 1,
    check_parameters(Rule, Next, New_rule).

check_parameters(Rule, Number, New_rule) :-
    arg(Number, Rule, Parameter),
    arg(Number, New_rule, Parameter),
    Next is Number - 1,
    check_parameters(Rule, Next, New_rule).

check_remainder(Cond_list, ' and', _) :-
    functor(Cond_list, ',', _),
    !.

check_remainder(explanation(_, _), '', 0) :- !.

check_remainder(explanation(_, _), ') then', if) :- !.

check_remainder(_, ') then', if).

check_remainder(_, ' and', _).

check_session_file(Filename, Stream) :-
    file_exists(Filename),
    open(Filename, read, Stream),
    check_file_header(Stream),
    display_message(@message_window,
          'Loading old session...Please wait'),
    create_log_window,
    !.

check_session_file(Filename, _) :-
    file_exists(Filename),
    !,
    send(@load_window, flash),
   create_load_window('4.1 File is not of FFDES format.'),
   process_message_1.

check_session_file(_, _) :-
   send(@load_window, flash),
   create_load_window('4.1 File does not exist.'),
   process_message_1.

check_true([[Predicate, true], 1], [p, History]) :-
   convert_to_string(long, Predicate, String),
   concat(String, ' is true.', History),
   !.

check_true([[r, Predicate, true], 1], [p, History]) :-
   convert_to_string(long, Predicate, String),
   concat(String,' is true.', History),
   !.

check_true([[r, Predicate, [true]], 1], [p, History]) :-
   convert_to_string(long, Predicate, String),
   concat(String, ' is true.', History),
   !.

check_true(Rule, Rule).

check_value_to_response(Value, 'unknown') :-
   Value = 'unknown',
   !.

check_value_to_response(Value, Response) :-
   Value = Response.

clean :-
   kill_prowindows,
   retractall(history(_)),
   retractall(user_response_database(_, _, _, _)),
    retractall(change_in_database),
    retractall(whynot_history(_,_)),
    retractall(old_cursor(_,_)),
    retractall(executable(_,_)),
    retractall(error_complete_flag),
    retractall(change_in_progress),
    retractall(last_query(_)),
    retractall(default(_,_)),
    retractall(builtin(_)),
    retractall(calculated_values(_,_)),
    retractall(selected(_,_,_,_,_)),
    retractall(conds(_, _)),
    retractall(rules(_, _)),
    retractall(functor_list(_)),
    retractall(user_name(_)),
    retractall(knowledge_level(_)),
    retractall(quantity_explanation(_)),
    retractall(quality_explanation(_)),
    retractall(goal(_)),
    retractall(user_level(_)),
    retractall(end_user(_)),
    retractall(explanation(_)),
    retractall(user_data(_,_,_,_,_,_,_)),
    retractall(user_list(_)),
    retractall(keyword_list(_)),
    retractall(user_yes),
    retractall(add_keyword),
    retractall(delete_keyword),
    retractall(replace_keyword),
    retractall(change_explanation_flag),
    retractall(why_complete1),
    retractall(why_complete).


confirmed(_, _, _) :-
    change_in_database,
   !,
   fail.

confirmed(explanation(_, _), 0, _) :-
   !.

confirmed(true, 1, _) :-
   !.

confirmed(error_message(Property, String), 1, _) :-
   !,
   error_message(Property, String).

confirmed(Rules, [History_1 | [History_tail]], Level) :-
   functor(Rules, ',', _),
   arg(1, Rules, Rule_1),
   !,
   confirmed(Rule_1, Save_his, Level),
   check_true(Save_his, History_1),
   check_his(Rule_1, History_1),
   arg(2, Rules, Rules_tail),
   !,
   confirmed(Rules_tail, Save_tail, Level),
   check_true(Save_tail, History_tail),
   check_his(Rules_tail, History_tail).

confirmed(Negated_rule, History, Level) :-
   functor(Negated_rule, '\+', _),
   arg(1, Negated_rule, Rule),
   !,
   \+ confirmed(Rule, History, Level),
   check_his(Rule, History).

confirmed((Condition -> True;otherwise-> False),
          History, Level) :-
   !,
   (Condition -> confirmed(True, History, Level)
   ; confirmed(False, History, Level)).

confirmed((Condition -> True; False), History, Level) :-
   !,
   (Condition -> confirmed(True, History, Level)
   ; confirmed(False, History, Level)).

confirmed((Condition -> True), History, Level) :-
   !,
   (Condition -> confirmed(True, History, Level)).

confirmed(Rule, [[r, Rule, Cond_used] | [History]], Level) :-
   functor(Rule, F, _),
   executable_dir(Executables),
   nonmember(F, Executables),
   dynamic_dir(L),
   memberchk(F,L),
   preprocess(Rule, Condition_set),
   !,
   confirm_rule(Rule, Condition_set, Cond_used, History, Level),
   check_his(Rule, History).

confirmed(Rule, [p, String], _) :-
   functor(Rule, append, _),
   !,
   Rule,
   convert_to_string(long, Rule, String1),
   concat(String1, ' is true.', String).

confirmed(!, 0, Level) :-
   assert(cut(Level)).

confirmed(execute_all(Pred), History, Level) :-
   !,
   execute_all(Pred, History, Level).
```

```
confirmed(Rule, [p, String], _) :-
   insert_builtin_dict(Rule),
   Rule,
   test_and_convert_to_string(Rule, String),
   delete_builtin_dict,!.

confirmed(Rule, 0, Level) :-
   assert(executable(Level,Rule)),
   delete_builtin_dict,!,
   fail.

test_and_convert_to_string(nl,' ').
test_and_convert_to_string(!, ' ').
test_and_convert_to_string(Rule, ' ') :-
   functor(Rule, write, _).
test_and_convert_to_string(Rule, String) :-
   convert_to_string(long, Rule, String1),
   concat(String1, ' is true.', String).

confirm_rule(_, [], _, _, _) :-
   !,
   fail.

confirm_rule(Rule, [Condition_set | _],
            Conditions, History, Level) :-
   instantiate_rule(Rule, Condition_set, Conditions),
   insert_rule_dict(Rule, Conditions, Level),
   Next_level is Level + 1,
   confirmed(Conditions, History, Next_level),
   check_his(Rule, History),
   delete_rule_dict(Level),
   retractall(cut(Next_level)),
   !.

confirm_rule(_, _, _, _, Level) :-
   change_in_database,
   Next is Level + 1,
   retractall(cut(Next)),
   !,
   fail.

confirm_rule(Rule, [Conditions | Tail],
                 Conds, History, Level) :-
   save_whynot_history(Conditions,Level),
   delete_rule_dict(Level),
   !,
   Next is Level + 1,
   (cut(Next) ->
      retractall(cut(Next)),
      fail
   ; otherwise ->
      confirm_rule(Rule, Tail, Conds, History, Level)).


convert_1(X, Z) :-
   name(X, Y),
   append(Y, [0'.], W),
   read(Z) from_chars W.

convert_to_string(X, Y) :-
   write(X) onto_chars Z,
   name(Y,Z).

convert_to_string(long, Rule, String) :-
   functor(Rule, Predicate, Arity),
   functor(New_rule, Predicate, Arity),
   check_parameters(Rule, Arity, New_rule),
   convert_to_string(New_rule, String).


default(filter_name, 'fir_filter').
default(highest_frequency, 3000).
```

```
delete_builtin_dict :-
   builtin(String),
   retractall(builtin(_)),
   print_string(String, 0, why, delete).

delete_rule_dict(Level) :-
   rules(Level, Rule_key),
   retractall(rules(Level, _)),
   print_string(Rule_key, 0, why, delete),
   remove_conditions_dict(Level).

determine_explain_type(string(Text),string(t)) :-
   add_ques(@why_text, Text).

determine_explain_type(string(Filename),string(f)) :-
   file_exists(Filename),
   send(@why_rule_window, show, false),
   why_file_view,
   open(Filename, read, In_stream),
   set_input(In_stream),
   read(Text_string),
   why_file_read(Text_string),
   close(In_stream),
   process_callbacks(retract(done_view)),
   !.

determine_explain_type(string(Filename),string(f)) :-
   concat('The file ', Filename, String),
   concat(' does no exist.', String, String1),
   send(@why_text, print, String1),
   send(@why_text, print, 'Inform knowledge base engineer.').

determine_explain_type(string(Graphics),string(g)) :-
   send(@why_text, print, Graphics).
```

```
determine_question(parm(_, Dim), parm, Dim).

determine_question(parm(_), parm, []).

determine_question(prop(_), prop, []).

display_and_save('if_construct', _, _, _).
display_and_save(String, Ident, Tail,
            [Spaces, Where, If, Level]) :-
   (Ident == 0 ->
      check_remainder(Tail, End_string, If),
      concat(Spaces, String, String1),
      concat(String1, End_string, Output)
   ;  Output = String),
   print_string(Output, Ident, Where, append),
   save_conds_key(Level, Output, Ident, Where).

display_user_response(Value, Name, Dict_key) :-
   convert_to_string(Value, Value_string),
   convert_to_string(Name, Name_string),
   concat(Name_string, ' : ', Temp),
   concat(Temp, Value_string, Dict_key),
   send(@user_log_dict, append, Dict_key).


ensure_proper_response(Object, Message, Name, Value) :-
   Message = message(Object, Name, Value),
   \+ Value = why,!.
ensure_proper_response(Object, Message, Name, Value) :-
   Message = message(Object, Name, Value),
   \+ Value = change_user_info,!.
ensure_proper_response(Object, _, Name, Value) :-
   get(@sys, wait, Message),
   ensure_proper_response(Object, Message, Name, Value).
```

```
execute_all(Rule, History, Level) :-
   bagof((Rule, Conditions), clause(Rule, Conditions), List),
   test_preconditions(List, Sorted_list),
   execute_confirmed(Rule, Sorted_list, History, Level).

execute_confirmed(_, _, _, _) :-
   change_in_database,
   !,
   fail.

execute_confirmed(_, [], 0, _) :- !.

execute_confirmed(Rule, [Conditions | Tail],
              [History_1 | History], Level) :-
   insert_rule_dict(Rule, Conditions, Level),
   Next_level is Level + 1,
   confirmed(Conditions, History_1, Next_level),
   check_his(Rule, History_1),
   delete_rule_dict(Level),
   execute_confirmed(Rule, Tail, History, Level),!.

execute_confirmed(Rule, [_ | Tail], History, Level) :-
   delete_rule_dict(Level),
   execute_confirmed(Rule, Tail, History, Level).


execute_message(_, view_response,
            Key, Object, Name, Value) :-
   !,
   user_response_database(Property, _, _, Key),
   get_property_question(Property, Question),
   open_view_ques_ans,
   add_ques(@ques_ans_window, Question),
   get(@sys, wait, Message),
   get_specific_message(Message, ques_done),
   send(@ques_ans_window, show, false),
   get_message(Object, Name, Value).

execute_message(_, response_change, Key,
                            Object, Name, Value) :-
   send(Object, show, false),
   !,
   assert(change_in_progress),
   change_previous_response(Key, Object),
   retractall(change_in_progress),
   send(Object, show, true),
   get_message(Object, Name, Value).

execute_message(Object, Name, Value, Object, Name, Value):- !.

execute_precondition(Precond, _Weight) :-
   (Precond = [] ->
          !
    ; otherwise ->
          Precond).

execute_previous_question(parm, Prop, [], Value) :-
   !,
   parm(Prop, Value).
execute_previous_question(parm, Prop, Dim, Value) :-
   !,
   parm(Prop, Value, Dim).
execute_previous_question(prop, Prop, _, y) :-
   prop(Prop).


explanation(_,_).


generate_question_string(X, [], X).

generate_question_string(Question,[Head | Tail],
                                     Return_string) :-
   concat(Question, Head, String),
   generate_question_string(String, Tail, Return_string).


generate_if_then_else(Condition, True, False,
                        [Spaces, Where, _, Level]) :-
   concat(Spaces, '     ', New_spaces),
   (functor(Condition, ',', _) ->
      arg(1, Condition, Cond),
      arg(2, Condition, Tail),
      write('cond : '),write(Cond),
      write('string : '),write(String),
      write('ident : '),write(Ident),
      write('level : '),write(Level),
      generate_string(Cond, String, Ident,
            [New_spaces, Where, if, Level]),
      concat('if (', String, Output),
      display_and_save(Output, Ident, Tail,
            [Spaces, Where, if, Level]),
      process_conditions([New_spaces, Where,
                          if, Level], Tail)
   ; otherwise ->
      generate_string(Condition, String, Ident,
               [New_spaces, Where, if, Level]),
      concat('if (', String, Output),
      display_and_save(Output, Ident, [],
                  [Spaces, Where, if, Level])),
   process_conditions([New_spaces, Where, 0, Level], True),
   process_conditions([New_spaces, Where, 0, Level], False).


generate_string(explanation(String, Ident), String, Ident, _) :-
   !.

generate_string((Condition -> True; False),
                  'if_construct', _, Control) :-
   generate_if_then_else(Condition, True, False, Control).

generate_string((Cond -> True;otherwise-> False),
                          'if_construct', _, Control) :-
   generate_if_then_else(Cond, True, False, Control).

generate_string((Condition -> True),
                            'if_construct', _, Control) :-
   generate_if_then_else(Condition, True, [], Control).

generate_string(Condition, String, 0, _) :-
   functor(Condition, '\+', _),
   arg(1, Condition, Cond_1),
   convert_to_string(long, Cond_1, String1),
   concat(String1, ' is false', String),
   !.

generate_string(Condition, String, 0, _) :-
   convert_to_string(long, Condition, String).


get_message(Object, Name, Value) :-
   get(@sys, wait, Message),
   ( clause(change_in_progress,_) ->
        ensure_proper_response(Object, Message, Name, Value),
        retractall(change_in_progress)
     ; otherwise ->
        Message = message(Object1, Name1, Value1),
        !,
        execute_message(Object1, Name1, Value1,
                             Object, Name, Value)).


get_preconditions(Y, Pre, Weight, Cond) :-
   functor(Y, ',', _),
   arg(1, Y, precondition(Pre)),
   arg(2, Y, Tail),
   arg(1, Tail, weight(Weight)),
   arg(2, Tail, Cond),
   !.

get_preconditions(Y, [], [], Y).

get_property_question(Property, Question) :-
   question(Property, _, Question),
   !.

get_property_question(Property, Question) :-
   question(Property, _, Question, _),
   !.

get_property_question(Property, Question) :-
   question(Property, Question).

get_short_form(X, Short_x, Length) :-
   get_short_x(4, X, Short_x, Length).

get_short_x(0, _, [..., Length], Length).
get_short_x(Number, [Head | Tail], [Head | Short], Length) :-
   Next is Number - 1,
   get_short_x(Next, Tail, Short, Length).


insert_builtin_dict(Rule) :-
   convert_to_string(long, Rule, String),
   assert(builtin(String)),
   print_string(String, 0, why, append).


insert_rule_dict(Rule, Conditions, Level) :-
   convert_to_string(long, Rule, String),
   concat(String, ' if', String1),
   assert(rules(Level, String1)),
   print_string(String1, 0, why, append),
   process_conditions(['    ', why, 0, Level], Conditions).


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

instantiate_rule(Rule, Predicate, Conditions) :-
   arg(1, Predicate, Rule),
   arg(2, Predicate, Conditions).


is_a_list(X) :-
   var(X),
   !,
   fail.

is_a_list([_ | _]).

known(Property, Value) :-
   user_response_database(Property, Value, _, _),
   \+ Value == unknown,!.
known(Prop, Value) :-
   calculated_values(Prop,Value).
known(Property, Value) :-
   user_response_database(Property, Value, _, _),!.


length_of_list(List, Arity, Num) :-
   functor(List, ',', _),
   !,
   arg(2, List, Tail),
   Num1 is Num + 1,
   length_of_list(Tail, Arity, Num1),
   !.
```

```
length_of_list(_, Arity, Num) :-
   Arity is Num + 1.


load_previous_responses(Stream) :-
   read(Stream, [Property, Response, Query, Dict_key]),
   assert(old_session_response(Property,
            Response, Query, Dict_key)),
   !,
   load_previous_responses(Stream).
load_previous_responses(_).

parm(Parameter, _, Return) :-
   old_session_response(Parameter, Response, Query, Dict_key),
   !,
   send(@user_log_dict, append, Dict_key),
   retractall(old_session_response(Parameter, _, _, _)),
   add_and_check(Parameter, Response, Query, Return, Dict_key).

parm(Parameter, Value, Dimension) :-
   user_response_database(Parameter, 'unknown', _, _),
   !,
   retractall(user_response_database(Parameter, _, _, _)),
   parm(Parameter, Value, Dimension).

parm(Parameter, _, Value) :-
    user_response_database(Parameter, StoredValue, _, _),
    !,
    Value = StoredValue.

parm(Parameter, _, Value) :-
   calculated_values(Parameter, Value),!.

parm(Parameter, List, Dimension) :-
   question(Parameter, point, Question),
   send(@message_window, clear),
   open_point_window(Question),
   get_message(@point_window, Name, Response),
   !,
   process_chosen(@point_window, Name, Response,
                        Return_list, Dimension),
   send(@point_window, show, false),
   !,
   display_user_response(Return_list, Parameter, Dict_key),
   add_and_check(Parameter, Return_list,
           parm(Parameter,Dimension), List, Dict_key).

parm(Parameter, Value, Alternate_answers) :-
   question(Parameter, Type, Question, _),
   send(@message_window, clear),
   parm_menu(Parameter, Type, Question,
                Alternate_answers, Value).


parm(Parameter, Return) :-
   old_session_response(Parameter, Response,
                              Query, Dict_key),
   !,
   send(@user_log_dict, append, Dict_key),
   retractall(old_session_response(
                   Parameter, _, _, _)),
   add_and_check(Parameter, Response,
                        Query, Return, Dict_key).

parm(Parameter, Value) :-
   user_response_database(Parameter, 'unknown', _, _),
   !,
   retractall(user_response_database(Parameter, _, _, _)),
   parm(Parameter, Value).

parm(Parameter, Value) :-
    user_response_database(Parameter, StoredValue, _, _),
     !,
     Value = StoredValue.

parm(Parameter, Value) :-
   calculated_values(Parameter, Value).

parm(Parameter, Value) :-
   question(Parameter, Type, Question, Answers),
   asserta(save_user_def_val(Parameter)),
   send(@message_window, clear),
   !,
   parm_menu(Parameter, Type, Question, Answers, Value),
   retractall(save_user_def_val(_)).

parm(Parameter, Value) :-
   question(Parameter, Type, Question),
   !,
   asserta(save_user_def_val(Parameter)),
   send(@message_window, clear),
   parm_atom(Parameter, Type, Question, Value),
   retractall(save_user_def_val(_)).

parm(Parameter, Value) :-
   question(Parameter, Question),
   asserta(save_user_def_val(Parameter)),
   send(@message_window, clear),
   parm_atom(Parameter, a, Question, Value),
   retractall(save_user_def_val(_)).

parm_menu(Parameter, c, Q, A, Value) :-
   sort_if_default(Parameter, A, Sorted_A),
   add_ans(@parm_brows, @parm_menu_dict, Sorted_A),
   open_parm(Q, menu_ques, Parameter),
   get_message(@menu_ques, Name, Response),
   !,
   process_chosen(@menu_ques, Name,
                    Response, Return_value, _),
    send(@menu_ques, show, false),
    display_user_response(Return_value, Parameter, Dict_key),
    !,
    add_and_check(Parameter, Return_value,
            parm(Parameter, A), Value, Dict_key).

parm_menu(Parameter, list, Q, A, List) :-
    open_selection_window(Q, A),
    get_message(@selection_window, Name, Response),
    !,
    process_chosen(@selection_window, Name,
                    Response, Return_list, _),
    send(@selection_window, show, false),
    display_user_response(Return_list, Parameter, Dict_key),
    add_and_check(Parameter, Return_list,
                            parm(Parameter, A), List, Dict_key).


parm_atom(Parameter, a, Q, Value) :-
    open_parm(Q, atom_ques, Parameter),
    get_message(@atom_ques, Name, Response),
    !,
    process_chosen(@atom_ques, Name, Response, Return_value, _),
    send(@atom_ques, show, false),
    convert_l(Return_value, Y),
    display_user_response(Y, Parameter, Dict_key),
    !,
    add_and_check(Parameter, Y, parm(Parameter), Value, Dict_key).


parm_atom(Parameter, slider, Q, Value) :-
    range(Parameter, _), % see if range exists
    !,
    open_parm(Q, slider_ques, Parameter),
    get_message(@slider_ques, Name, Response),
   !,
   process_chosen(@slider_ques, Name, Response, Return_value, _),
   send(@slider_ques, show, false),
   display_user_response(Return_value, Parameter, Dict_key),
   !,
   add_and_check(Parameter, Return_value,
                    parm(Parameter), Value, Dict_key).


parm_atom(Parameter, slider, Q, Value) :-
   parm_atom(Parameter, a, Q, Value).


parmrange(Parameter,Minimum,Maximum) :-
                   parm(Parameter, Value),
                   Minimum=<Value,
                   Maximum>=Value.


parmset(Parameter, Set) :-
   parm(Parameter, Value),
   memberchk(Value, Set).


preprocess(Rule, Confirm_list) :-
   bagof((Rule, Conditions), clause(Rule,
                         Conditions), List),
   usingl(X),
   append(X,[Rule],X1),
   retractall(usingl(_)),
   assert(usingl(X1)),
   asserta(using(List)),
   retractall(first_time),
   assert(first_time),
   nl,write('____ Rule   '),write(Rule),nl,
   !,
   test_preconditions(List, Confirm_list).


print_how([], _) :- !.

print_how(0, _) :- !.

print_how([0], _) :- !.

print_how([p, ' '], _) :- !.

print_how([p, String], Where) :-
   blank_line(Where),
   print_string(String, 0, Where, _),
   !.

print_how([0 | [History]], Where) :-
   print_how(History, Where),!.

print_how([[0] | [History]], Where) :-
   print_how(History, Where), !.

print_how([History | [Tail]], Where) :-
   print_how(History, Where),
   print_how(Tail, Where),!.

print_how([[r, Rule, Conditions] | [History]], Where) :-
   convert_to_string(long, Rule, String),
   concat(String, ' if', String1),
   blank_line(Where),
   print_string(String1, 0, Where, _),
   process_conditions([' ', Where, 0, _], Conditions),
   print_how(History, Where),!.


print_file_header(Stream) :-
```

```
   system_name([Line1, Line2, Line3, Line4]),
   writeq(Stream, Line1), write(Stream, '.'), nl(Stream),
   writeq(Stream, Line2), write(Stream, '.'), nl(Stream),
   writeq(Stream, Line3), write(Stream, '.'), nl(Stream),
   writeq(Stream, Line4), write(Stream, '.'), nl(Stream).


print_string(' ', _,_ ,_).

print_string(String, 0, why, append) :-
   send(@explain_dict, append, String),
   send(@explain_dict, text_string,
              'no explanation available'),
   send(@explain_dict, string_ident, 't'),
   !.

print_string(String, 0, why, delete) :-
   send(@explain_dict, delete, String),!.

print_string(String, 0, how_display, _) :-
   send(@how_display_window, print, String),
   !.

print_string(String, 0, whynot_display, _) :-
   send(@whynot_wind, print, String),!.

print_string(String, 0, _, _) :-
   write(String),nl,
   !.

print_string(String, Ident, why, append) :-
   send(@explain_dict, text_string, String),
   send(@explain_dict, string_ident, Ident),
   !.

print_string(_, _, _, _) :- !.
```

```
print_whynot(Level, Where) :-
   whynot_history((Rule, Condition),Level),
   convert_to_string(long, Rule, String),
   concat(String, ' if', Out_string),
   blank_line(Where),
   print_string(Out_string, 0, Where, _),
   process_conditions(['     ', Where, 0, _], Condition),
   Next is Level + 1,
   display_whynot_executables(Next, Where),
   print_whynot(Next, Where).
print_whynot(_,_).


process_chosen(_, _, completed, [], _) :-
   !.

process_chosen(Object, _, change_user_info, Value, _) :-
   !,
   create_change_user_info,
   get_message(Object, Name_2, Response_2),
   process_chosen(Object, Name_2, Response_2, Value, _).

process_chosen(Object, _, why, Value, _) :-
   !,
   explanation(Ex),
   (Ex == yes ->
   process_why
   |
   nl,write('no explanation chosen '),nl,
   display_message(@message_window,
     'Explanation facility has turned off ...')
   ),
   get_message(Object, Name_2, Response_2),
   send(@message_window, clear),
   process_chosen(Object, Name_2, Response_2, Value, _).
process_chosen(Object, _, abort, Value, _) :-
   !,
   send(Object, show, false),
   open_abort_wind,
   process_message_1,
   send(Object, show, true),
   get_message(Object, Name_2, Response_2),
   process_chosen(Object, Name_2, Response_2, Value, _).

process_chosen(_, _, unknown, unknown, _) :-
   !.

process_chosen(@point_window, _,
        Point, [List | Value], Dim) :-
   convert_1(Point, List),
   length_of_list(List, Dim, 0),
   !,
   send(@point_window, label, ''),
   send(@point_response, append, Point),
   get_message(Object, Name, Response),
   process_chosen(Object, Name, Response, Value, Dim).

process_chosen(@point_window, _, _, Value, Dim) :-
   !,
   send(@point_window, flash),
   concat('The point must be a ', Dim, String),
   concat(String, ' dimensional point.', String1),
   send(@point_window, label, String1),
   get_message(Object, Name, Response),
   !,
   process_chosen(Object, Name, Response, Value, Dim).

process_chosen(@selection_window, _, Selection,
                              [Selection | Value], _) :-
   !,
   send(@selection_response, append, Selection),
   get_message(Object, Name, Response),
   process_chosen(Object, Name, Response, Value, _).

process_chosen(_, _, Value, Value, _).



process_conditions(_, []) :-
   !.

process_conditions(Control, Conditions) :-
   functor(Conditions, ',', _),
   arg(1, Conditions, Cond_1),
   arg(2, Conditions, Tail),
   generate_string(Cond_1, String, Ident, Control),
   display_and_save(String, Ident, Tail, Control),
   process_conditions(Control, Tail),
   !.

process_conditions(Control, Condition) :-
   generate_string(Condition, String, Ident, Control),
   display_and_save(String, Ident, [], Control).



process_explanation(Old, New) :-
   display_explanation(Old, New),
   process_message_1.


process_message_1 :-
   get(@sys, wait, Message),
   Message = message(Object, Name, Value),
   Goal =.. [Name, Object, Value],
   Goal -> true | error_2.
```

```
prop(Property) :-
   old_session_response(Property, Response,
                              Query, Dict_key),
   send(@user_log_dict, append, Dict_key),
   retractall(old_session_response(
                          Property, _, _, _)),
   assert(user_response_database(Property,
                    Response, Query, Dict_key)),
   !,
   Response = y.

prop(Property) :-
   user_response_database(Property, 'unknown', _, _),
   !,
   retractall(user_response_database(Property, _, _, _)),
   prop(Property).

prop(Property) :-
   user_response_database(Property, Value, _, _),
   !,
   Value = y.


prop(Property) :-
   send(@message_window, clear),
   question(Property, Question),
   open_prop(Question, prop_ques),
   get_message(@prop_ques, Name, Response),
   !,
   process_chosen(@prop_ques, Name,
                Response, Return_value, _),
   send(@prop_ques, show, false),
   display_user_response(Return_value,
                         Property, Dict_key),
   !,
   add_and_check(Property, Return_value,
                prop(Property), y, Dict_key).


quit_proc(_,_) :-
   write_user_file,
   kill_prowindows,
   clean.

range(num_of_spots, [0, 50]).

read_is(X, Y) :-
   send(X, show, false),
   assert(last_query(Y)),
   get_user_file,
   question_user_1,
   execute_user_is_query(Y),
   !,
   send(@user_log_window, show, false),
   open_ans_wind(yes),
   process_message_1.

read_is(_,_) :-
   send(@user_log_window, show, false),
   open_ans_wind(no),
   process_message_1.


remove_conditions_dict(Level) :-
   conds(Level, Condition_key),
   retractall(conds(Level, Condition_key)),
   send(@explain_dict, delete, Condition_key),
   remove_conditions_dict(Level).

remove_conditions_dict(_) :- !.
```

```
remove_default(Long_form, [Long_form|Rest], Rest) :-
   !.

remove_default(Long_form_1, [Long_form_2|Rest1],
                                [Long_form_2|Rest2]) :-
   remove_default(Long_form_1, Rest1, Rest2).

remove_dict_items(Dictionary) :-
   get_chain_ref(Dictionary, members, Key_list),
   \+ Key_list = [],
   send_list(Dictionary, delete, Key_list).

remove_dict_items(_).


remove_response(Property) :-
   user_response_database(Property, _, _, Dict_key),
   retractall(user_response_database(Property, _, _, _)),
   send(@user_log_dict, delete, Dict_key),
   send(@user_log_window, clear),
   send(@user_log_window, dict, @user_log_dict).


reset_previous_display(@menu_ques) :-
   !,
   send(@menu_ques, load, 'FFDES.question'),
   delete_file('FFDES.question'),
   send(@parm_brows, clear),
   send(@parm_menu_dict, free),
   send(@sys, load_from_file, 'FFDES.dictionary'),
   send(@parm_brows, dict, @parm_menu_dict),
   delete_file('FFDES.dictionary').

reset_previous_display(@selection_window) :-
   !,
   send(@selection_window, load, 'FFDES.question'),
   delete_file('FFDES.question'),
   send(@selection_input, clear),
   send(@selection_dict, free),
   send(@sys, load_from_file, 'FFDES.dictionary'),
   send(@selection_input, dict, @selection_dict),
   delete_file('FFDES.dictionary'),
   send(@selection_response, load, 'FFDES.responses'),
   delete_file('FFDES.responses').

reset_previous_display(@point_window) :-
   !,
   send(@point_window, load, 'FFDES.question'),
   delete_file('FFDES.question'),
   send(@point_response, load, 'FFDES.responses'),
   delete_file('FFDES.responses').

reset_previous_display(_).


save_conds_key(Level, Condition, 0, why) :-
   assert(conds(Level, Condition)),
   !.

save_conds_key(_, _, _, _) :- !.

save_current_display(@menu_ques) :-
   !,
   send(@menu_ques, save, 'FFDES.question'),
   send(@parm_brows, clear),
   send(@parm_menu_dict, save_in_file, 'FFDES.dictionary').

save_current_display(@selection_window) :-
   !,
   send(@selection_window, save, 'FFDES.question'),
   send(@selection_input, clear),
   send(@selection_dict, save_in_file, 'FFDES.dictionary'),
   send(@selection_response, save, 'FFDES.responses').

save_current_display(@point_window) :-
   !,
   send(@point_window, save, 'FFDES.question'),
   send(@point_response, save, 'FFDES.responses').

save_current_display(_).

save_database(Filename) :-
   display_message(@message_window, 'Saving responses...'),
   open(Filename, write, Stream),
   last_query(Query),
   retractall(last_query(_)),
   print_file_header(Stream),
   writeq(Stream, Query),
   write(Stream, '.'), nl(Stream),
   save_response_database(Stream),
   close(Stream),
   write_user_file.

save_response_database(Stream) :-
   user_response_database(Property, Value, Query, Dict_key),
   writeq(Stream, [Property, Value, Query, Dict_key]),
   write(Stream, '.'),
   nl(Stream),
   retractall(user_response_database(Property, _, _, _)),!,
   save_response_database(Stream).

save_response_database(_).

save_whynot_history(Conditions, Level) :-
   whynot_history(_,Level),
   !,
   retractall(whynot_history(_,Level)),
   assert(whynot_history(Conditions, Level)).
```

```
save_whynot_history(Conditions, Level) :-
   assert(whynot_history(Conditions, Level)).


sort_if_default(Parameter, A, [Long_form|New_list]) :-
   default(Parameter, Long_form),  % check for default
   !,
   remove_default(Long_form, A, New_list).

sort_if_default(_, A, A).

start_old_session(_,Filename) :-
   check_session_file(Filename, Stream),
   read(Stream, Query),
   load_previous_responses(Stream),
   close(Stream),
   read_is(@load_window, Query),
   !.

store_defaults(Item, Default_value) :-
   default(Item, _),
   !,
   retractall(default(Item, _)),
   assert(default(Item, Default_value)).

store_defaults(Item, Default_value) :-
   assert(default(Item, Default_value)).

store_value(Prop, Value) :-
   calculated_values(Prop, _),
   !,
   retractall(calculated_values(Prop,_)),
   assert(calculated_values(Prop, Value)).
store_value(Prop, Value) :-
   assert(calculated_values(Prop, Value)).
```

```
system_name(['FFDES June 15, 1989',
             'inference : Linda Strajnic & Tibor Toronyi',
             'knowledge base : Arunita Sarkar',
             'rule editor : Hien Bui']).


test_preconditions([],[]).

test_preconditions([Predicate | Tail], [(Rule, Condition)
                                | Condition_tail]) :-
   arg(1, Predicate, Rule),
   arg(2, Predicate, Complete_list),
   get_preconditions(Complete_list, Precondition,
                                     Weight, Condition),
   execute_precondition(Precondition, Weight),
   !,
   test_preconditions(Tail, Condition_tail).

test_preconditions([_ | Tail], Condition_list) :-
   test_preconditions(Tail, Condition_list).

unix_shell(_,_) :-
   clean,
   unix(shell).

write_out:-
  assert(functor_list([])),
  using(L),!,
  write_element(L).


write_element([L]):-
    get_functor(L),!.
write_element([H|T]):-
   get_functor(H),
   write_element(T).
write_element([]).

get_functor(L):-
   functor(L,',',_),
   arg(1,L,Rule),
   write_append(Rule),
   arg(2,L,Rule1),
   get_functor(Rule1).
get_functor(L):-
   write_append(L).

write_append(L):-
   functor(L,'!',_),!.
write_append(L):-
   functor(L,'\+',_),
   arg(1,L,Rule),
   write_append(Rule),!.
write_append(L):-
   functor(L,'==',_),!.
write_append(L):-
   functor(L,';',_),!.
write_append(L):-
   functor(L,'is',_),!.
write_append(L):-
   functor(L,'true',_),!.

write_append(Rule):-
   functor(Rule,Name,_X),
   functor_list(L),
        (Name == store_defaults ->arg(1,Rule,E),
                          append(L,[Name,E],L1)|
           (Name == user_def_val ->arg(1,Rule,E),
                           append(L,[Name,E],L1)|
           (Name == user_def_fuzzy_val-> arg(1,Rule,E),
                                append(L,[Name,E],L1)|
             (Name == store_value -> arg(1,Rule,E),
                                append(L,[Name,E],L1)|
                        append(L,[Name],L1))))),
    retractall(functor_list(_)),
    assert(functor_list(L1)).


process_why_design_user :-
    add_ques(@message_window,
          'Explanations are available by clicking the
    left mouse button on the rule you wish explained.'),
    open_why_window_design_user,
    process_callbacks(retract(why_complete)),
    send(@message_window, clear),
    send(@why_rule_window, show, false).


process_why:-
     end_user(X),
     (X == yes ->  write_out_and_keyword |
                    process_why_design_user).

insert_empty_list:-
    retractall(functor_list(_)),
    assert(functor_list([])).
write_out_and_keyword:-
                        write_out,
                        try_keyword,
                        insert_empty_list,
                        process_callbacks(retract(why_complete1)),
                        send(@message_window, clear),
                        send(@why_rule_window1, show, false).


try_keyword:-
   retractall(keyword_list(_)),
   functor_list(L),nl,write(L),nl,
  (first_functor_list ->
       retractall(first_functor_list),
       asserta(functor_list_backup(L))|true
       ),
  assert(keyword_list([])),
  user_level(Level),
  find_keyword_of_functor(L,Level),
  keyword_list(K),
  nl,write(K),
  open_why_window.

find_keyword_of_functor([explanation,V|T],Level):-
  find_keyword_of_functor([V|T],Level),!.
find_keyword_of_functor([user_def_val,V|T],Level):-
   save_user_def_val(Parameter),
   nl,write(Parameter),write(V),nl,
   list_rated_functor(Oldlist),
   (Parameter == V ->
      (Level > 1 ->
          Level1 is Level - 1
          |
          Level1 = Level
          ),
      (first_time ->
         % retractall(first_time),
          functor_list(Fl),
          nthl(1,Fl,Rule),
          write(Rule),nl,
          template(Rule,Key),
          nthl(Level,Key,Oldkey),
          write(Oldkey),nl,
          keyword_list(Kl),
          (member(Oldkey,Kl) ->
              del_element(Oldkey,Kl,Kl1),
              retractall(keyword_list(_)),
              nthl(Level1,Key,Newkey),
        append([Newkey],Kl1,Kl2),
        assert(keyword_list(Kl2)),
        nl,write(Kl1),nl,
        nl,write(Kl2),nl,
        (member(Rule,Oldlist)->
            (member(V,Oldlist) ->
                  true
                  |
                  append(Oldlist,[V,Level1],Newlist),
                  retractall(list_rated_functor(_)),
                  write(Newlist),nl,
                  assert(list_rated_functor(Newlist))
                  )
            |
            append(Oldlist,[Rule,Level1],New),
            append(New,[V,Level1],Newlist),
            retractall(list_rated_functor(_)),
            nl,write(Newlist),nl,
            assert(list_rated_functor(Newlist))
            )
        |
        append(Oldlist,[V,Level1],Newlist),
        nl,write(Newlist),nl,
        retractall(list_rated_functor(_)),
        assert(list_rated_functor(Newlist))
        )
    |
    append(Oldlist,[V,Level1],Newlist),
    nl,write(Newlist),nl,
    retractall(list_rated_functor(_)),
    assert(list_rated_functor(Newlist))
    )
|
(member(V,Oldlist) ->
    nextto(V,Level1,Oldlist)
    |
            Level1 = Level
            )
        ),
        get_list_of_keyword(user_def_val,V,Level1),
        find_keyword_of_functor(T,Level),!.
find_keyword_of_functor([user_def_fuzzy_val,V|T],Level):-
  get_list_of_keyword(user_def_fuzzy_val,V,Level),
  find_keyword_of_functor(T,Level),!.
find_keyword_of_functor([store_value,V|T],Level):-
  get_list_of_keyword(store_value,V,Level),
  find_keyword_of_functor(T,Level),!.
find_keyword_of_functor([known,V|T],Level):-
  find_keyword_of_functor([V|T],Level),!.
find_keyword_of_functor([store_defaults,V|T],Level):-
  get_list_of_keyword(store_defaults,V,Level),
  find_keyword_of_functor(T,Level),!.
find_keyword_of_functor([H|T],Level):-
  get_list_of_keyword(H,Level),
  find_keyword_of_functor(T,Level).
find_keyword_of_functor([],_).


get_list_of_keyword(user_def_val,V,Level):-
   template((user_def_val(V,_)),Key),
   which_keyword_list(Key,Level).
get_list_of_keyword(user_def_fuzzy_val,V,Level):-
   template((user_def_fuzzy_val(V,_)),Key),
   which_keyword_list(Key,Level).

get_list_of_keyword(store_value,V,Level):-
   template((store_value(V,_)),Key),
   which_keyword_list(Key,Level).
get_list_of_keyword(store_defaults,V,Level):-
   template((store_defaults(V,_)),Key),
   which_keyword_list(Key,Level).
get_list_of_keyword(H,Level):-
```

```
   template(H,Key),
   which_keyword_list(Key,Level).
get_list_of_keyword(_,_).

which_keyword_list(List,Level):-
   length(List,Length),
   (Level > Length -> last(Keyword, List)|
                      nth1(Level, List, Keyword, _)),
   which_keyword(Keyword).

which_keyword(Keywordlist):-
   quality_explanation(Number),
   length(Keywordlist,Length),
   (Number > Length -> last(Keyword, Keywordlist)|
              nth1(Number, Keywordlist, Keyword, _)),
   append_keyword(Keyword).


append_keyword(Keyword):-
   keyword_list(List),
   append(List,[Keyword],List1),
   retractall(keyword_list(_)),
   assert(keyword_list(List1)).

delete_keyword(Keyword):-
   keyword_list(List),
   del_element(Keyword,List,List1),
   retractall(keyword_list(_List)),
   assert(keyword_list(List1)).


question_user_1:-
   object(@quest1),
   send(@quest1, show, true),!.
question_user_1:-
   new(@quest1, dialog),
   send(@quest1, append, label(
                 'Did you use the system before? ')),
   send(@quest1, append, button(
                     'Yes', cascade(0, user_yes, 0))),
   send(@quest1, append, button(
                       'No', cascade(0, user_no, 0))),
   send(@quest1, open, point(510, 300)),
   process_message_1.

user_yes(X,_):-
   send(X, destroy),
   assert(user_yes),
   question_user_2.
user_no(X,_):-
   send(X, destroy),
   question_user_2.

question_user_2:-
   new(@quest2,  dialog),
   send(@quest2, size, size(450,50)),
   new(@user_name, text_item(
            'Your name:  ', '',
        cascade(@quest2, read_user_name, 0))),
   send(@user_name, width, 100),
   send(@user_name, advance, none),
   send(@quest2, open, point(510,300)),
   send(@quest2, append, @user_name),
   process_message_1.

read_user_name(X, Y) :-
   send(X, destroy),
   assert(user_name(Y)),
   nl,write(Y),nl,
   (user_yes -> retract(user_yes), if_yes | if_no).

if_no:-
```

```
   explanation_or_not,
   explanation(Ex),
   (Ex == yes ->
   (end_or_design_user,
   end_user(X),
   (X == yes ->
    question_user_6,
    question_user_3,
    question_user_4,
    question_user_5
     | nl)) | true).

question_user_3:-
   new(@quest3, dialog),
   send(@quest3, append, label('Please indicate
     your level of knowledge in designing FIR filter')),
   send(@quest3, append, label(
                  '-- from naive to expert (1 - 5)')),
   new(@quest33, browser),
   send(@quest3, above, @quest33),
   send_list(@quest33, append, [1, 2, 3, 4, 5]),
   send(@quest33, selected,
                     cascade(@quest3, selected_3, 0)),
   send(@quest33, clicked, 0),
   send(@quest3, open, point(510,300)),
   process_message_1.

selected_3(X,Range):-
   send(X, destroy),
   convert_1(Range, Number),
   assert(knowledge_level(Number)),
   assert(user_level(Number)),
   nl,  write(Number),nl.


question_user_4:-
```

```
   new(@quest4, dialog),
   send(@quest4, append, label(
         'Please select quantity of explanation')),
   send(@quest4, append, label(
              '-- from short to long (1 - 5)')),
   new(@quest44, browser),
   send(@quest4, above, @quest44),
   send_list(@quest44, append, [1, 2, 3, 4, 5]),
   send(@quest44, selected, cascade
                     (@quest4, selected_4, 0)),
   send(@quest44, clicked, 0),
   send(@quest4, open, point(510,300)),
   process_message_1.

selected_4(X,Range):-
   send(X, destroy),
   convert_1(Range, Number),
   assert(quantity_explanation(Number)),
   nl, write(Number),nl.


question_user_5:-
   new(@quest5, dialog),
   send(@quest5, append, label(
    'How would you like the quality of explanation? ')),
   send(@quest5, append, label(
          '-- from less to more technical (1 - 5) ')),
   new(@quest55, browser),
   send(@quest5, above, @quest55),
   send_list(@quest55, append, [1, 2, 3, 4, 5]),
   send(@quest55, selected,
             cascade(@quest5, selected_5, 0)),
   send(@quest55, clicked, 0),
   send(@quest5, open, point(510,300)),
   process_message_1.
```

```
selected_5(X,Range):-
   send(X, destroy),
   convert_1(Range, Number),
   assert(quality_explanation(Number)),
   nl,  write(Number),nl.

question_user_6:-
   new(@quest6, dialog),
   send(@quest6, append, label('What is your goal ? ')),
   send(@quest6, append, button(
             'Student', cascade(0, user_student, 0))),
   send(@quest6, append, button(
               'Expert', cascade(0, user_expert, 0))),
   send(@quest6, open, point(510,300)),
   process_message_1.

user_student(X,_):-
   send(X, destroy),
   assert(goal(student)),
   nl,  write(student),nl.
user_expert(X,_):-
   send(X, destroy),
   assert(goal(expert)),
   nl,  write(expert),nl.

get_user_file:-
   open('user_file', read, Stream),
   read(Stream,List),
   assert(List),
   get_content_file(Stream).

get_content_file(Stream):-
   read(Stream, Data),
   any_more(Stream, Data).

any_more(Stream,end_of_file):-
```

```
  close(Stream),!.
any_more(Stream,Data):-
  assertz(Data),
  get_content_file(Stream).


if_yes:-
   user_name(Name),
   user_list(List),
   (member(Name,List)-> del_element(Name,List,List1),
                        retractall(user_list(_List)),
                        assert(user_list(List1)),
                        retrieve_info(Name)|
                        nl,if_no).

retrieve_info(Name):-
  (user_data(Name,Kl,Qn,Ql,G,Ul,Eu,Ex) ->
                       assert(knowledge_level(Kl)),
                       assert(quantity_explanation(Qn)),
                       assert(quality_explanation(Ql)),
                       assert(goal(G)),
                       assert(user_level(Ul)),
                       assert(end_user(Eu)),
                       assert(explanation(Ex)),
    retract(user_data(Name,Kl,Qn,Ql,G,Ul,Eu,Ex)),!|
    user_data(N,K,Qn1,Ql1,G1,Ul1,Eu1,Ex1),
    retract(user_data(N,K,Qn1,Ql1,G1,Ul1,Eu1,Ex1)),!,
    assertz(user_data(N,K,Qn1,Ql1,G1,Ul1,Eu1,Ex1))).


write_user_file:-
   open('user_file', write, Stream),
   user_list(Ulist),
   user_name(Name),
   knowledge_level(Kl),
   quantity_explanation(Qn),
    quality_explanation(Ql),
    goal(G),
    user_level(Ul),
    end_user(Eu),
    explanation(Ex),
    append([Name],Ulist,Ulist1),
    write_canonical(Stream, user_list(Ulist1)),
    write(Stream, '.'),
    nl(Stream),
    write_canonical(Stream, (user_data(
           Name, Kl, Qn, Ql, G, Ul,Eu,Ex))),
    write(Stream, '.'),
    nl(Stream),
    write_back_user_file(Stream),
    close(Stream).

write_back_user_file(Stream):-
    user_data(Name, Kl, Qn, Ql, G, Ul, Eu,Ex),
    write_canonical(Stream, (user_data(
             Name, Kl, Qn, Ql, G, Ul, Eu,Ex))),
    write(Stream, '.'),
    nl(Stream),
    retract(user_data(Name, Kl, Qn,
                    Ql, G, Ul, Eu,Ex)),!,
    write_back_user_file(Stream),!.
write_back_user_file(_Stream).


reduce_knowledge_level:-
    knowledge_level(Kl),
    (Kl > 1 -> (Kl1 is Kl - 1,
               retractall(knowledge_level(_)),
               assert(knowledge_level(Kl1)))|true).

template(calculate_stopband_edge,[
        ['transition width'],
        ['scale'],
        ['stopband edge']]).

template(call_bartlett,[
        ['triangular'],
        ['bartlett']]).


change_abort_wind(Wind, y) :-
   !,
   send(Wind, show, false),
   send(@abort_ques, clear),
   send(@abort_ques, print,
   'Do you wish to save the questions and answers'),
   send(@abort_ques, print, '(for the current query)
                                     processed so far?'),
   send(@yes_cascade, name, enter_filename),
   send(@no_cascade, name, enter_filename),
   send(Wind, show, true),
   process_message_1.

change_abort_wind(Wind, n) :-
   send(Wind, show, false).

change_cursor(Window, Type) :-
   get(Window, cursor, Cursor),
   assert(old_cursor(Window, Cursor)),
   cursor_types(Type, New_cursor),
   send(Window, cursor, New_cursor),!.
change_cursor(_,_).

create_ans_wind(Y) :-
   new(@Y, dialog('Answer to IS query')),
   send(@Y, size, size(500, 300)),
   send(@Y, append, label('Answer is:', none)),
   send(@Y, append, label(' ',none)),
   send(@Y, append, label(' ', none)),
   send(@Y, append, label(' ', none)),
   send(@Y, append, label(Y, none)),
   send(@Y, append, label(' ', none)),
   send(@Y, append, label(' ', none)),
   send(@Y, append, button('CONTINUE',
          cascade(0, open_and_process, main))).

create_atom_wind(Q, Parameter) :-
   new(@atom_ques, browser),
   send(@atom_ques, size, size(500,80)),
   new(@atom_response, dialog),
   new(@atom_text, text_item('Ans: ', '',
                 cascade(@atom_ques, 0, 0))),
   check_default(Parameter, @atom_text),
   send(@atom_text, advance, none),
   send(@atom_response, append, @atom_text),
   send(@atom_response, append, label(' ', none)),
   send(@atom_response, append, button('UNKNOWN',
                 cascade(@atom_ques, 0, unknown))),
   send(@atom_response, append, button('WHY',
              cascade(@atom_ques, 0, why))),
   send(@atom_response, append, button(
       'ABORT',cascade(@atom_ques,0,abort))),
   send(@atom_response, append, button(
                     'CHANGE USER INFO',
    cascade(@atom_ques, 0,change_user_info))),
   send(@atom_ques, above, @atom_response),
   send(@atom_ques, open, point(510, 300)),
   add_ques(@atom_ques, Q).


create_cursors :-
   new(@hour_map, bitmap(16,16)),
   send(@hour_map, load, 'hglass.cursor'),
   new(@hour_glass, cursor(@hour_map)),
   new(@no_entry_map, bitmap(16,16)),
   send(@no_entry_map, load, 'do_not_enter'),
   new(@no_entry_cursor, cursor(@no_entry_map)),
   new(@stop_map, bitmap(16,16)),
   send(@stop_map, load, 'stop.cursor'),
   new(@stop_cursor, cursor(@stop_map)).


create_error_window(String) :-
   object(@error_window),
   !,
   add_ques(@error_window, String),
   send(@error_window, show, true).

create_error_window(String) :-
   new(@error_window, browser),
   send(@error_window, size, size(500, 80)),
   new(@error_return, dialog),
   send(@error_return, append, button('Continue',
                cascade(0, error_complete,0))),
   send(@error_window, above, @error_return),
   send(@error_window, open, point(445, 275)),
   add_ques(@error_window, String).


create_filename_wind :-
   new(@filename_wind, dialog),
   send(@filename_wind, size, size(500,70)),
   send(@filename_wind, append, text_item(
                'Enter name of file (IN QUOTES):',
               '', cascade(@filename_wind,0,0))).

create_how :-
   object(@how_display_window),
   send(@how_display_window, clear),
   send(@how_display_window, show, true),
   !.

create_how :-
   new(@how_display_window, browser
                  ('3.1.1 HOW Explanation')),
   send(@how_display_window, size, size(500,546)),
   new(@how_return, dialog('HOW Explanation')),
   new(@how_button, button('QUIT',
     cascade(@how_display_window, open_and_process, main))),
   send(@how_return, append, @how_button),
   send(@how_display_window, above, @how_return),
   send(@how_display_window, open, point(510,300)).

create_is_wind :-
   retractall(usingl(_)),
   retractall(using(_)),
   assert(usingl([])),
   assert(first_functor_list),
   retractall(list_rated_functor(_)),
   assert(list_rated_functor([])),
   new_dialog(@is_query, '1.0 Query', is_wind_items),
   send(@is_query, size, size(500,150)),
   new(@enter_is, text_item('IS: ', '',
        cascade(@is_query, read_is, 0))),
   send(@enter_is, width, 100),
   send(@enter_is, advance, none),
   send(@is_query, append, @enter_is).

is_wind_items(_, button('PREVIOUS SESSION',
             cascade(0, open_and_process, load)),
    _, []).
is_wind_items(_, button('QUIT',
            cascade(0, open_and_process, main)),
            right,[]).
is_wind_items(_, label(' ', none), below,[]).
is_wind_items(_, label(
  'Enter the query in brackets', none), below, []).


create_load_window(Label) :-
   object(@load_window),
   !,
   send(@load_window, label, Label),
   send(@load_window, show, true).

create_load_window(Label) :-
   new_dialog(@load_window, Label, load_items),
   send(@load_window, size, size(500,70)),
   send(@load_window, open, point(510,300)).

load_items(@old_filename, text_item(
        'Enter filename of previous session: ', '',
       cascade(0, start_old_session, 0)), _, []).
load_items(_, label(' ',none), _, []).
load_items(_, button('QUIT', cascade(
               0, open_and_process, main)), _, []).

create_log_window :-
   object(@user_log_window),
   !,
   send(@user_log_window, show, true).

create_log_window :-
   new(@user_log_window, browser('Previous Responses ')),
   send(@user_log_window, size, size(500,150)),
   send(@user_log_window, open, point(510, 0)),
   create_log_window_icon,
   send(@user_log_window, clicked,
              cascade(0, view_response, 0)),
   send(@user_log_window, selected,
            cascade(0, response_change, 0)),
   new(@user_log_dict, dict),
   send(@user_log_window, dict, @user_log_dict).

create_log_window_icon :-
   file_exists('response.icon'),
   new(@response_bitmap, bitmap(64, 64)),
   send(@response_bitmap, load, 'response.icon'),
   send(@user_log_window, icon, @response_bitmap).
create_log_window_icon.


create_menu_wind(Q) :-
   new(@menu_ques, browser),
   send(@menu_ques, size, size(500, 80)),
   send(@parm_brows, selected,
            cascade(@menu_ques, 0, 0)),
   send(@parm_brows, below, @menu_ques),
   new(@menu_response, dialog),
   send(@menu_response, size, size(500, 95)),
   send(@menu_response, append, button(
        'UNKNOWN',cascade(@menu_ques,0, unknown))),
   send(@menu_response, append, button('WHY',
               cascade(@menu_ques, 0, why))),
   send(@menu_response, append, button('ABORT',
                  cascade(@menu_ques,0,abort))),
   send(@menu_response, append, button(
                       'CHANGE USER INFO',
        cascade(@menu_ques, 0,change_user_info))),
   send(@menu_response, below, @parm_brows),
   send(@menu_ques, open, point(510, 300)),
   add_ques(@menu_ques, Q).


create_message_window_icon :-
   file_exists('message.icon'),
   new(@message_bitmap, bitmap(64, 64)),
   send(@message_bitmap, load, 'message.icon'),
   send(@message_window, icon, @message_bitmap).
create_message_window_icon.

create_point_window(Q) :-
   new(@point_window, browser),
   send(@point_window, size, size(500,80)),
   new(@point_input, dialog),
   new(@input_x, text_item('Point: ', '',
              cascade(@point_window, 0,0))),
   send(@input_x, advance, clear),
   send(@point_input, append,
        label('Enter each point or set of', none)),
   send(@point_input, append,
        label('points separately in the', none)),
   send(@point_input, append,
        label('form :', none)),
   send(@point_input, append,
        label(' (X, X1, X2, ...)', none)),
   send(@point_input, append, label(' ', none)),
   send(@point_input, append, @input_x),
   send(@point_input, below, @point_window),
   new(@point_response, browser),
   send(@point_response, right, @point_input),
   new(@point_options, dialog),
   send(@point_options, append, button('COMPLETED',
           cascade(@point_window, 0, completed))),
   send(@point_options, append, button('UNKNOWN',
           cascade(@point_window, 0, unknown))),
  send(@point_options, append, button('WHY',
           cascade(@point_window, 0, why))),
   send(@point_options, append, button('ABORT',
          cascade(@point_window, 0, abort))),
   send(@point_options, append, button('CHANGE USER INFO',
         cascade(@point_window, 0,change_user_info))),
   send(@point_options, below, @point_response),
   send(@point_window, open, point(510,300)),
   add_ques(@point_window, Q).

create_prop_wind(Q) :-
   new(@prop_ques, browser),
   send(@prop_ques, size, size(500, 80)),
   new(@prop_response, dialog),
   send(@prop_response, append,
     button('YES', cascade(@prop_ques, 0, y))),
   send(@prop_response, append,
     button('NO', cascade(@prop_ques, 0, n))),
   send(@prop_response, append, label(' ',none)),
   send(@prop_response, append,
     button('UNKNOWN',cascade(@prop_ques, 0, unknown))),
   send(@prop_response, append, button('WHY',
        cascade(@prop_ques, 0, why))),
   send(@prop_response, append, button('ABORT',
                  cascade(@prop_ques,0,abort))),
   send(@prop_response, append, button(
         'CHANGE USER INFO',
        cascade(@prop_ques, 0,change_user_info))),
   send(@prop_ques, above, @prop_response),
   send(@prop_ques, open, point(510, 300)),
   add_ques(@prop_ques, Q).


create_selection_window(Q, A) :-
   new(@selection_dict, dict),
   new(@selection_window, browser),
   send(@selection_window, size, size(500,80)),
   new(@selection_input, browser),
   send(@selection_input, dict, @selection_dict),
   send(@selection_input, clicked,
          cascade(@selection_window, 0, 0)),
   send(@selection_input, selected,
          cascade(@selection_window, 0, 0)),
   send(@selection_input, below, @selection_window),
   new(@selection_response, browser),
   send(@selection_response, right, @selection_input),
   new(@selection_options, dialog),
   send(@selection_options, append, button('COMPLETED',
         cascade(@selection_window, 0, completed))),
   send(@selection_options, append, button('UNKNOWN',
            cascade(@selection_window, 0, unknown))),
   send(@selection_options, append, button('WHY',
           cascade(@selection_window, 0, why))),
   send(@selection_options, append, button('ABORT',
           cascade(@selection_window, 0, abort))),
   send(@selection_options, append, button('CHANGE USER INFO',
             cascade(@selection_window,0,change_user_info))),
   send(@selection_options, below, @selection_response),
   add_ans(@selection_input, @selection_dict, A),
   send(@selection_window, open, point(510,300)),
   add_ques(@selection_window, Q).


create_slider_wind(Ques, [Min, Max], Parameter) :-
   new(@slider_ques, browser),
   send(@slider_ques, size, size(500,80)),
   new(@slider_response, dialog),
   new(@slider, slider('Choose', Min, Max, Min,
                cascade(@slider_ques, 0, 0))),
   check_default(Parameter, @slider),
   send(@slider_response, append, @slider),
   send(@slider_response, append, label(' ', none)),
   send(@slider_response, append, button('UNKNOWN',
       cascade(@slider_ques, 0, unknown))),
  send(@slider_response, append, button('WHY',
                 cascade(@slider_ques,0,why))),
   send(@slider_response, append, button('ABORT',
              cascade(@slider_ques, 0, abort))),
   send(@slider_response, append, button('CHANGE USER INFO',
                cascade(@slider_ques, 0,change_user_info))),
    send(@slider_ques, above, @slider_response),
    send(@slider_ques, open, point(510, 300)),
    add_ques(@slider_ques, Ques).

create_whynot :-
    object(@whynot_wind),
    !,
    send(@whynot_wind, clear),
    send(@whynot_wind, show,true).

create_whynot :-
    new(@whynot_wind, browser('3.2.1 WHYNOT Explanation')),
    send(@whynot_wind, size, size(500,546)),
    new(@whynot_return, dialog('WHYNOT Explanation')),
    new(@whynot_button, button('RETURN',
               cascade(0,open_and_process,main))),
    send(@whynot_return, append, @whynot_button),
    send(@whynot_wind, above, @whynot_return),
    send(@whynot_wind, open, point(510,300)).
    which cursor object a given type references. */

cursor_types(hour, @hour_glass).
cursor_types(stop, @stop_cursor).
cursor_types(no_entry, @no_entry_cursor).



display_explanation(_, how) :-
    \+ clause(history(_),_),
    !.
display_explanation(_,whynot) :-
    \+ clause(whynot_history(_,_),_),
    !.

display_explanation(Old, New) :-
```

```
   send(Old, show, false),
   open_display_explanation(New).


display_how(_, _) :-
   send(@query_display, show, false),
   display_message(@message_window,
        'Displaying HOW...please wait'),
   create_how,
   send(@how_button, show, false),
   history(History),
   print_how(History, how_display),
   send(@how_display_window, line_no, 0),
   send(@how_button, show, true),
   process_message_1.

display_message(Window, String) :-
   send(Window, clear),
   display_message_strings(Window, [String]),
   send(Window, line_no, 0).

display_message_strings(_, []) :- !.

display_message_strings(Window, [Line | Tail]) :-
   send(Window, append, Line),
   display_message_strings(Window, Tail).

display_why_explanation(_, _) :-
   get(@explain_dict, current, _),
   get(@explain_dict, text_string, String),
   get(@explain_dict, string_ident, Identifier),
   determine_explain_type(String, Identifier).


display_whynot(_,_) :-
   send(@query_display, show, false),
   display_message(@message_window,
  'Displaying WHYNOT...please wait'),
   create_whynot,
   send(@whynot_button, show, false),
   print_whynot(0, whynot_display),
   send(@whynot_wind, line_no, 0),
   send(@whynot_button, show, true),
   send(@message_window, clear),
   process_message_1.

display_whynot_executables(Level, Where) :-
   clause(executable(Level, Rules),_),
   convert_to_string(long, Rules, String),
   concat(String, ' has failed', Outstring),
   blank_line(Where),
   print_string(Outstring, 0, Where, _).
display_whynot_executables(_, _).


enter_filename(Object, y) :-
   !,
   send(Object, show, false),
   create_filename_wind,
   send(@filename_wind, open, point(510,300)),
   get_message(_, _, Value),
   send(@filename_wind, show, false),
   convert_1(Value, Converted_value),
   save_database(Converted_value),
   kill_prowindows,
   retractall(user_response_database(_, _, _, _)),
   abort.

enter_filename(Object, n) :-
   send(Object, show, false),
   kill_prowindows,
   abort.
```

```
error_complete(_, _) :-
   assert(error_complete_flag).

error_message(Property, String) :-
   create_error_window(String),
   send(@error_window, flash),
   (calculated_values(Property, _) ->
       retractall(calculated_values(Property,_))
    | remove_response(Property)),
   process_callbacks(retractall(error_complete_flag)),
   send(@error_window, show, false).


execute_user_is_query(Y) :-
   convert_1(Y, Z),
   confirmed(Z, History, 0),
   !,
   retractall(history(_)),
   assert(history(History)).

execute_user_is_query(Y) :-
   change_in_database,
   !,
   retractall(change_in_database),
   retractall(builtin(_)),
   retractall(conds(_,_)),
   retractall(rules(_,_)),
   get_chain_ref(@explain_dict, members, Entries),
   send_list(@explain_dict, delete, Entries),
   execute_user_is_query(Y).

how_file(_, Filename) :-
   display_message(@message_window,
  'Writing HOW explanation to file Please wait'),
   tell(Filename),
   history(History),
   print_how(History, Filename),
   told,
   open_and_process(@query_display, main).

main_dialogs :-
   new(@message_window, browser('FFDES - message window')),
   send(@message_window, size, size(500,60)),
   send(@message_window, open, point(510, 217)),
   display_message(@message_window, 'Please wait...'),
   create_message_window_icon,
   new_dialog(@main, 'Main functions', main_dialog_items),
   send(@main, size, size(520,70)),
   send(@main, open, point(510,300)),
   new(@parm_menu_dict, dict),
   new(@parm_brows, browser),
   new(@explain_sheet, sheet),
   send(@explain_sheet, append, [text_string, string_ident]),
   new(@explain_dict, dict),
   send(@explain_dict, sheet, @explain_sheet),
   add_ques(@message_window,
      'A selection can be made by clicking the
    left button of the mouse on the option you wish.'),
   process_message_1.

main_dialog_items(_, button('QUERY',
     cascade(@main, open_and_process, is_query)), _, []).
main_dialog_items(_, button('EDITOR',
      cascade(@main, open_and_process, editor_menu)),
     right,[]).
main_dialog_items(_, button('EXPLANATIONS',
     cascade(@main, open_and_process,
            explain_menu)), right, []).
main_dialog_items(_, button('SYSTEM',
    cascade(@main, open_and_process, system_menu)),
     right, []).
main_dialog_items(_, button('QUIT', cascade
        (@main, quit_proc, 0)), right, []).

create_system_menu :-
   new_dialog(@system_menu, '4.0 System functions',
                               system_items),
   send(@system_menu, size, size(500,150)),
   send(@system_menu, open, point(510,300)).

system_items(_, button('UNIX',
       cascade(0, unix_shell, 0)), _, []).
system_items(_, button('SDA',
       cascade(0, call_sda, 0)), right, []).
system_items(_, button('CONSULT',
     cascade(0, reconsult_kbase, 0)), right, []).
system_items(_, button('RETURN',
  cascade(0, open_and_process, main)), right, []).

create_explanation_menu :-
   new_dialog(@explain_menu,
      '3.0 Query explanations', explain_items),
   send(@explain_menu, size, size(500,150)),
   send(@explain_menu, open, point(510, 300)).

explain_items(_, button('HOW',
     cascade(0, process_explanation, how)), _, []).
explain_items(_, button('WHYNOT',
        cascade(0, process_explanation, whynot)),
                              right, []).
explain_items(_, button('RETURN',
    cascade(0, open_and_process, main)), right, []).

create_editor_menu :-
   new_dialog(@editor_menu,
         '2.0 Edit functions', editor_items),
   send(@editor_menu, size, size(500,150)),
   send(@editor_menu, open, point(510, 300)).
```

```
editor_items(_, button('ADD', none), _, []).
editor_items(_, button('DELETE', none), right, []).
editor_items(_, button('EDIT', none), right, []).
editor_items(_, button('RETURN',
     cascade(0, open_and_process, main)), right, []).

reconsult_kbase(_,_) :-
   send(@system_menu, show, false),
   create_consult_wind,
   get(@sys, wait, Message),
   get_specific_message(Message, reconsult),
   open_and_process(@consult_wind, main).

get_specific_message(message(_, ques_done,_),
                 ques_done) :- !.
get_specific_message(message(_,
                open_and_process, _), _) :- !.
get_specific_message(message(_,
                   cycle_check, _), Looking) :-
   get(@kbase_consult, selection, Kbase),
   send(@kbase_file, show, Kbase),
   get(@inf_consult, selection, Inf),
   send(@inference_file, show, Inf),
   get(@sys, wait, Message),
   get_specific_message(Message, Looking).
get_specific_message(message(From,
                   reconsult, _), reconsult) :-
   !,
   reconsult,
   send(@message_window, clear).
get_specific_message(_, Looking_for) :-
   get(@sys, wait, Message),
   get_specific_message(Message, Looking_for).

create_consult_wind :-
   object(@consult_wind),
   !,
   send(@consult_wind, show, true).

create_consult_wind :-
   new_dialog(@consult_wind, '4.4 Consult knowledge
                  base or inference', consult_items),
   send(@consult_wind, size, size(500,150)),
   send(@consult_wind, open, point(510,300)).

consult_items(_, label('Press CONSULT when done,
                 RETURN to flip between options.',
      none), _,[]).
consult_items(_, label(' ', none),_,[]).
consult_items(@kbase_consult, menu('Consult
       knowledgebase: ', cycle, cycle_check), _,
      [append: [true,false]]).
consult_items(@kbase_file, text_item('FILENAME:',
                  'knowledge',none), right, []).
consult_items(@inf_consult, menu('Consult
           inference:     ', cycle, cycle_check),
      below, [append: [true,false]]).
consult_items(@inference_file, text_item('FILENAME:',
                  'inference', none), right, []).
consult_items(_, label(' ', none), below,[]).
consult_items(_, button('CONSULT',
       cascade(@consult_wind,reconsult,0)), below, []).
consult_items(_, button('QUIT',cascade(@consult_wind,
                            open_and_process, main)),
      right,[]).

reconsult :-
   display_message(@message_window, 'Consulting
                            files...please wait.'),
   get(@kbase_file, selection, Kinput),
   get(@inference_file, selection, Inf_input),
   convert_to_string(Kinput, Kbase),
   convert_to_string(Inf_input, Inference),
   ( get(@kbase_consult, selection, true),
                file_exists(Kbase) ->
        consult(Kbase)
   ; get(@kbase_consult, selection, true) ->
        write(Kbase), write(' does not exist'),nl
   ; true),
   ( get(@inf_consult, selection, true),
                  file_exists(Inference) ->
        consult(Inference)
   ; get(@inf_consult, selection, true) ->
        write(Inference), write(' does not exist'),nl
   ; true).


open_abort_wind :-
   object(@abort_ques),
   !,
   send(@abort_ques, show, true).

open_abort_wind :-
   new(@abort_ques, browser('Abort')),
   send(@abort_ques, size, size(500, 70)),
   new(@abort_response, dialog),
   new(@yes_cascade, cascade(@abort_ques,
                  change_abort_wind, y)),
   new(@abort_yes, button('YES', @yes_cascade)),
   send(@abort_response, append, @abort_yes),
   new(@no_cascade, cascade(@abort_ques,
                     change_abort_wind, n)),
   new(@abort_no, button('NO', @no_cascade)),
   send(@abort_response, append, @abort_no),
   send(@abort_ques, above, @abort_response),
   send(@abort_ques, open, point(510,300)),
   send(@abort_ques, print, 'Are you sure
             that you want to abort???'),
   send(@abort_ques, print, 'Press the left
                    button to confirm or the'),
   send(@abort_ques, print, 'middle button to cancel.').


open_and_process(X, Y) :-
   open_proc(X, Y),
   process_message_1.

open_ans_wind(Y) :-
   object(@Y),
   !,
   send(@Y, show, true).

open_ans_wind(yes) :-
   create_ans_wind(yes),
   send(@yes, open, point(510, 300)).

open_ans_wind(no) :-
   create_ans_wind(no),
   send(@no, open, point(510, 300)).

open_display_explanation(New) :-
   object(@query_display),
   !,
   (New = how ->
        send(@query_display, label,
                     '3.1 How explanation'),
        send(@filename, message,
                     cascade(0, how_file,0)),
        send(@query_button, message,
                 cascade(0, display_how, 0))
    ; otherwise ->
        send(@query_display, label,
                      '3.2 Whynot explanation'),
        send(@filename, message, cascade(0,
                              whynot_file, 0)),
           send(@query_button, message, cascade(0,
                               display_whynot, 0))),
      send(@query_display, show, true).

  open_display_explanation(New) :-
      new(@query_display, dialog('3.1 How explanation')),
      send(@query_display, size, size(500, 200)),
      send(@query_display, append, label(' ',none)),
      send(@query_display, append,
           label('Enter filename, (no quotes or
                    periods allowed),', none)),
      send(@query_display, append,
           label('  or click DISPLAY for
                        screen view.', none)),
      send(@query_display, append, label(' ',none)),
      new(@filename, text_item('File : ', '',
                        cascade(0, how_file,0))),
      send(@filename, width, 20),
      send(@filename, advance, none),
      send(@query_display, append, @filename),
      send(@query_display, append, label(' ', none)),
      new(@query_button, button('DISPLAY',
                       cascade(0, display_how, 0))),
      send(@query_display, append, @query_button),
      send(@query_display, append, button('QUIT',
               cascade(0, open_and_process, main))),
      send(@query_display, open, point(510,300)),
      (New = whynot ->
           send(@query_display, label,
                  '3.2 Whynot explanation'),
           send(@filename, message,
                  cascade(0, whynot_file, 0)),
           send(@query_button, message,
                 cascade(0, display_whynot, 0))
        ; otherwise -> true).
```

```
open_parm(Ques, slider_ques, Parameter) :-
   object(@slider_ques),
   !,
   add_ques(@slider_ques, Ques),
   range(Parameter, [Min, Max]),
   send(@slider, high, Max),
   send(@slider, low, Min),
   send(@slider, selection,Min),
   check_default(Parameter, @slider),
   send(@slider_ques, show,true).

open_parm(Ques, atom_ques, Parameter) :-
   object(@atom_ques),
   !,
   check_default(Parameter,@atom_text),
   add_ques(@atom_ques, Ques),
   send(@atom_ques, show, true).

open_parm(Ques, Y, _) :-
   object(@Y),
   !,
   add_ques(@Y, Ques),
   send(@Y, show, true).

open_parm(Ques, menu_ques, _) :-
   !,
   create_menu_wind(Ques).
open_parm(Ques, atom_ques, Parameter) :-
   !,
   create_atom_wind(Ques, Parameter).
open_parm(Ques, slider_ques, Parameter) :-
   !,
   range(Parameter, [Min, Max]),
   create_slider_wind(Ques, [Min, Max], Parameter).

open_point_window(Q) :-
```

```
   object(@point_window),
   !,
   add_ques(@point_window, Q),
   send(@point_response, clear),
   send(@point_window, show, true).

open_point_window(Q) :-
   create_point_window(Q).

open_proc(X, Y) :-
   object(@Y),
   !,
   remove_user_log_window(Y),
   send(X, show, false),
   send(@Y, show, true).

open_proc(X, explain_menu) :-
   send(X, show, false),
   create_explanation_menu.

open_proc(X, system_menu) :-
   send(X, show, false),
   create_system_menu.

open_proc(X, load) :-
   send(X, show, false),
   create_load_window('4.1 Old session filename').

open_proc(X, is_query) :-
   send(X, show, false),
   create_log_window,
   create_is_wind,
   send(@is_query, open, point(510, 300)).

open_proc(X, editor_menu) :-
   send(X, show, false),
   create_editor_menu.


open_prop(Ques, Y) :-
   object(@Y),
   !,
   add_ques(@Y, Ques),
   send(@Y, show, true).

open_prop(Ques, prop_ques) :-
   create_prop_wind(Ques).

open_selection_window(Q, A) :-
   object(@selection_window),
   !,
   add_ques(@selection_window, Q),
   add_ans(@selection_input, @selection_dict, A),
   send(@selection_window, show, true).

open_selection_window(Q, A) :-
   create_selection_window(Q, A).


open_view_ques_ans :-
   object(@ques_ans_window),
   !,
   send(@ques_ans_window, clear),
   send(@ques_ans_window, show, true).

open_view_ques_ans :-
   new(@ques_ans_window, browser('')),
   send(@ques_ans_window, size, size(500,150)),
   new(@ques_ans_done, dialog('')),
   send(@ques_ans_done, append, button('DONE',
                    cascade(0,ques_done,0))),
   send(@ques_ans_window, above, @ques_ans_done),
   send(@ques_ans_window, cursor, @stop_cursor),
   send(@ques_ans_window, open, point(510,200)).


open_why_window_design_user :-
   object(@why_rule_window),!,
   redisplay_why_dict_design_user,
   send(@why_rule_window, show, true).


open_why_window_design_user :-
   new(@why_rule_window,
         browser('Explanation facility')),
   new(@why_complete, dialog('Return window')),
   new(@why_button, button('DONE',
                 cascade(0, why_done, 0))),
   send(@why_complete, append, @why_button),
   new(@why_text, browser('Explanation facility')),
   send(@why_text,size, size(400,100)),
   send(@why_text, left, @why_complete),
   send(@why_rule_window, size, size(500,300)),
   send(@why_rule_window, above, @why_complete),
   send(@why_rule_window, open, point(0,217)),
   send(@why_rule_window, selected,
        cascade(0, display_why_explanation, 0)),
   send(@why_rule_window, clicked,
      cascade(0, display_why_explanation, 0)),
   redisplay_why_dict_design_user.

redisplay_why_dict_design_user:-
   send(@why_button, show, false),
   send(@why_rule_window, clear),
   send(@why_text, clear),
   send(@why_rule_window, dict, @explain_dict),
   send(@why_rule_window, line_no, 0),
   send(@why_button, show, true).



open_why_window:-
   object(@why_rule_window1),!,
   send(@why_rule_window1, destroy),
    open_why_window.

open_why_window:-
   new(@why_rule_window1,
          browser('Explanation facility')),
   new(@why_complete1, dialog('Return window')),
   new(@why_button11, button('DONE',
                 cascade(0, why_done1, 0))),
   send(@why_complete1, append, @why_button11),
   user_level(Level),
   (Level =:= 5 ->
     (new(@why_button1, button('UPDATE KEYWORD',
                  cascade(0, enter_keyword, 0))),
      new(@why_button2, slider('LESS TECHNICAL:',1,5,
          Level, cascade(0,more_less_technical, 0))),
      send(@why_complete1, append, @why_button2))|
      (Level =:= 1 ->
      new(@why_button1, slider('MORE TECHNICAL:',1,5,
           Level, cascade(0, more_less_technical, 0)))|
     new(@why_button1, slider('MORE-LESS TECHNICAL:',1,5,
        Level, cascade(0, more_less_technical, 0))))),
   send(@why_complete1, append, @why_button1),
   send(@why_rule_window1, size, size(500,300)),
   send(@why_rule_window1, above, @why_complete1),
   send(@why_rule_window1, open, point(0,217)),
   redisplay_why_dict.
```

```
redisplay_why_dict:-
   keyword_list(List),
   send(@why_button11, show, false),
   send(@why_rule_window1, clear),
   send(@why_rule_window1, append,List),
   send(@why_button11, show, true).

remove_user_log_window(main) :-
   object(@user_log_window),
   send(@user_log_window, show, false),
   !.

remove_user_log_window(query_display) :-
   send(@user_log_window, show, false),
   !.

remove_user_log_window(_).

reset_cursor(Window) :-
   old_cursor(Window, Cursor),
   retractall(old_cursor(Window,_)),
   send(Window, cursor, Cursor),!.
reset_cursor(_).

more_less_technical(_,V):-
  nl,write(V),nl.
enter_keyword(_,_):-
  open_keyword_option.

open_keyword_option:-
  object(@keyword_option),!,
  send(@keyword_option, show, true).
open_keyword_option:-
  new(@keyword_option, dialog('Update keyword')),
  send(@keyword_option, append, button(
  'ADD KEYWORD',cascade(0,add_keyword,0))),
  send(@keyword_option, append, button('DELETE KEYWORD',
                           cascade(0,delete_keyword,0))),
  send(@keyword_option, append, button('REPLACE KEYWORD',
                           cascade(0,replace_keyword,0))),
  send(@keyword_option, append, button('ASSIGN WEIGHT KEYWORD',
                    cascade(0,assign_weight_keyword,0))),
  send(@keyword_option, append, button('DONE',
    cascade(0,done_key_option,0))),
  send(@keyword_option, open, point(0,70)),
  process_message_1.

add_keyword(X,_):-
  assert(add_keyword),
  open_enter_keyword('Please enter your own keyword').
delete_keyword(X,_):-
  assert(delete_keyword),
  open_enter_keyword('Please enter keyword
             which you want to delete').
replace_keyword(X,_):-
  assert(replace_keyword),
  open_enter_keyword('Please enter keyword
                 which you want to replace').
assign_weight_keyword(X,_):-
   nl, write(' assign weight to keyword  '), nl.
done_key_option(X,_):-
  send(X, show, false).


open_enter_keyword(M):-
  new(@enter_keyword, dialog(M)),
  send(@enter_keyword, size, size(500,50)),
  (replace_keyword -> retractall(replace_keyword),
  new(@user_keyword1, text_item('Old keyword:  ',
  '', cascade(@enter_keyword, delete_old_keyword, 0))),
   new(@user_keyword, text_item('New keyword:  ',
   '', cascade(@enter_keyword, add_new_keyword, 0))),
          send(@user_keyword1, width, 100),
          send(@user_keyword1, advance, true),
          send(@enter_keyword, append, @user_keyword1)
   |new(@user_keyword, text_item('Keyword:  ','',
       cascade(@enter_keyword, get_user_keyword, 0)))),
   send(@user_keyword, width, 100),
   send(@user_keyword, advance, none),
   send(@enter_keyword, open, point(0,620)),
   send(@enter_keyword, append, @user_keyword),
   process_message_1.

get_user_keyword(X,Y):-
   send(X,destroy),
   (add_keyword -> retractall(add_keyword),
         append_keyword(Y)|
       (delete_keyword -> retractall(delete_keyword),
        delete_keyword(Y)| true)),
   open_why_window.
delete_old_keyword(_X,Y):-
   delete_keyword(Y),
   process_message_1.

add_new_keyword(X,Y):-
    send(X, destroy),
    append_keyword(Y),
    open_why_window.



why_done(_, _) :-
    assert(why_complete).
why_done1(_, _) :-
    (object(@keyword_option)-> send(@keyword_option,
                                 show, false)|true),
    (object(@enter_keyword) -> send(@enter_keyword,
                                        destroy)|true),
   assert(why_complete1).

why_file_complete(_, _) :-
   send(@why_file_return, show, false),
   send(@why_file_window, show, false),
   send(@why_rule_window, show, true),
   assert(done_view).

why_file_read(end_of_file) :- !.

why_file_read(String) :-
   send(@why_file_window, print, String),
   read(In_string),
   why_file_read(In_string).

why_file_view :-
   object(@why_file_window),
   send(@why_file_window, clear),
   send(@why_file_window, show, true),
   send(@why_file_return, show, true),
   !.

why_file_view :-
   new(@why_file_window, browser('RULE EXPLANATION')),
   new(@why_file_return, dialog('')),
   send(@why_file_return, append,
                button('RETURN', cascade(0,
                     why_file_complete, 0))),
   send(@why_file_window, size, size(500,600)),
   send(@why_file_window, above, @why_file_return),
   send(@why_file_window, open, point(510,300)).

whynot_file(_, Filename) :-
   display_message(@message_window, 'Writing WHYNOT
                  explanation to file Please wait'),
   tell(Filename),
   print_whynot(0,Filename),
   told,
   open_and_process(@query_display, main).

explanation_or_not:-
   new(@explain_or_not, dialog),
   send(@explain_or_not, append, label('Do you like
            the system provides explanations ? ')),
   send(@explain_or_not, append, button('YES',
                         cascade(0, yes_explain, 0))),
   send(@explain_or_not, append, button('NO',
                          cascade(0, no_explain, 0))),
   send(@explain_or_not, open, point(510,300)),
   process_message_1.

yes_explain(X,_Y):-
  send(X,destroy),
  assert(explanation(yes)).
no_explain(X,_):-
  assert(explanation(no)),
  send(X,destroy).

end_or_design_user:-
   new(@end_or_design_user, dialog),
   send(@end_or_design_user, append, label
     ('Explanations for end-user or design-user? ')),
   send(@end_or_design_user, append,
        button('End User',cascade(0, end_user, 0))),
   send(@end_or_design_user, append,
        button('Design User',cascade(0,design_user, 0))),
   send(@end_or_design_user, open, point(510,300)),
   process_message_1.

end_user(X,_):-
  send(X,destroy),
  assert(end_user(yes)).
```

```
design_user(X,_):-
  send(X,destroy),
  assert(end_user(no)).



create_change_user_info:-
  new(@change_info, dialog('Changing
                        user information')),
  send(@change_info, append,label('Which
      information do you want to change? ')),
  send(@change_info, append, label(' ')),
  send(@change_info, append, button('Explanation',
     cascade(0, change_explanation,0))),
  send(@change_info, append, button('End user',
             cascade(0, change_end_user,0))),
  send(@change_info, append, button('Level of knowledge',
          cascade(0,change_knowledge_level ,0))),
  send(@change_info, append, button('Quantity of Explanation',
            cascade(0,change_quantity_explanation ,0))),

  send(@change_info, append, button('Quality of Explanation',
          cascade(0,change_quality_explanation ,0))),

  send(@change_info, append, button('Goal',
                    cascade(0,change_goal ,0))),
  send(@change_info, open, point(0,200)),
  process_message_1.


change_explanation(X,_):-
  send(X, destroy),
  retractall(explanation(_)),
  assert(change_explanation_flag),
  explanation_or_not.
change_end_user(X,_):-
```

```
  send(X, destroy),
  retractall(end_user(_)),
  end_or_design_user.
change_knowledge_level(X,_):-
  send(X, destroy),
  retractall(knowledge_level(_)),
  retractall(user_level(_)),
  question_user_3.
change_quantity_explanation(X,_):-
  send(X, destroy),
  retractall(quantity_explanation(_)),
  question_user_4.
change_quality_explanation(X,_):-
  send(X, destroy),
  retractall(quality_explanation(_)),
  question_user_5.
change_goal(X,_):-
  send(X, destroy),
  retractall(goal(_)),
  question_user_6.


kill_prowindows :-
   (object(@atom_response) ->
      send(@atom_response,destroy)|true),
   (object(@atom_text) ->
      send(@atom_text,destroy)|true),
   (object(@hour_map) ->
      send(@hour_map,destroy)|true),
   (object(@hour_glass) ->
      send(@hour_glass,destroy)|true),
   (object(@no_entry_map) ->
      send(@no_entry_map,destroy)|true),
   (object(@no_entry_cursor) ->
      send(@no_entry_cursor,destroy)|true),
   (object(@stop_map) ->
   send(@stop_map,destroy)|true),
(object(@stop_cursor) ->
   send(@stop_cursor,destroy)|true),
(object(@error_window) ->
   send(@error_window,destroy)|true),
(object(@error_return) ->
   send(@error_return,destroy)|true),
(object(@filename_wind) ->
   send(@filename_wind,destroy)|true),
(object(@how_display_window) ->
   send(@how_display_window,destroy)|true),
(object(@how_return) ->
   send(@how_return,destroy)|true),
(object(@how_button) ->
   send(@how_button,destroy)|true),
(object(@enter_is) ->
   send(@enter_is,destroy)|true),
(object(@user_log_window) ->
   send(@user_log_window,destroy)|true),
(object(@user_log_dict) ->
   send(@user_log_dict,destroy)|true),
(object(@response_bitmap) ->
   send(@response_bitmap,destroy)|true),
(object(@menu_ques) ->
   send(@menu_ques,destroy)|true),
(object(@menu_response) ->
   send(@menu_response,destroy)|true),
(object(@message_bitmap) ->
   send(@message_bitmap,destroy)|true),
(object(@point_window) ->
   send(@point_window,destroy)|true),
(object(@point_input) ->
   send(@point_input,destroy)|true),
(object(@input_x) ->
   send(@input_x,destroy)|true),
(object(@point_response) ->
   send(@point_response,destroy)|true),
(object(@point_options) ->
   send(@point_options,destroy)|true),
(object(@prop_ques) ->
   send(@prop_ques,destroy)|true),
(object(@prop_response) ->
   send(@prop_response,destroy)|true),
(object(@selection_dict) ->
   send(@selection_dict,destroy)|true),
(object(@selection_window) ->
   send(@selection_window,destroy)|true),
(object(@selection_input) ->
   send(@selection_input,destroy)|true),
(object(@selection_response) ->
   send(@selection_response,destroy)|true),
(object(@selection_options) ->
   send(@selection_options,destroy)|true),
(object(@slider_ques) ->
   send(@slider_ques,destroy)|true),
(object(@slider_response) ->
   send(@slider_response,destroy)|true),
(object(@slider) ->
   send(@slider,destroy)|true),
(object(@whynot_wind) ->
   send(@whynot_wind,destroy)|true),
(object(@whynot_return) ->
   send(@whynot_return,destroy)|true),
(object(@whynot_button) ->
   send(@whynot_button,destroy)|true),
(object(@message_window) ->
   send(@message_window,destroy)|true),
(object(@parm_menu_dict) ->
   send(@parm_menu_dict,destroy)|true),
(object(@parm_brows) ->
   send(@parm_brows,destroy)|true),
(object(@explain_sheet) ->
   send(@explain_sheet,destroy)|true),
(object(@explain_dict) ->
   send(@explain_dict,destroy)|true),
(object(@abort_ques) ->
   send(@abort_ques,destroy)|true),
(object(@abort_response) ->
   send(@abort_response,destroy)|true),
(object(@yes_cascade) ->
   send(@yes_cascade,destroy)|true),
(object(@abort_yes) ->
   send(@abort_yes,destroy)|true),
(object(@no_cascade) ->
   send(@no_cascade,destroy)|true),
(object(@abort_no) ->
   send(@abort_no,destroy)|true),
(object(@query_display) ->
   send(@query_display,destroy)|true),
(object(@filename) ->
   send(@filename,destroy)|true),
(object(@query_button) ->
   send(@query_button,destroy)|true),
(object(@ques_ans_window) ->
   send(@ques_ans_window,destroy)|true),
(object(@ques_ans_done) ->
   send(@ques_ans_done,destroy)|true),
(object(@why_rule_window) ->
   send(@why_rule_window,destroy)|true),
(object(@why_complete) ->
   send(@why_complete,destroy)|true),
(object(@why_button) ->
   send(@why_button,destroy)|true),
(object(@why_text) ->
   send(@why_text,destroy)|true),
(object(@why_file_window) ->
   send(@why_file_window,destroy)|true),
(object(@why_file_return) ->
   send(@why_file_return,destroy)|true),
(object(@is_query) ->
   send(@is_query,destroy)|true),
(object(@load_window) ->
   send(@load_window,destroy)|true),
(object(@main) ->
   send(@main,destroy)|true),
(object(@system_menu) ->
   send(@system_menu,destroy)|true),
(object(@explain_menu) ->
   send(@explain_menu,destroy)|true),
(object(@editor_menu) ->
   send(@editor_menu,destroy)|true),
(object(@consult_wind) ->
   send(@consult_wind,destroy)|true).
```

VITA AUCTORIS

Hien My Bui was born in 1959 in Vinh Long, VietNam. She completed her high school education at Pho Thong Cap 3 High School in 1976. She graduated from the University of Windsor in 1989 with a Bachelor of Science in Computer Science. In September 1993 she finished her Master of Science in Computer Science at the University of Windsor.