A DATABASE INTERFACE SYSTEM

FOR THE

PREOP EXPERT SYSTEM

A DATABASE INTERFACE SYSTEM
FOR THE
PREOP EXPERT SYSTEM

By

MATHABO M. PHARASI B.Sc.

A Project

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements

for the Degree

Master of Science

McMaster University

MASTER OF SCIENCE (1991) McMASTER UNIVERSITY
(Computation) Hamilton, Ontario

TITLE: PREOP-DIS. A Database Interface System for the PREOP Expert System

AUTHOR: Mathabo M. Pharasi, B.Sc. (UNISA)

SUPERVISOR: Dr. N. Solntseff

NUMBER OF PAGES: viii, 145

ABSTRACT

PREOP is a medical expert system which is being developed on the expert-system shell NEXPERT. PREOP determines the risk of cardiac complications for a patient about to undergo non-cardiac surgery, and recommends a programme of medication intake for the patient. PREOP's final database and hardware requirements are uncertain, and so are its final functionalities. These uncertainties necessitate a database interface which facilitates overall system flexibility, maintainability, modularity, and integration with existing information systems. These objectives are best achieved by a loosely coupled database interface which is not under NEXPERT's control.

PREOP-DIS is a loosely coupled database interface system for PREOP. This project uses the relational model to design PREOP's database and implements the database on the relational DBMS SYBASE. The project implements PREOP-DIS as a C program that communicates data between PREOP and the SYBASE database by dynamically making calls to NEXPERT and to the SYBASE dataserver.

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to my supervisor, Dr. N. Solntseff, for his guidance, support and encouragement throughout the development of this project. I also wish to thank my first reader, Dr. P.J. Ryan, for his valuable comments.

This work is dedicated to my late father and to my mother, without whose encouragement and countless sacrifices it would not have been possible for me to embark on my studies, and to my daughter Hlasedi.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Chapter 1

INTRODUCTION

1.1 Introduction

An expert system is an Artificial Intelligence (AI) application designed to simulate the reasoning of an expert in a particular field of expertise such as medical diagnosis or risk analysis for stockbrokers. Expert systems have been used with success, especially in the commercial world [BLA90]. The lack of online data access and integration with other information systems has plagued expert systems since their introduction in the early 70's; in fact this was one of the reasons why MYCIN, the first expert system for medical diagnosis, was never in routine clinical use [BUC84].

Medical expert systems have not been as successful as their commercial cousins in being accepted as decision-support tools [HUN84] [BUC84]. Although a few systems are in general medical use, such as INTERNIST-1 which is used to diagnose internal medicine disorders, none has been methodically evaluated. What makes a good medical expert system is therefore not clear, and the circumstances under which a physician will use an expert system is not understood.

The Pre-Operative Assessment (PREOP) expert system is

being developed to assist consultants in general internal medicine to assess pre-operatively the risk of post-operative or peri-operative cardiac complications in patients about to undergo non-cardiac surgery and to advise on the management of such patients. PREOP's expert knowledge is stored as *antecedent-consequent* rules of the form:

IF patient-age > 70 (*antecedent*)

THEN an age risk-factor is established (*consequent*)

The results of the PREOP project will be used for evaluating expert system techniques in medicine and for investigating the integration of expert systems into mainstream medical information systems.

The PREOP Database Interface System (PREOP-DIS) is a data communication mechanism between the PREOP expert system and an external database. This project designs PREOP's database and implements PREOP-DIS as a C program that dynamically makes calls to PREOP and to a database dataserver. PREOP-DIS is designed to be portable, flexible, and to facilitate PREOP's access to several Database Management Systems (DBMS).

Given the general unacceptability of expert systems to the medical profession, it was necessary and appropriate to use a prototype as a development tool for PREOP-DIS. The aim of a prototype system is to establish system requirements and design weaknesses by letting the user experiment with the

prototype. The system requirements thus derived can then be used to develop a production-quality system [SOM89]. One of the major benefits of prototyping is that a working model, however rudimentary, is available quickly to demonstrate the feasibility of the system, especially to a sceptical user or management community. Buchanan [BUC84] has this to say about prototyping: "One important reason for the success of our early efforts was Shortliffe's ability to provide quickly a working prototype".

The users of expert systems are generally the experts themselves, and so is the case with PREOP. Its intended users are consultants in general internal medicine, assisted by research assistants (RA) with nursing qualifications and experience. The primary duty of the RA will be to provide initial routine data to the system, data currently recorded on hospital charts. Another reason for using RA's is to reduce the consultants' time on the system so that they can concentrate on problem solving as much as possible. The MYCIN evaluation tests have shown that a physician's time on the system and poor user-friendliness are negative factors in system acceptability [BUC84].

1.2 Background

The Health Information Research Unit (HIRU) of the Department of Clinical Epidemiology and Biostatistics, the

Department of Computer Science and Systems, and the Faculty of Business, all of McMaster University, are collaborating in the development of three expert systems for medical use. One of these systems is PREOP. The second is a computerized dosing system for adjusting the dose of the drug cyclosporin for patients with rheumatoid arthritis. The third is a system for the management of chest pain, the diagnosis of ischemic heart disease and its prevention and treatment. Work on the first two systems is in progress. Development of the chest pain management system will depend on the progress of PREOP and the computerized dosing system.

PREOP is a revision of a similar system which was developed by HIRU using PERSONAL CONSULTANT PLUS™, an expert system shell developed by TEXAS INSTRUMENTS INC. in the programming language SCHEME. The original system was rewritten because of lack of portability across operating systems, its closed architecture, a practical limit to the number rules that can be used, and difficulty in modifying rules that interact with other rules.

NEXPERT™ is a rule-based expert-system shell developed by NEURON DATA INC. as a modular collection of C functions. It consists of an inference engine, an integrated development environment, and an interface for communicating with other external applications. NEXPERT is available for several operating systems including VMS, UNIX, DOS and the Apple

operating system for the Macintosh, which factor makes it highly portable across operating systems. The development environment has no limitations on the size of knowledge base that can be built. Its utility to view the rule network online facilitates addition and modification of rules whilst keeping track of the effects of these changes. NEXPERT has a mechanism for executing external procedures. An external application written in C, Pascal, FORTRAN, or assembler can in turn interrogate and manipulate NEXPERT's knowledge base by using a system of function calls which collectively form the Application Programming Interface. This open architecture makes it possible for a NEXPERT application to be embedded in other non-NEXPERT applications.

Because of the above-mentioned strengths of NEXPERT over PC PLUS, NEXPERT was chosen as an alternative development tool for PREOP. Work on NEXPERT started in September 1989, and is expected to be ready for preliminary testing in September 1991.

PREOP consists of an inference engine, a knowledge base, an external database, a database interface module and a user interface module. The knowledge base includes the set of rules that is used to store the consultants' reasoning expertise and a representation of data from the external database. The user interface is used by the inference engine to hold a dialogue with the expert user during the inferencing process.

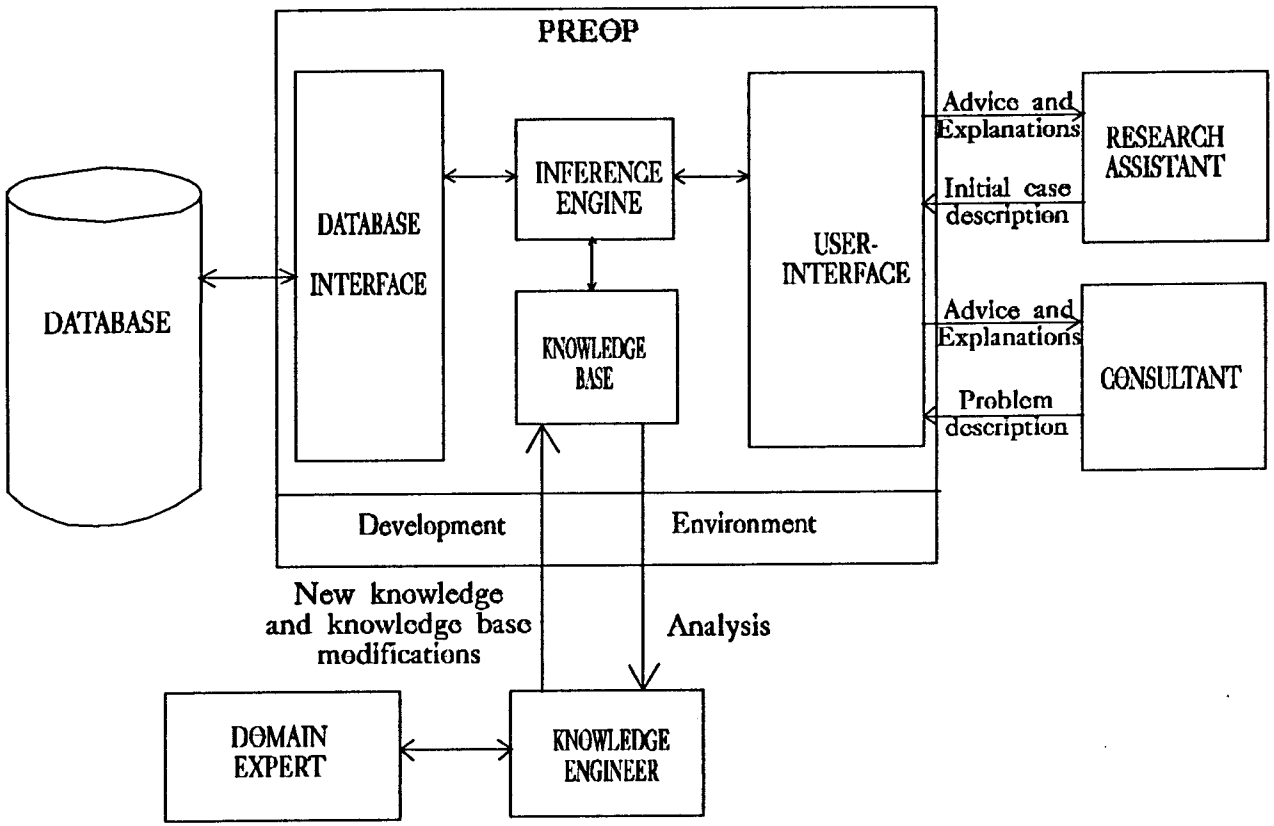


Figure 1.1 PREOP system overview and its interaction with users

1.3 System Scope and Constraints

The database environment of the final PREOP expert system has not yet been fully defined. It depends partly on the choice of database to be used by the Chedoke-McMaster hospital to store patient demographic information. Another source of uncertainty is whether medical information such as laboratory tests, X-Ray pictures and ECG records will be kept in a database or will be accessed manually from hospital charts. Therefore, the homogeneity of the final database must not be taken for granted, since the required demographic, historical and current medical information may be stored using different technologies. At present the Chedoke-McMaster hospital uses SYBASE™ tables and VMS RMS files. SYBASE is an SQL-based relational Database Management System developed by SYBASE Inc. [SYB89]. VMS is Digital Equipment Corporation's (DEC) operating system for its VAX mainframes, RMS is a file management system for the VAX [DEC89].

The question of hardware environment adds yet another uncertainty factor to the PREOP project. There are two dominant views within PREOP's project team on the choice of hardware. One is that a mainframe, preferably a VAX, should be used, whilst the other prefers a DOS implementation. A compromise option is to develop a general system that will run on both DOS and VAX machines. These uncertainties, as well as the fact that the size and scope of the final product is

uncertain, suggest a database design which is implementation independent and a prototype system which is easy to maintain and extend.

Confidentiality of patient information is an important consideration for medical information systems. In this regard PREOP-DIS must take into account guidelines for computerized medical records as set by the College of Physicians and Surgeons of Ontario, which state that there must be restricted access to a system, and that the system must have an audit trail feature [CPS88].

Chapter 2

EXPERT SYSTEM CONCEPTS

This chapter will introduce the concept of expert systems as an application of Artificial intelligence. It will discuss some of the problem solving methods used in expert systems, with a detailed description of rule-based systems.

2.1 AI and Expert Systems

The past two decades have seen the development of computer systems that can perform "intelligent" tasks such as game playing, natural language comprehension, medical diagnosis, etc. The field of study concerned with such systems is called Artificial Intelligence (AI). AI technology is applied in seemingly diverse areas such as expert systems, machine vision, robotics, natural language processing, automatic theorem proving, automatic programming, and intelligent data retrieval. This chapter will discuss expert systems as an application of AI.

There is no clear definition of AI (which may not be surprising since there is no clear definition of intelligence), instead several similar definitions can be found in the literature. For example, Nilsson [NIL80] defines

an AI system as one that emulates human mental activity, while Winston [WIN84] defines AI as the study of ideas that enable computers to be intelligent. Campbell [CAM84a] discusses three criteria that have generally been used to characterize a system as belonging in the AI field, viz., the system reproduces results which have previously only been observable in intelligent humans, uses recognized AI techniques such as backtracking or the programming languages LISP and PROLOG, or exhibits behaviour that can be described as learning. Campbell goes on to speculate that a plausible fourth criterion is that the system be voted as an AI system by the American Association of Artificial Intelligence (AAAI), or by major conferences of the International Joint Conference on Artificial Intelligence (IJCAI) and the European Conference on Artificial Intelligence (ECAI).

Notwithstanding numerous definitions of AI, AI specialists generally agree with Minsky that AI "..is a science of making machines do things that would require intelligence if done by men" [YAZ84]. An important aspect of this definition is the "black-box" nature of the system "..without any prejudice toward making the system simple, biological or humanoid" [MIN68].

Much of AI work, with the exception of expert systems, has not progressed far beyond research laboratories and universities. Expert systems are AI's commercial success

story. "The evolution of expert system technology passed a milestone in the past 3 years. During 1987-89, dozens of major companies, many from Fortune 500, used expert system technology to build high value production systems. Digital Equipment Corporation estimates that its landmark expert system, XCON/XSEL saves the company $40 million per year" [BLA90]. There are several such success stories outside the corporate world, for example, DENDRAL, regarded by some as the original expert system [BUC84], is a system for identifying chemical structures from mass spectrograms. DENDRAL has been shown to perform consistently better than any human expert in its particular field [CAM84b]. MYCIN, a system to diagnose human bacterial infections and to prescribe appropriate treatment, vies with DENDRAL for the position of the first useful expert system [BUC84]. MYCIN is a landmark for all expert systems which represent knowledge with rules of the form: IF (conditions are true) THEN (perform actions).

It is ironic that expert systems, as the most successful of AI applications, are the most difficult to define. As is the case with AI, there is no clear definition of expert systems. Whereas Nilsson [NIL80] uses the familiar "black-box" definition that expert systems provide human users with expert conclusions about specialized areas, Winston [WIN84] avoids the term altogether and prefers instead to use the term "problem solver". An informal characterization of an expert

system is that it must be able to produce conclusions which can surprise a human expert in the field [NAR84].

On the other hand it is generally accepted that an expert system consists of at least two parts: an assembly of knowledge and an inference engine which reasons about this knowledge. Heuristics are an essential and most powerful part of this knowledge. They represent intuitive problem solving tools that have a high rate of success. Keyes [KEY90] captures this idea when she says "We all know it's easy to expert-systemize about 80% of the knowledge in a given field since it comes from text-books and procedure manuals. It's the other 20%, the heuristics, the gut feelings, the 'what makes this person special', that is so hard to encode. But it's this 20% that makes an expert system expert. Without it, you have merely a pretty-smart system".

2.2 Knowledge Acquisition and Representation

One reason for the success of expert systems is that instead of being general problem solvers, they concentrate on specialized areas of knowledge. "The fact that they have cornered valuable blocks of specialized knowledge is the foundation of their healthy bank-balances" [CAM84a]. The specialized area of knowledge, referred to as the **domain** of expert application, generally has professional experts already practising in the field. These professionals are referred to

as **domain experts.**

The conceptualization and structuring of the domain expert's knowledge and its representation in the computer, i.e., knowledge acquisition and representation, is a bottleneck in the development of expert systems [BUC84] [CAM84a]. This process of transferring domain expertise from a domain expert into a computer system has developed into a well defined sub-field of AI known as **knowledge engineering.** The computer professional who interviews the domain expert, analyzes the expert knowledge thus derived and chooses the appropriate formalism to represent this knowledge in its proper context is known as a knowledge engineer. The most difficult task of the knowledge engineer is to extract knowledge that even the experts are not aware that they have. The knowledge engineer therefore requires good communication and analytical skills in addition to computer expertise.

2.3 Generate-and-Test Systems

The nature of expert systems as problem solvers calls for the use of one of several problem solving paradigms, the two most popular being the generate-and-test systems the rule-based systems [WIN84].

A **generate-and-test** system consists of two modules, a generator and a tester. The **generator** produces possible solutions and the **tester** evaluates each proposed solution and

either accepts or rejects it. Heuristics are used to limit the number of solutions proposed. This paradigm is generally used to solve identification problems, in which case the proposed solutions are referred to as hypotheses.

The expert system DENDRAL, which identifies chemical structures from their mass spectrograms, is a generate-and-test system. The generator module enumerates all possible chemical structures from a given chemical formula and synthesizes a mass spectrogram for each enumerated structure. The tester module compares each synthetic spectrogram to a real experimental spectrogram to find a reasonable match.

2.4 Rule-Based Systems

A **rule-based** system uses sequences of rules as a problem solving tool. A rule has the following form:

IF (condition-1, condition-2, ..., condition-n)

THEN (action-1, action-2, ..., action-m)

This means that if all the conditions in the rule are *true* the specified actions are all performed. The *IF* clause is known as the Left-Hand-Side (LHS) and the *THEN* clause as the Right-Hand-Side (RHS).

The idea of using rules to represent knowledge dates back to ancient Babylonian tablets about 650BC. Some of these tablets catalogued rules to predict a person's destiny, such as the following:

"If a man unwillingly treads on a lizard and kills it, (then) he will prevail over his adversary" [BUC84].

(Judging from the list of these rules in [BUC84], either ancient Babylonian societies had no women or women's destinies were too insignificant to predict.)

A **production system** is a rule-based system that operates on a specialized area of the real world, the domain, and consists of three distinct components: a database, a set of production rules, and a control mechanism [NIL80]. A set of terminating (or goal) conditions may also be specified. Whilst production systems were first proposed as a general computational formalism in 1943, it was not until the early 70's with the advent of MYCIN that they were used in expert systems [BUC84]. Productions systems have since become the most popular methodology for expert systems.

A **database** is a structured collection of all known facts about the domain. It represents the state of the domain at any given point in time. The database is alternatively referred to as the declarative representation of knowledge. The set of terminating conditions represent possible states of the database.

The set of **production rules** embody generalizations about and relationships among entities in the domain. A rule is applicable if its conditions match the facts of the database.

An applicable rule is said to be fired if its actions are effected. Several rules may be applicable at the same time, but only one rule will be fired at a time. Uncertainty can be built into a rule by specifying its accuracy in the RHS, or the weight of its LHS that is necessary to make the rule applicable. The following is an example of a MYCIN rule which incorporates uncertainty, from [BUC84]:

> "IF the gram stain of the organism is neg, and
> the morphology of the organism is rod, and
> the earobicity of the organism is anaerobic
> THEN there is 0.6 suggestive evidence that the
> identity of the organism is bacteroides"

The set of production rules is referred to as the rule base or the procedural representation of knowledge. The database and the rule base are together referred to as the knowledge base. Expert systems are therefore regarded as knowledge-based systems.

Production systems differ from traditional hierarchically organized programs in that the entire database is accessible to all rules, i.e., the database has a global scope; rules operate only on the database and do not call other rules, i.e., communication between rules is via the database [BUC84] [NIL80].

The control mechanism, or **inference engine**, is a domain independent interpreter of rules. It uses a control strategy to select an applicable rule to fire and to keep track of the sequence of rules that have been fired so far. The sequence of applicable rules is determined from the initial database by repeatedly applying *modus ponens*. Modus ponens is a theorem that states that if A implies B and it is given that A is true, then B is also assumed to be true. The rules are fired until no more can be fired or until the database satisfies some terminating condition. Most inference engines possess knowledge about their state of inferencing and can therefore explain how they reached a conclusion or why they need information.

Although a user interface is not an essential part of a classical production system, it is found in almost all expert systems, especially analyzing systems such as MYCIN. One reason is that some information about the domain cannot be represented in a database and therefore dialogue with the user becomes necessary. For example, a patient's temperature or pulse rate will be taken by a user of MYCIN. Another reason is that the user may suspend the inferencing process to enquire why a certain conclusion was reached.

The early expert systems were domain specific and showed no clear separation between the knowledge base and the inference engine. Currently the trend is to develop **expert-**

system shells whose major components are an inference engine, a user interface and a knowledge base development environment. One expert-system shell can be used to develop several knowledge bases each, representing a different domain. Each knowledge base is stored on a file separate from the shell. EMYCIN (Essential MYCIN) is an expert-system shell that represents a generalization of MYCIN [BUC84]. Whereas the early systems were written in dialects of the logic programming languages LISP or PROLOG, today's popular shells are developed as a collection of C, FORTRAN or Pascal procedures [BLA90]. Such systems are therefore portable and can be embedded in existing applications.

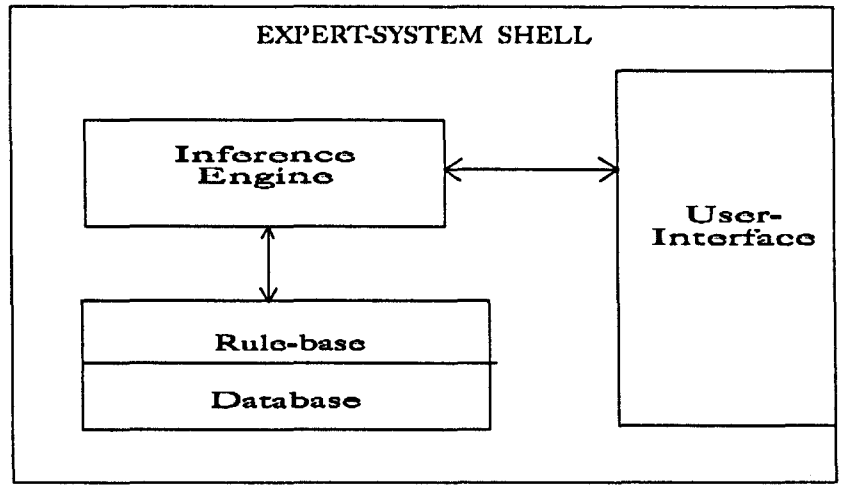


Figure 2.1 Major components of a rule-based expert-system

Table 2.1 Major commercial expert-system shells. Adapted from [BRW90]

VENDOR	ES-SHELL	LANGUAGE
Texas Instruments	PC Plus	Scheme
Gold Hill	Goldworks	Lisp
Aion	ADS	Pascal
Neuron Data	Nexpert	C
Software Artistry	PC Expert	C,Pascal,Modula-2

2.4.1 Control Strategies

The algorithm that an inference engine uses to fire an applicable rule from several applicable rules is its control strategy. Control strategies fall into two major groups: **tentative** and **irrevocable** control strategies. In an irrevocable strategy applicable rules are fired irrevocably without provision for later re-consideration. In a tentative strategy an applicable rule is fired according to some order, but provision is made to restore the database to a state just before the rule was fired.

In Figure 2.2 each node represents a database state, starting with the initial state. Each directed arc is an applicable rule that transforms the database from one state to another. A path is a sequence of arcs between two nodes. Figure 2.2 will be used to describe the different strategies in detail.

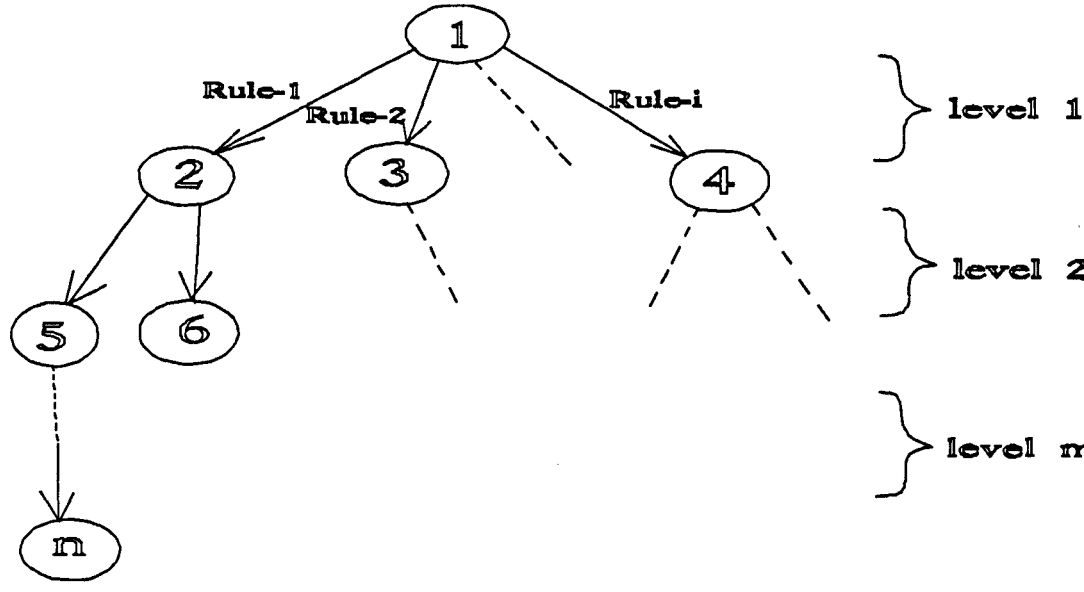


Figure 2.2 Tree of database states and applicable rules

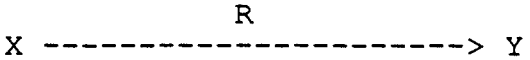
Breadth-first and depth-first are well-known irrevocable strategies. In a **breadth-first** strategy all applicable rules at one level are fired before the next level is considered. In a **depth-first** strategy an applicable rule with the highest priority (according to some pre-determined order) is fired and its path is followed until no more rules can be fired. If no goal state is reached, the next applicable rule in the lowest level is chosen for firing and process is repeated. For both strategies the inference engine must remember all the database states that have been produced.

Backtracking is a tentative strategy similar to the depth-first strategy, with the difference that if a rule does not lead to a goal-state its actions are undone and the next applicable rule is fired. Therefore the inference engine need

only remember the states and rules in the current path.

2.4.2 Forward/Backward Deduction

Let X,Y be database states and R be the sequence of rules which finally produces state Y when applied on state X. X and Y are known as the initial and goal sets respectively. The triple (X,Y,R) is defined as a **problem** if only 2 of its components are known. Problem solving is then the process of searching for the 3rd component [BRU90]. Rule-based systems are concerned with deduction problems of the form (X,?,R) and (?,Y,R). In this case Y represents a goal hypothesis, X the data which is required to make Y true, and R the sequence of rules which are used to deduce Y from X. The *IF* part of a rule is the antecedent whilst the *THEN* part is the consequent.



Forward deduction, or forward chaining, is the process of searching for the goal state in the triple (X,?,R). Starting with the initial state, the inference engine executes a sequence of inference cycles until the goal state is reached. At every inference cycle the inference engine deduces the next database state from the current one by firing one rule from the set of applicable rules. Forward deduction is data oriented. It is generally used by synthesizing systems such as XCON, a forward deduction system used by DEC to configure

computer systems [WIN84].

Backward deduction, or backward chaining, is the process of applying the sequence R in reverse order to determine the initial state which is necessary to produce the goal state Y in the triple (?,Y,R). At each inference cycle an applicable rule is one whose actions will produce the current goal state. The state that satisfies the conditions of the chosen applicable rule becomes the new subgoal state. The process is repeated until no applicable rule is found. Backward deduction is alternately known as goal-oriented reasoning. It is generally used by analyzing systems such as PROSPECTOR, an analyzing system which interprets oil-well logs [WIN84].

2.5 Summary

An expert system is designed to automate routine problem solving reasoning of an expert in a particular field, and to be used as a decision support tool by the expert. Because expert systems manipulate knowledge, they are regarded as knowledge-based systems.

The process of conceptualizing an expert's knowledge and choosing structures for its representation in the computer, known as knowledge engineering, is the most difficult task in the development of expert systems. Whereas knowledge engineering is a well-defined subfield of AI, it does not as yet have accepted practices or standard methodologies [ELI90].

A production system consists of a database, a set of production rules (or rule base) and a control mechanism that uses the facts in the database to reason about the rules. The execution time version of the database and the rule base are together known as the knowledge base. While the knowledge base depends on the system's domain of application, the control mechanism is domain independent and can be used with different knowledge bases.

An expert system shell is a general-purpose production system with a user interface and a development environment added. Expert-system shells are commercially viable because they can be sold to clients with different problem domains. Moreover, the knowledge base is built by the clients themselves, who have less a chance of failure than the shell vendors.

Learning systems, i.e., systems that can modify their own rules or add new rules whilst preserving system consistency, are yet to be developed. In the meantime it seems unrealistic to consider machine-learning as a necessary characteristic of expert systems.

Chapter 3

NEXPERT CONCEPTS

NEXPERT is an expert-system shell based on the production system model. It consists of an inference engine, a development environment called NEXPERT Object™, a user interface, an Application Programming Interface for communicating with external applications and a Database Links tool for accessing external databases [NEU88] [NEU89]. This chapter will discuss the structures NEXPERT Object uses to represent the knowledge base, the different control strategies used by the inference engine, and the Application Programming Interface. The limitations of the Database Links tool with respect to the PREOP project will then be discussed.

3.1 The Knowledge Base

The primary user of NEXPERT Object is a knowledge engineer, who uses it to develop the knowledge base (KB). The KB consists of a rule base and a database known as an object base. Part of the object base is an internal representation of an external database (or part thereof) which may be managed by a DBMS. The other part is control structures which are used locally to control the reasoning process and which do not in

themselves add new knowledge to the system. The KB is stored as a text file. A KB-file is loaded into NEXPERT and compiled by the inference engine before it can be ready for execution.

3.1.1 The Object Base

The basic structure in an object base is the **object**. It is an abstraction of some identifiable item, event or phenomenon in the real world. Closely related to the object structure is the **property** structure. In fact the one and only purpose of properties is to characterize object attributes. The type of a property is the set of all possible values that an object property can have. NEXPERT property types are boolean, integer, float, string, date, and time. For example, the object *patient* may have properties *id, sex, height* of types integer, string, float respectively. One property can be used to characterize several objects.

A **class** is a generalization of several similar objects. It encompasses the concept of object membership to a class. One object can have more than one parent class. Objects in the same class generally have one or more properties in common, although this is not a requirement of class membership. A class is characterized by properties in the same way as objects are. When an object is defined as a member of some class, NEXPERT can automatically attach the class properties and property values to the object. This phenomenon is known as

inheritance. It is the knowledge engineer's prerogative to specify the inheritability of class properties and their values by its member objects. In addition to class properties, an object can have other properties that are not common in the class. The relationship between an object and its parent class is equivalent to the IS-A relationship in the entity-relationship model for databases. A class can have **sub-classes**.

The class structure is not only useful for organizing the object base, it is also used to manipulate some or all member objects without explicitly referring to them one by one. For example, the statement

DO <|players|>.score + 1 (on) <|players|>.score

will add 1 to the score of all member objects in the class *players*. This method of referring to member objects as a group is known as **interpretations**.

An object which has identifiable components can have **sub-objects** to represent these components. A sub-object can inherit the properties of its parent object. As with classes, a parent object can be used to manipulate its sub-objects.

An object is "split" into slots according to its properties. A **slot** is the smallest functional unit in the object base, i.e., it is the smallest structure that can be manipulated by a NEXPERT statement or function. A slot is identified by the object property it refers to in the form:

object.property, similar to the **relation.attribute** form used in relational databases. The data type of a slot is the data type of its property.

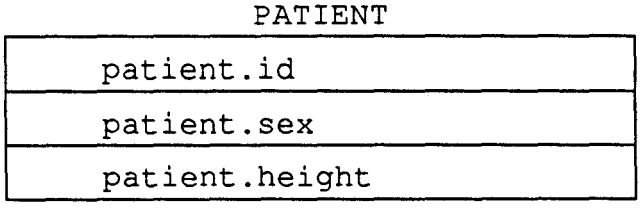


Figure 3.1 Slots for object patient

If a slot has the special property **Value**, this property can be left out when referring to the slot. For example in the object *error* the slot identifiers *error* and *error.Value* refer to the same slot. The context in which the identifier is used will determine whether it refers to an object or to its slot. Objects which have only the property *Value* are generally used locally to control the inference flow.

NEXPERT does not allow interpretations on the property *Value*, i.e., a term such as <|players|>.Value is invalid.

NEXPERT has three special slot functions of type boolean: UNKNOWN, KNOWN, NOTKNOWN. If a slot value has not yet been determined its UNKNOWN function will evaluate to *true* and its KNOWN and NOTKNOWN functions will evaluate to *false*. If, when prompted for a slot value, a user responds that she does not know, the slot's NOTKNOWN and KNOWN functions will evaluate to *true*. If the user supplies a value, the slot's UNKNOWN and

NOTKNOWN functions will evaluate to *false*. At any given time a slot's UNKNOWN and KNOWN functions will not evaluate to the same value. The NOTKNOWN function is the closest NEXPERT can get to representing uncertainty.

The RESET statement is used to reset a slot to its "null" value, i.e., its value becomes undetermined and its UNKNOWN function evaluates to *true* again.

A **metaslot** (slot about a slot) is used to define the behaviour of a slot during an inference session. It is used by the inference engine only and cannot be used in a NEXPERT statement or function. Metaslots are used to define the initial value of a slot, the action to be taken each time the slot value changes, or the string that will be used in the user interface to prompt the user for the slot value. Metaslots are inheritable. Unlike properties, a metaslot can be associated with only one slot.

Table 3.1 Similarities between the object base and the relational model for databases

Object Base	Relational Model
class	relation
object	tuple
property	attribute

The similarities between the object base and the relational model for databases facilitates the design of the object base from an external relational database and the communication of data between the two. Using Table 3.1, member objects of a class can be seen as tuples of a relation.

3.1.2 The Rule Base

A rule base is a set of production rules. It is the "program" of a NEXPERT system. The statements and functions that make up a rule are used to interrogate and manipulate the object base, to access external databases, and to execute external procedures. A rule has the format:

```
          LHS                              RHS
---------------------------    ---------------------------
(if) condition-1 (is true)     (then) hypothesis (is true)
(and) condition-2 (is true)    ( do ) action-1
          .                             action-2
          .                                .
(and) condition-n (is true)              .
                                         action-m
```

The terms in brackets are implied and are not part of the rule syntax. The LHS consists of an ordered sequence of conditions. A condition is boolean valued; it can be a relational expression, another hypothesis, or an executable statement. When a statement is used as a condition it evaluates to *true* (*false*) if it executes successfully

(unsuccessfully). Any slot of type boolean can be used as a hypothesis. The first "line" in the RHS is always an hypothesis, the rest is an ordered sequence of executable statements.

If all the conditions on the LHS, in order from top to bottom, evaluate to true, the hypothesis is set to true and the actions in the RHS are performed from top to bottom. An applicable rule is one whose LHS is true. A rule whose RHS has been executed is said to be fired. A conflict arises if two or more rules are simultaneously applicable.

An **inference category** of a rule is an integer which is used for conflict resolution. Of two applicable rules, the one with the highest inference category will be fired first. When a rule is first created its inference category is automatically set to 1.

A **context** is a metarule (rule about a rule) which tells the inference engine when to evaluate a rule. If ruleB is defined in the context of ruleA, ruleB can only be evaluated after evaluation of ruleA. Contexts are usually used for rules which are related but whose relationship somehow cannot be represented in the text of a rule. For example it is reasonable to determine if a patient is pregnant only if the patient is female.

A **strategy** is an attribute of a rule context which is used to further restrict the conditions under which a rule is

evaluated. In our example, the strategy *Propagate when True* in contextA tells the inference engine to evaluate ruleB only if ruleA evaluates to *true*. The context mechanism thus acts as a heuristic for minimizing the cost (in time and number of evaluated rules) of an inference session.

Rules which refer to the same slots are said to have **strong links**. Conflicts usually arise with rules which have no strong links. This conflict can be resolved by using inference categories or contexts. Two rules are said to have **weak links** if they have no strong links and one rule is in the context of the other. A symmetric weak link occurs if both rules are in each other's contexts.

As in all production systems, rules cannot "call" other rules. In a NEXPERT Object rule this constraint is bypassed by resetting the rule's hypothesis to its original "null" value and later referring to this reset hypothesis in a statement. In its attempt to re-determine the value of the reset hypothesis, the inference engine will re-evaluate its rule. When a rule is re-evaluated all the hypotheses which appear in its LHS are reset and their rules are similarly re-evaluated. This may cause cascading resets and re-evaluations. A rule will loop if its RHS resets the rule's hypothesis and subsequently refers to this hypothesis. The rule will be fired repeatedly until its LHS evaluates to *false*. The knowledge engineer must make sure that looping will eventually

terminate.

3.2 The Inference Engine

The inference engine is a knowledge base independent control mechanism and functions as the "operating system" of NEXPERT. Its reasoning regime is not fixed to either forward or backward deduction; both can be used during a single session. Forward and backward deduction are referred to in NEXPERT as forward-chaining and backward-chaining respectively. The inference engine is started when the NEXPERT function NXP_Control(NXP_CTRL_KNOWCESS) is executed either directly from an external application or via the NEXPERT user interface.

Forward-chaining occurs when a set of slot values are used to evaluate and fire some sequence of rules. This evaluate-fire sequence is stopped when all applicable rules have been fired. Because forward-chaining is data oriented, it may result in the evaluation of rules which are not linked.

Backward-chaining is used for goal oriented reasoning. Backward-chaining starts from a given goal hypothesis and proceeds backwards to establish the set of slots which make the hypothesis true. If several rules point to the same hypothesis the knowledge designer can specify in one of the rules if all of the rules or only one of them will be evaluated during deductive reasoning.

The NXP_Volunteer() function is used by an external routine or by NEXPERT's user interface to assign a value to a slot. Forward-chaining is initiated when the inference engine is started after some slot's value has been volunteered.

Similarly, the NXP_Suggest() function is used to initiate backward-chaining. It has the same form as the NXP_Volunteer() function except that its parameter must be a hypothesis. The truth value of the suggested hypothesis is only assigned to it at the end of the inference session.

3.3 The Application Programming Interface

NEXPERT OBJECT has been developed in the programming language C; all its functions, executable statements and system control procedures are C functions. The Application Programming Interface is a collection of C functions which can be called by an external C, Pascal, FORTRAN, or assembler program to interrogate or modify the knowledge base or to control the inference engine.

The inference engine can in turn execute external routines written in C, Pascal, FORTRAN, or assembler. This is achieved by specifying an EXECUTE statement in the LHS/RHS of a rule, which has the form:

```
EXECUTE Arg1 Arg2
```

where *Arg1* is a quoted string referring to the name of the

external routine, *Arg2* is a list of parameters that are passed to the routine. *Arg2* has the form:

```
@ATOMID=atomlist, @STRING=stringval
```

where *atomlist* is an ordered list of comma-separated slots which will be manipulated by the external routine, *stringval* is a string literal.

The external routine must be defined in advance to the inference engine by means of the NEXPERT function NXP-SetHandler(). This function has the form:

```
NXP-SetHandler(nxp-code, ext-name, "nxp-name")
```

where *nxp-code* is a NEXPERT code that indicates how the external routine will be used, *ext-name* is the name of the routine and *nxp-name* is its reference in NEXPERT. The two most frequently used *nxp-codes* are NXP_PROC_EXECUTE and NXP_PROC_QUESTION.

NXP_PROC_EXECUTE indicates that the corresponding routine will be accessed by means of an EXECUTE statement. An external routine corresponding to an EXECUTE statement must have the standard form:

```
int ext-name(atomstr, atomcnt, atomlist)
```

where *atomstr* is a pointer to the @STRING parameter of the corresponding EXECUTE statement, *atomcnt* is the number of

slots in the @ATOMS parameter, and *atomlist* is an array of pointers to the slots in the @ATOMS parameter. If the routine is used as a condition in a rule it must return 1 (one) on successful execution and 0 (zero) otherwise. If NEXPERT does not receive any return value it assumes that the routine execution was unsuccessful.

For NXP_PROC_QUESTION, *ext-name* is a general routine that will be executed whenever NEXPERT requires a slot value from the user, *nxp-name* is a null character pointer. The user-defined routine will replace NEXPERT's default questionhandler routine. A user-defined questionhandler routine has the standard form:

```
int ext-name(slotid, prompt)
```

where *slotid* is the slot whose value is required and *prompt* is a pointer to the metaslot prompt string associated with the slot.

A NEXPERT statement which is of significance to a questionhandler routine is the IS statement, which is used as a condition in the LHS of a rule and has the form:

```
IS slotid "value-1", "value-2", ..., "value-n"
```

The condition is true if the value of *slotid* is any of the list of literals. These literals are compiled into a metaslot of possible choices for *slotid*, and will be suggested to the

user by the questionhandler when it prompts for the value of *slotid*.

Because NEXPERT treats the execution of an external routine like any function call, the routine must be defined in NEXPERT's process environment. Therefore both the routine and NEXPERT must be functions in a program that embeds NEXPERT. When the embedding program executes the function NXP_SetHandler(), it effectively passes to NEXPERT a function pointer to the routine. A program that embeds NEXPERT generally includes the following steps:

1. NXP_Control(NXP_CTRL_INIT) is executed to initialize NEXPERT's working memory. This function must be executed before any of the NEXPERT functions are executed.

2. NXP_SetHandler(NXP_PROC_EXECUTE, ext-name,"nxp-name") is executed for each routine that will be used in an EXECUTE statement.

3. The function NXP_LoadKB("KB-name", KB-address) loads the knowledge base with filename *KB-name* at the address pointed to by *KB-address*. This function must be executed for each knowledge base that is required.

4. A slot in the knowledge base can be assigned value by means of the function NXP_Volunteer(). The inference engine can be prepared for goal oriented deduction by means of the function NXP_Suggest(), which takes an hypothesis as a

parameter. Deduction will be data oriented if no hypothesis is suggested.

Execution of these two functions depends on the knowledge base design and is therefore not mandatory.

5. NXP_Control(NXP_CTRL_KNOWCESS) starts the inference engine. At this point program control is passed on to NEXPERT until the inference session terminates.

6. NXP_Control(NXP_CTRL_EXIT) exits NEXPERT and de-allocates its working memory.

3.4 The Database Links tool

Data is information about specific instances, knowledge is information about general concepts and the relationships between them [WIE86]. Since the terms *specific* and *general* depend on the context in which they are used, data in one context may be knowledge in another. According to Brodie [BRO86a] the distinction between knowledge bases and databases is that "the former require a semantic theory for the interpretation of their contents, while the latter require a computational theory for their efficient implementation on physical machines". It follows then that a system's definition of this distinction will determine how knowledge and data are organized and managed within the system.

The philosophy behind the design and functioning of NEXPERT is to make a clear distinction between data and

knowledge [NEU88]. Data are facts about the expert domain as stored in an external database, knowledge is all the elements of the knowledge base. This distinction is based on the fact that NEXPERT is primarily a knowledge processing system and therefore does not attempt to undertake the data management task of a DBMS. NEXPERT communicates with DBMS's by means of special interfaces known as database bridges. There is a database bridge for each DBMS dataserver that NEXPERT can access.

The close relationship between the object base and relational databases is not only reflected in structural similarities (see section 3.1.1), but is also evident in the types of database transactions that NEXPERT supports. An **atomic transaction** is suitable for queries where the search key is unique; **sequential transactions** are supported by queries which have an "ordered by" clause; **grouped transactions** are suitable for queries which return a set of tuples.

Database access is specified in a rule with the WRITE and RETRIEVE statements, which can appear as conditions in the LHS or as actions in the RHS. These statements return a value *true* (or *false*) on successful (or unsuccessful) execution. A RETRIEVE statement has the form:

```
RETRIEVE Arg1 Arg2
```

where *Arg1* is a quoted string representing either a filename with its path, or a loginname/password for the DBMS dataserver, as in "pharasi/mathabo". *Arg2* is an unordered list of assignments of the form:

Parameter = value(s)

Parameter can be any of the following: @TYPE, @BEGIN, @QUERY, @END, @NAME, @CURSOR, @ATOMS, @CREATE, @FIELDS, @PROP(or @SLOTS).

The meaning of these parameters are explained below:

@TYPE

@TYPE=SYBASE

File types that are supported are spreadsheet files (SYLK, WKS, NXP), RDB and SQL based relational database files (RDBMS, ORACLE, SYBASE), and flat files (DBaseIII, SYLKDB, WKSDB, NXPDB). Network, hierarchic, inverted-list and object-oriented databases are not supported.

@TYPE is a required parameter; it defaults to the type corresponding to the file extension in *Arg1*, or to SYLK if it cannot be determined from the filename extension.

A database bridge is selected on the basis of the @TYPE parameter; in our example it will be the SYBASE-bridge.

@BEGIN

@BEGIN="use preopdb"

The begin string is used during initialization of the dataserver. Therefore it will depend on the database type or manipulation language of the database type which is specified as the @TYPE parameter.
For SYLKDB it holds the name of a database range.
For RDB-based files it can be the string "begin transaction".
For SQL-based files any valid SQL statement(s) can be used. In our example it tells SYBASE which database to use.

@QUERY

@QUERY="classratio where class=likelihood_ratio.class
. and category=surgery.category"

A query string is only relevant to relational databases. The query functions select, insert, and update are not part of the query; they are implied by the RETRIEVE, WRITE statements. Tuples are not transferred directly between the dataserver and the database bridge; a virtual table is formed from which either the database or the object base is updated.

@END

@END="dbexit"

An end string is usually used to specify the action to be taken by the dataserver at the end of a transaction. Such an action may be to rollback or commit a transaction, or to exit the server as in our example.

@NAME

@NAME="!dbfield-1!!dbfield-2!"

This parameter is used only for grouped transactions, to uniquely identify tuples. The identifying key is the set of attributes specified in the parameter.

@CURSOR

@CURSOR=record.counter

The specification or non-specification of this integer valued parameter is an indication of the transaction type. If it is not specified a grouped transaction is assumed. If it is specified and its value is unknown an atomic transaction is assumed. If its value is non-zero a sequential transaction is assumed. In such a case the first tuple that is accessed is the one whose position is equal to the cursor value + 1. At every subsequent tuple access, the cursor value is incremented by 1; it is set to -1 when all the tuples have been accessed. A negative cursor value will cause the database access statement not to be executed at all.

@ATOMS

@ATOMS="class1"

The parameter is one or more classes/objects. It is used to restrict the query to the specified classes and/or objects. In this example only objects of class1 will be considered.

This is a required parameter for grouped transactions.

@FIELDS

@FIELDS="column-1, column-2,...column-k"

The list of fields are the database columns which will be used to form the virtual table. The order of these fields must match the order of the properties (or slots) in the @PROP (or @SLOT) parameter.

@PROP (or @SLOT)

@PROP="prop-1, prop-2,...prop-k"

(or @SLOT="slot-1, slot-2,...slot-k")

@PROP is used for grouped transactions only. Its value is an ordered list of properties of the classes/objects in the @ATOMS parameter. The slots corresponding to these class/object properties are mapped to database fields in the @FIELDS parameter by order of position. @SLOT is used for atomic and sequential transactions only. The slots in the parameter are mapped to database fields.

@FILL

@FILL = ADD (or null)

This parameter is used only in a grouped retrieval. The parameter value ADD indicates that a dynamic object must be created for each retrieved tuple. The name of the new object

is determined from the tuple by concatenating its values for the attributes that are specified in the @NAME parameter.

@CREATE

@CREATE=|class-2|

This parameter can be a class, sub-class, object, or sub-object.

Objects that are dynamically created by the query will be linked to this parameter as member objects if the parameter is a class/sub-class, or as sub-objects if it is an object/sub-object.

3.5 Database Links Limitations

NEXPERT's stated objective is to "allow a clean separation of knowledge and facts" at both the organizational and processing levels [NEU88]. Its knowledge base architecture is a manifestation of this objective. However, the Database Links tool has several limitations which, from the knowledge engineer's point of view, blur the distinction between data processing and knowledge processing. This sub-section will look at some of these limitations.

The use of a database bridge requires knowledge engineers (KE's) to have a detailed knowledge of the corresponding DBMS, i.e., **the knowledge engineers are required to be database programmers**. This means that their attention must alternately

switch from knowledge processing to data processing. This problem is compounded if the system accesses several DBMS's because then the KE will be required to know the syntax of several database languages.

DBMS support is restricted and potentially costly. The fact that hierarchic, network, and inverted-list databases are not supported limits the scope of available sources of data. To duplicate this information in a relational database would be a costly alternative. This limitation may also make it difficult to integrate a NEXPERT system with other systems.

An expert system that needs to access different DBMS's would have to acquire a bridge for each DBMS. The KE would have to be familiar with each DBMS language. This would be an undesirable cost, especially in the light of the constantly continuing rise in the cost of computer software and computer personnel.

The transaction types that are supported are limited. Only the read, amend, and append transactions are supported for non-DBMS files. Record deletions and insertions are not supported for all file types.

Sequential processing is restricted. A sequential write is tied to a sequential retrieve, i.e., a sequential write is just an update of a sequentially retrieved record. For the DBMS ORACLE, sequential writes are not supported and nested sequential reads are limited to 3 depths.

The use of defaults for certain parameters can be dangerous. If the @TYPE parameter is not supplied and the system cannot deduce it from the file extension (arg1 of WRITE/RETRIEVE), it defaults to the SYLK spreadsheet type. For grouped transactions the @NAME parameter defaults to the database attribute *name* irrespective of whether the attribute exists or not.

These arbitrary defaults may lead the inference engine to erroneous conclusions. In the case of the non-existent *name* attribute a read query will evaluate to *false*, which may be taken to mean that the required database records do not exist.

Database bridges reduce NEXPERT's control over the results of a database access. Because the bridge traps all the diagnostic messages resulting from a database access and only passes a *true/false* to the inference engine, the expert system is forced to assume that the query failed, even if the system might behave differently if it knew of other errors such as non-existent tables or a closed dataserver.

Expert systems are generally difficult to change once they are built [ROB90], and NEXPERT does not escape this difficulty. The absence of an OR operator in the LHS tends to increase the size of the system because the KE is forced to define as many rules as there are OR's in the original reasoning.

There are several factors which affect the

maintainability of a system. These factors include: module size, system knowledge by programmers, system modularity, and coupling between system modules [GUI83] [LIE81] [SOM89]. If several rules use the same database query the WRITE/RETRIEVE statement must be specified manually as many times as there are such rules. This has a potential for increasing development costs because of typing errors. This potential is further increased by the fact that some of these errors will go undetected because of the use of defaults.

The fact that the parameters of a WRITE/RETRIEVE statement are DBMS-dependent makes database bridges tightly coupled to their corresponding DBMS. If any parameter of the statement changes, all the rules which use this statement must be changed manually. This is the major limitation of database bridges especially after the system has been developed, since a database is likely to be re-organized several times during its lifetime. An increase in the size of the rule base will further compound this problem.

A database access may require the definition of extra objects (e.g the @CURSOR slot) or extra rules for control. These extra control rules may be used to ensure that a record retrieved from a text file is the required one. This causes the size of the rule base to increase by factors that have nothing to do with knowledge processing. The use of defaults make debugging difficult, since errors are not trapped at

compile time.

Tight coupling also affects the cost of system development. The advantages of structured programming techniques in decreasing the man-hour cost of system development and maintenance are well documented [GUI83] [SOM89].

NEXPERT's use of database bridges ties two unrelated functions together, viz., knowledge processing and data processing in one module. Therefore changes in the database specification will have a ripple effect in the knowledge base.

This coupling of separate functions has an effect on overall project management. The knowledge designer is required to be a database programmer, or alternatively both the knowledge designer and the database programmer must work on the same module. The first option limits the choice of knowledge designer, whilst the second risks increased costs due to human dynamics within a project team [SOM89]. Both options will move testing to a later phase, which may increase the man-hour costs of the project.

Chapter 4

A LOOSELY COUPLED EXPERT DATABASE SYSTEM

The database and knowledge base (KB) of a knowledge-based system (KBS) complement each other in representing the domain of the KBS. The database is generally stored outside the KBS and is controlled by a DBMS. Therefore integration with a DBMS is therefore an important consideration in the design of a KBS.

The importance of concurrent management of databases and knowledge bases is reflected in ongoing research on the combination of AI and DBMS technology for the development of expert database systems (EDBS) [BRO86b] [KER89].

Central to this research is the distinction between databases and KB's. There are three main views on this distinction: One view is that while KB's describe and operate on classes of objects, databases are representations of specific objects. A corollary view is that the source of knowledge is an expert and that of data is generally a clerk [WIE86]. In the second view the most important distinction is that databases require algorithms for their efficient storage and retrieval from physical machines. KB's require a semantic model for interpretation of their contents [BRO86a]. A third

view is that databases and KB's both represent knowledge about an application domain, therefore the distinction made in the first view is "shallow" [BRA86a]. Databases are seen as simple, large KB's. They contain only assertions in a canonical normal form and therefore their expressive power is limited [ISR86] [VAS86].

Brachman describes 3 orthogonal ways of looking at information in knowledge-based systems: the knowledge level, system engineer level, and symbol level [BRA86a] [BRA86b]. Of importance at the **knowledge level** is the information content of the system as seen by the user, rather than the organization and representation of this information. Levesque's view of a database and a KB as functionally equivalent abstract data types is a knowledge level view of information [LEV86]. The **system engineer** level is concerned with the organization of information to facilitate system comprehension and maintainability. An example of this level is Wiederhold's characterization of data and knowledge in terms of the objects, classes, and object instances [WIE86]. The **symbol level** is concerned with efficiency and integrity issues like data structures and inference strategies. Brodie's emphasis on the requirement for a computational model to manage a database and a semantic model to manage a KB is an emphasis of the symbol level [BRO86a].

In spite of these differences in the characterization of

the distinction between data and knowledge, it is generally agreed that the distinction is necessary for the design and organization of expert database systems [BRO86b] [WIE86].

The object of KBS-DBMS research is to increase the functionality of the EDBS. Research strategies on expert database systems have focused on three approaches towards EDBS architecture:

1. An existing relational DBMS is enhanced with inferential capabilities. The disadvantage of this approach is that the implementation of complex logic in a database is difficult. For example, the order of clauses in PROLOG has a semantic content that may be difficult to represent in a relational database.

A related approach is to enhance an existing logic programming system with DBMS functionality. Several projects have implemented this design using the programming language PROLOG or its dialects, with much of the work concentrating on recursive queries [IOA89] [VAS86]. This approach involves the rewrite of a large portion of the database.

Systems using the enhancement design approach do not make a distinction between the program and DBMS. They are therefore said to be tightly coupled.

2. Most EDBS's employ an interface to couple the KBS to a DBMS. The degree of coupling can be tight or loose. Vassilou

makes this distinction between tight and loose coupling: "..this strategy (of interfacing) can be further divided depending on the degree of coupling, **loose** and **tight**, corresponding respectively to a **static** and **dynamic** use of the communication mechanism." [VAS86]. Dynamic use of the interface occurs during the inference process; static use occurs before the inference process begins [BRA86b] [VAS86]. The predominant view of a loosely coupled system is one where the KBS dynamically accesses the database by means of a data communication channel [BRO86a] [IOA89] [STO89]. The KBS makes calls to the dataserver in the same way as any DBMS user. In this view interfacing represents loose coupling, enhancements represent tight coupling. This report will use the latter definition of loose/tight coupling.

The communication mechanism may be an extension of an expert system shell as in the case of NEXPERT Database Links, or an extension of a programming language as in the case of the system BERMUDA, in which PROLOG is coupled with IDM [IOA89].

The advantages of loose coupling are that little development effort is required since no changes are made to the two systems, the EDBS benefits from the strengths of the two technologies, and communication may be extended to several DBMS's. Most commercial expert systems can be loosely coupled to one or more DBMS's.

Although loose coupling is an easy practical solution to EDBS design, it has limitations that derive from the fact that the KB is main-memory resident [STO89]: In the case of a learning system the application must have procedures to save any rules that are updated or added during the inference process, reconciliation of several users' rule bases may be impossible since the rule base is not dynamically shared amongst them. If the database is volatile there may be an inconsistency between itself and the KBS.

3. The limitations of the first two approaches are addressed by a class of systems known as Knowledge Based Management Systems (KBMS). KBMS's are intended to integrate selected features from both AI and database systems into a single environment for the construction and management of both databases and KB's [BRO86a] [VAS86]. This approach is the main focus of current AI/DBMS research.

PREOP-DIS is based on the argument that the incompleteness of PREOP's system specifications necessitate a prototype system that is flexible and easy to maintain, that facilitates modular, incremental development and has control over all its functions.

These objectives can only be achieved when the interface between NEXPERT and its database is independent of both the DBMS and NEXPERT, instead of being embedded in NEXPERT as is the case with the NEXPERT Database Links tool. An independent

interface will provide a greater degree of loose coupling than is provided by the Database Links tool.

PREOP-DIS is an independent database interface for NEXPERT. It dynamically makes calls to the DBMS dataserver on behalf of NEXPERT, and uses NEXPERT's Application Programming Interface to communicate data to and from NEXPERT.

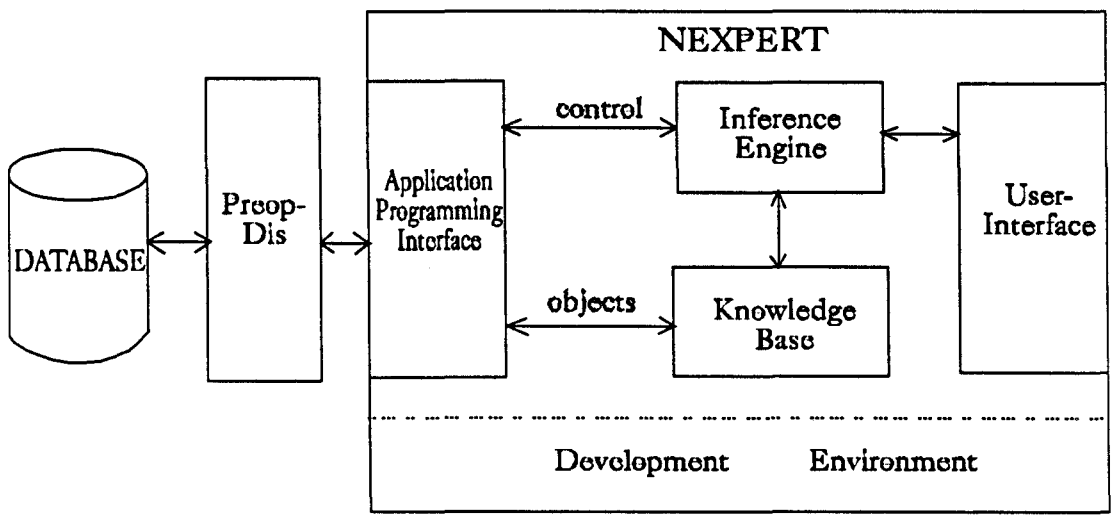


Figure 4.1 A database interface system that is independent of NEXPERT control

The architecture of PREOP-DIS has the following advantages:

It provides increased and unconstrained database support. If the interface language is judiciously chosen, the number of DBMS's that can be supported is greater or equal to the number that is supported by NEXPERT.

Query processing is dependent only on the DBMS. The limit on the permitted types of queries or the depth of query nesting are only those imposed by the DBMS dataserver.

The system is more informed about the status of any DBMS call, therefore it has increased control over communication with the DBMS. For example, if the database server is closed the system will be able to advise the user and then terminate gracefully.

Control over security of access is direct and uncomplicated. Users who for some reason are not authorized to access the database but have access to PREOP will be prevented from violating this security by the interface.

Modular system development is enhanced because the DBMS type and its and database structure become truly transparent to the inference engine. The database interface and the knowledge base can then be developed concurrently, each with a relatively independent schedule. The overlap in module development will contribute to a decrease in the cost (in man-hours) of the entire system development cycle.

The cost of system maintenance is reduced because changes in the database structure, DBMS type, server initializing process, query type (sequential, group, atomic) are confined to the interface and do not impact the knowledge base.

Chapter 5

DATABASE DESIGN

This chapter will discuss basic database concepts, data models used in database design, the design process for a relational database, and the PREOP database design based on the relational model.

5.1 Database Concepts

A software engineer views a database as a collection of related files. From a management point of view, a database is a collection of data that models an organization's activity. Both these views include support for functions which describe, query and update the database, concurrent access by several users, user interfaces that offer machine independent features, and a model of the database that makes it easy to understand the organization in question [GAR89].

A Database Management System (DBMS) is an integrated software system that efficiently manages large amounts of persistent data [ULL88]. The main objectives of a DBMS are to support correct concurrent data access by several users, data integrity, data security, reduced redundancy, data independence from physical storage considerations, and the

efficient processing of database operations [DAT86] [GAR89]. A DBMS provides a database language for users to communicate with the database. This language includes a Data Definition Language (DDL) for creating new database objects, and a Data Manipulation Language (DML) for querying or updating the database. The DDL together with the DML constitute the Data Sub-Language (DSL).

The architecture of a database system can be seen as a hierarchy of three levels: external, conceptual and internal [DAT86]. Central to this architecture is the notion of a schema, which is associated with each level. A schema is a formal description of the database in terms of objects and relationships among these objects, obtained by using a DDL for the particular level [ULL88]. A schema can be translated into the next lower schema in the hierarchy by means of a schema mapping. The three-level database architecture is used as a basis for understanding DBMS functionalities.

The **external level** is the level closest to users, who may be online terminal users or programmers. It is concerned with the way individual users view the database. Each user accesses the database via a host language, which may be a programming language or query language provided by the DBMS. The DSL is embedded in the host language. The host language provides non-database operations such as local variables and if-then-else logic. The external DSL operates on user-defined records known

as user views or external views. Each user view is defined in terms of an external schema using an external DDL. An **external schema** is a description of a part of the database corresponding to a program or a user view [GAR89].

A **conceptual view** is defined by means of a conceptual schema using a conceptual DDL. The conceptual schema is based on some data model, such as the relational model. A conceptual view is a representation of the entire information content of a database. It is a canonical integration of the different user views, i.e., it is a community view of the database. A particular user view deals with data from one or several conceptual records, or data which does not actually exist in the conceptual database but can be constructed from it. An external DDL therefore supports logical data independence between the external and conceptual schema.

The **internal level** is the level closest to the machine. It is concerned with physical considerations such as record formats, byte offsets, and file organization schemes. An internal schema is not a machine implementation of a database. For example, it assumes an infinite linear address space without consideration for pages, cylinders or tracks [DAT86]. An internal schema is a " representation dependent description of the database corresponding to a precise specification of the storage structures and access methods used to store data in secondary memory" [GAR89]. It is defined in terms of an

internal schema which is unique to the DBMS.

Both the external and conceptual DDL's support physical data independence in that a reorganization of the physical data storage does not require a change in the external or conceptual schemas; only the mapping between the internal and conceptual schema will have to be changed.

5.2 Data Models

A data model is used to represent the real world. It is a mathematical formalism with two parts: a notation for describing data, and a set of operations used to manipulate that data [ULL88]. Data models that are used in database systems include: the entity-relationship, network, hierarchic, and relational models.

5.2.1 Entity-Relationship Model

The entity-relationship (ER) model is used primarily as an implementation independent design tool for the conceptual scheme without attention to efficiency. Ullman [ULL88] describes algorithms for translating the ER model into one of the other data models in order to derive a conceptual schema. The ER model is descriptive only, and lacks a definition of data operations. It uses an ER diagram to describe the modelled world in terms of entities, attributes, keys, and relationships among the entities.

An **entity** represents an identifiable object, event or phenomenon. Entities are characterized by named properties known as **attributes**. Entities with common attributes are grouped into entity sets. Because the set of attributes which can be used to define an entity set is infinite, it is the designer's task to define the set of meaningful attributes that define a particular entity set.

A domain is the set of all values that an attribute can assume. For example, the domain for the attribute height is the set of positive integers. A **key** for an entity set is the non-empty set of attributes whose values uniquely identify each entity in the entity set.

A **relationship** among entity sets E1, E2,...Ek is a set of ordered lists such that for each k-tuple (e1, e2,...ek) in the relationship, e1 ε E1, e2 ε E3,..., ek ε Ek [GAR89]. The most common relationships are those between two entity sets, i.e., binary relationships. They can be characterized as one-to-one, one-to-many, or many-to-many relationships, depending on the number of times entities from each entity set can participate in a relationship. This characterization of relationships is an assumption the designer makes about the modelled world. The designer must therefore ensure that the ER model satisfies this assumption since it cannot be derived from an inspection of the database.

A special kind of binary relationship is the IS-A

relationship. The relationship IS-A(A,B), alternately written as A IS-A B, means that the entity set A is a special kind of the entity set B. Each entity a ε A is related to exactly one b ε B, and no b ε B is so related to more than one element of A. Some entities in B may not be related to any element of A. The purpose of IS-A relationships is so the specialized entity set A can inherit the attributes of the entity set B. The key attributes of entity set A are those of entity set B. For each (a,b) ε IS-A(A,B) the values of key attributes for entities a, b are the same. An example of an IS-A relationship is that between the entity sets EMPLOYEE and MANAGER. Not all employees are managers while every manager is an employee, and both can be identified by an employee number.

An ER diagram represents entity sets with rectangles, relationships with diamonds linked by directed/undirected edges to their constituent entity sets, and attributes with circles linked by undirected edges to their corresponding entity sets.

5.2.2 Network Model

A network model describes the world in terms of multiple occurrences of two types of objects: record types and link types. Occurrences of a particular record type have the same record format, which is made up of named fields. A link type describes a relationship between two record types, the parent

record type and the child record type.

An occurrence of a link type is an ordered list of an occurrence of the parent record type and multiple occurrences of the child record type. The occurrences of parent record types in a particular link type are also ordered. Given a particular link type L with parent record type P and child record type C, each occurrence of P is a parent in exactly one occurrence of L, and each occurrence of C is a child in at most one occurrence of L. A child (parent) record type in one link type can be a parent (child) record type in a different link type.

A network model is roughly an ER model where the relationships are restricted to binary one-to-many relationships [ULL88]. It is implemented as a chain of pointers from one parent to its children, and to the next parent in a similar way. An example of a network DBMS is IDMS™ by Cullinet Software Inc. [DAT86].

5.2.3 Hierarchic Model

A hierarchic model is a specialized network model consisting of an ordered set of tree types, which in turn consist of an ordered hierarchy of parent-child link types. A tree type consists of a root parent record type and an ordered list of zero or more child record types which are in turn sub-root parent record types. Each tree type can be regarded as a

subtree of a hypothetical system root record type which has a single occurrence, so that the entire database consists of multiple occurrences of a single tree type. An example of a hierarchic DBMS is IBM's IMS™ [DAT86].

Both the network and hierarchic models are implementation oriented, therefore the languages they support are procedural, i.e., a user can specify a data access method for the required data. Since both models are represented as chains of pointers, a record type occurrence can be identified by its address. A system which identifies objects in this manner is known as an object-oriented system.

5.2.4 Relational Model

A relational database is viewed by users as a collection of independent tables of rows and columns. Each row corresponds to a record, each column corresponds to a field. A relation is not just a flat file, it is a disciplined file where:

1. all records are of the same type,
2. columns have no particular order (left to right),
3. rows have no order (top to bottom),
4. every field is single valued,
5. all records have a unique identifier [DAT86].

Records in a relation are identified by the values of their

fields, fields are identified by their names. A system such as a relational system, which supports object identity by value and not by order, is said to be value-oriented [GAR89]. Section 5.3 will discuss the relational model in detail.

5.3 The Relational Model

The relational model was first introduced by Codd in 1970 with the objective of allowing a high degree of data independence between application programs and the internal data representation, and to provide a basis for dealing with data redundancy and inconsistency [GAR89].

5.3.1 The Concept of Relation

The relational model is based on the mathematical concept of **relation**, which is a named subset of the Cartesian product of a finite list of domains. A **domain** is a set of values specified by a name. The **Cartesian product** D1 x D2 x...x Dk of domains D1, D2,...,Dk is the set of k-tuples (t1, t2,...,tk) such that t1 ε D1, t2 ε D2,..., tk ε Dk. The list of domains is not necessarily unique. The relations that are considered for database purposes are finite subsets of the Cartesian product of some finite list of domains. The domains that are considered in the relational model are sets of atomic values; set values are not allowed.

The **arity or degree** of a relation is the number of

domains, not necessarily unique, participating in the corresponding Cartesian product.

The members of a relation are known as **tuples**. The tuples of a relation can be viewed as rows of a table, and their components as columns corresponding to exactly one domain of the associated Cartesian product. The columns are identified by names and not by their order in the tuple. A named column of a relation is known as an **attribute**.

Because a relation is a set, each tuple in a relation is unique. A **candidate key** of a relation is the non-empty, minimal set of attributes whose values uniquely identify all tuples in the relation. This means that a candidate key has no proper subset which is also a candidate key. The existence of a candidate key is guaranteed since the set of all attributes uniquely identifies all tuples in a relation. A relation can have several candidate keys. One of these is arbitrarily called the **primary key**, the rest are called **alternate keys**. An attribute that participates in a primary key is a **prime** attribute, all other attributes are nonprime. A **foreign key** in relation R1 is the set of attributes of R1 which serves as a primary key in some relation R2. The choice of a key is determined by the designer.

A **relation schema** is a structure which describes a relation in terms of the relation name and attributes. A relation schema has the form:

REL(A1, A2,..., Ak)

where REL is the relation name and the Ai's are the attributes. Prime attributes are usually underlined or written in bold.

5.3.2 Integrity Rules

A null value is used in relations to represent an unknown or inapplicable attribute. This implies that all domains in a relational system include the null value.

The **referential integrity** rule states that given two relations R1, R2 where the primary key K of R1 is a foreign key in R2, each value of K in R2 must either exist in R1 or be completely null. The presence of a foreign key in the relation R2 can be understood to mean that each tuple in R2 refers to some tuple in R1. The referential integrity rule ensures that tuples of R1 that are being referenced actually exist.

The **entity integrity** rule states that all the attributes which participate in a primary key must have non-null values. This rule ensures that all key values are known, so that they may not be duplicated in the relation.

5.3.3 Relational Algebra

Relational algebra was introduced by Codd as a collection of unary and binary operators acting on relations and resulting in relations [GAR89]. Some of these operators can be

derived from six basic operators: UNION, DIFFERENCE, CARTESIAN PRODUCT, PROJECTION, RESTRICTION, and NATURAL JOIN.

UNION operates on two relations with the same arity and the same set of attributes, i.e., union compatible relations. The resultant relation has the same arity and set of attributes as each of the parameter relations.

t ε REL3 = UNION(REL1, REL2) <=> t ε REL1 or t ε REL2

DIFFERENCE operates on two union compatible relations and results in a relation that is a subset of the first of the parameter relations.

t ε REL3 = DIFFERENCE(REL1, REL2) <=> t ε REL1
and not (t ε REL2)

CARTESIAN PRODUCT operates on two relations and results in a relation whose attribute set is the union of the parameter relations' attribute sets with the attributes qualified by their relation names. If REL1, REL2 have relation schemas REL1(A1, A2,..., An), REL2(B1, B2,...Bm) respectively then REL3 = CARTESIAN PRODUCT(REL1, REL2) will have the relation schema REL3(REL1.A1,..., REL1.An, REL2.B1, REL2.Bm). REL3 will contain all combinations of tuples from REL1 and REL2.

PROJECTION builds a new relation from another by leaving out attributes which are not specified as parameters. If the resultant relation has duplicate tuples only one of them is kept. If X is a subset of the attribute set A of a relation

REL1, REL2 = PROJECTION(REL1, X) will have the relation schema REL2(X).

RESTRICTION builds a new relation from another by removing zero or more tuples from it based on some condition. The condition has the form: *attribute operator value* where operator is within { =, <, ≤, >, ≥, ~= }.

t ε REL2 = RESTRICT(REL1, condition) <=> condition(t) is true.

NATURAL JOIN operates on two relations with one or more attributes in common. If REL1, REL2 have relation schemes REL1(A1, A2,...An, X), REL2(X, B1, B2,...Bm) respectively the relation resulting from their natural join will have the schema REL3(REL1.A1,...,REL1.An, REL1.X, REL2.B1,...REL2.Bm) with tuples t such that if t1 ε REL1 and t2 ε REL2 and t1.X = t2.X then t = (t1.A1,...t1.An, t1.X, t2.B1,...t2.Bm) ε REL3.

5.3.4 Normalization

A poor database design, such as one that stores separate facts in one relation, may result in data redundancy and database updates that lead to data inconsistency. For example, consider a relation whose purpose is to record student course information, but also keeps student information like birthdate. For a particular student, birthdate will be repeated for every course that the student is registered for. The repeated value may be incorrectly recorded during tuple insertion. The resultant inconsistency is known as an

insertion anomaly. Deletion anomalies will occur if a student cancels all the courses but does not cancel registration. The student will be unknown to the database until he registers for a course.

The term **normalization** refers to the process of methodically restructuring the relation schema to reduce redundancy and to eliminate update anomalies. Gardarin [GAR89] defines **schema normalization** as "a phase of the (database) design process that aims at obtaining molecular relations representing, without loss of information, unique facts, concepts, or events in order to avoid data redundancy and update anomalies".

The concept of **normal form** is used to define normalization criteria that a schema must satisfy in order to eliminate certain redundancies and update anomalies. Numerous normal forms have been defined, ranging from first normal form (1NF) to fifth normal form (5NF) and maybe more [DAT86] [GAR89] [ULL88]. Most database designers normalize up to 3NF, if they normalize at all [DAT86] [GAR89]. This section will therefore discuss only 1NF, 2NF, and 3NF.

A relation schema is in 1NF if the values of all the domains are atomic. The definition of a relation guarantees that all relation schemas in a relational model are in 1NF.

Given a relation with subsets X and Y, Y is said to be **functionally dependent** on X, written X -> Y, if at any one

time the values of X uniquely determine the values of Y for each tuple in the relation. Y is said to be **fully functionally dependent** on X if there does not exist a proper subset X^1 of X such that X^1 -> Y. A full functional dependence of the form: X -> A where A is a single attribute is said to be an **elementary functional dependence**. The notion of elementary functional dependence is used to define normal forms from 2NF onwards. The terms functional dependence and dependence are used interchangeably.

The objective of 2NF is to avoid anomalies that may arise when a nonprime attribute is dependent on a subset of a candidate key. In such a case a value of the attribute will be repeated as many times as the corresponding subset value appears in different primary key values. A relation schema is in 2NF if each of its nonprime attributes is fully dependent on each of its candidate keys [GAR89].

3NF eliminates the redundancy that may arise when functional dependencies exist between nonprime attributes. A relation schema is in 3NF if it is in 2NF and X -> A implies that no subset of X is nonprime.

Boyce-Codd normal form (BCNF) was introduced to eliminate redundancy arising from functional dependencies between prime attributes, a problem that 3NF does not address. A relation schema R is in BCNF if, for any elementary functional dependence X -> A, X is a candidate key in R [GAR89].

5.4 Overview of the Database Design Process

A database schema for an organization is based on the data requirements of individual application systems, which may in turn have their own conceptual schemas. It is the database administrator's duty to integrate the individual external schemas and conceptual subschemas into a global conceptual schema which will be physically implemented to serve the entire organization.

Database design is a top-down process that proceeds from an analysis of user requirements to an internal database schema that will efficiently implement these data requirements. Gardarin [GAR89] describes the following steps in the design process:

1. Capture and abstraction of user requirements
2. Integration of subschema and external user views into a global conceptual schema
3. Normalization of the global schema
4. Optimization of the internal schema.

A step in the design process can be revisited several times to accommodate design changes that may result from the other steps; i.e., there are feedback loops between the design steps. Generally, some form of physical documentation, known as a **deliverable**, is required to mark the end of each design step.

Information about an organization's main activities, application systems that effect these activities, and overall data requirements is determined by means of interviews with users and inspection of documents. The result of this analysis is a **database requirements document** that gives a "high-level" description of objects, relationships between objects, constraints on the behaviour of the objects, and expected transactions on the objects. The requirements document is the basis of the entire design. "If the services, constraints, and properties specified in the requirements document are satisfied by the design then that design is acceptable" [SOM89].

An understanding of all objects and their functions is necessary to derive the global conceptual schema from individual views and ER diagrams without contradicting constraints from individual views. A global ER diagram is used to construct relation schemas and their related candidate keys based on a data model that is appropriate for the organization.

Normalization theory is applied to the conceptual schema to avoid update anomalies through the reduction of redundancy and potential inconsistencies. The normalized conceptual schema is translated into an internal schema based on the DBMS that will be used. A database table is an internal equivalent of a relation. Internal schemas that are based on the same

conceptual schema but use different DBMS's will be different.

An analysis of the transactions that the database must support may indicate that several facts that are frequently accessed together are kept in separate relations. Accessing these facts may require a JOIN operation, which is one of the most expensive operations (in time and memory) in database systems. **Denormalization** is the process of optimizing the internal schema by combining several database tables into one without losing dependence constraints from each of the tables. Performance requirements may also require changes in the choice of a primary key for some relation.

5.5 Database Design for the PREOP Database

The design of the PREOP database is based on the relational model. This section will present a "high-level" requirements specification of the database, an entity-relationship diagram as an abstraction of the specification, and a conceptual schema based on the entity-relationship diagram.

5.5.1 Database Requirements Specification

PREOP's function is to determine the risk of cardiac complications for a patient who is about to undergo non-cardiac surgery, and to make recommendations for the administration of medication before, during and after surgery.

This will assist consultants in general internal medicine with the assessment and management of the patient.

The determination of cardiac risk is modelled on a similar study conducted by Detsky et al [DET86] at the Toronto General Hospital. This database specification is therefore based on the study by Detsky et al, guided by the text of the system proposal for PREOP [RAM89] and discussions between the author and K.Langton who is a member of PREOP's project team.

Non-cardiac surgical procedures are divided into **major** and **minor** categories. For example, intraperitoneal surgery is a major surgical procedure, removal of an eye cataract is a minor procedure.

Cardiac complications are divided into two kinds: severe and serious. **Severe complications** include cardiac death, myocardial infarction, and alveolar pulmonary edema. **Serious complications** include congestive heart failures without alveolar pulmonary edema, and coronary insufficiency in combination with any of the severe complications.

Several patient conditions (past and present), known as risk-factors, contribute to the risk that a patient will develop a disease. Risk-factors for cardiac complications include: coronary artery disease, emergency operation, and patient age over 70. A predictive index, the modified multifactorial index (mmi), is used to associate scores with each risk-factor. An **mmi-score** for a risk-factor represents

the degree to which the risk-factor contributes to the risk of cardiac complications, on a score of 0 to 20. The total mmi-score for all risk-factors in an assessment is classified into a **risk-class** according to the range in which it falls (see Table 5.2).

In order to discuss the formulas used to determine the probability of cardiac complications the concepts of positive and negative test result, pretest probability, odds, and likelihood ratio will be introduced.

The purpose of a test is to determine the presence or absence of some condition. A test result is **positive** if the condition is established, otherwise it is **negative**.

The statistical data that is used in a predictive test is usually based on results from similar tests on a sample group of patients, the test group. A **pretest probability** is the incidence rate of positive test results in the test group. The **odds** that an event will occur is a ratio of its probability over the complement of probability. An event with an odds value of x is x times more likely to occur than it is likely not to occur.

A test **sensitivity** is the incidence rate of correct positive test results in the test group, **specificity** is the incidence rate of correct negative test results. A **likelihood ratio** is a ratio of sensitivity over the complement of sensitivity. A likelihood ratio of x means that a positive

test result is x times more likely to be correct than it is likely to be incorrect [SAC85]. A likelihood ratio can be seen as a measure of the predictive value of a pretest probability.

Two tables, one containing pretest probabilities and the other likelihood ratios, are used to determine the risk of cardiac complications. The table of pretest probabilities contains probability scores for each kind of complication (severe, serious) per surgery procedure. In the table of likelihood ratios each surgery category has a ratio for each risk-class.

The results of the assessment will be stored in a database to be available for test validation and for later research work. Hard copies of these results will be in a form that will facilitate their integration with existing medical records.

The database design must take into account guidelines for computerized medical records as set by the College of Physicians and Surgeons of Ontario, that: confidentiality of patient information must be protected, there must be a feature which identifies the origin and time of all record additions and modifications [CPS88].

Table 5.1 Pretest probabilities for selected surgery types. Adapted from [DET86]

	Severe	Serious
Aortic vascular	.1320	.2100
Intraperitoneal	.8096	.1255
Cataracts	.1600	.2100

Table 5.2 Likelihood ratios per surgery category per risk-class. Adapted from [DET86]

Risk Class	mmi-Range	Major Surgery	Minor Surgery
I	0 - 15	0.42	0.39
II	15 - 30	3.58	2.75
III	30+	14.93	12.20

$$\text{odds} = \frac{\text{probability}}{1 - \text{probability}}$$

$$\text{probability} = \frac{\text{odds}}{\text{odds} + 1}$$

$$\text{posttest odds} = \text{pretest odds} \times \text{likelihood ratio}$$

$$\text{posttest probability} = \frac{\frac{\text{pretest probability}}{1 - \text{pretest probability}} \times \text{likelihood ratio}}{\left(\frac{\text{pretest probability}}{1 - \text{pretest probability}} \times \text{likelihood ratio}\right) + 1}$$

Figure 5.1 Bayesian formulas used to calculate the posttest probability for cardiac complications. Adapted from [SAC85]

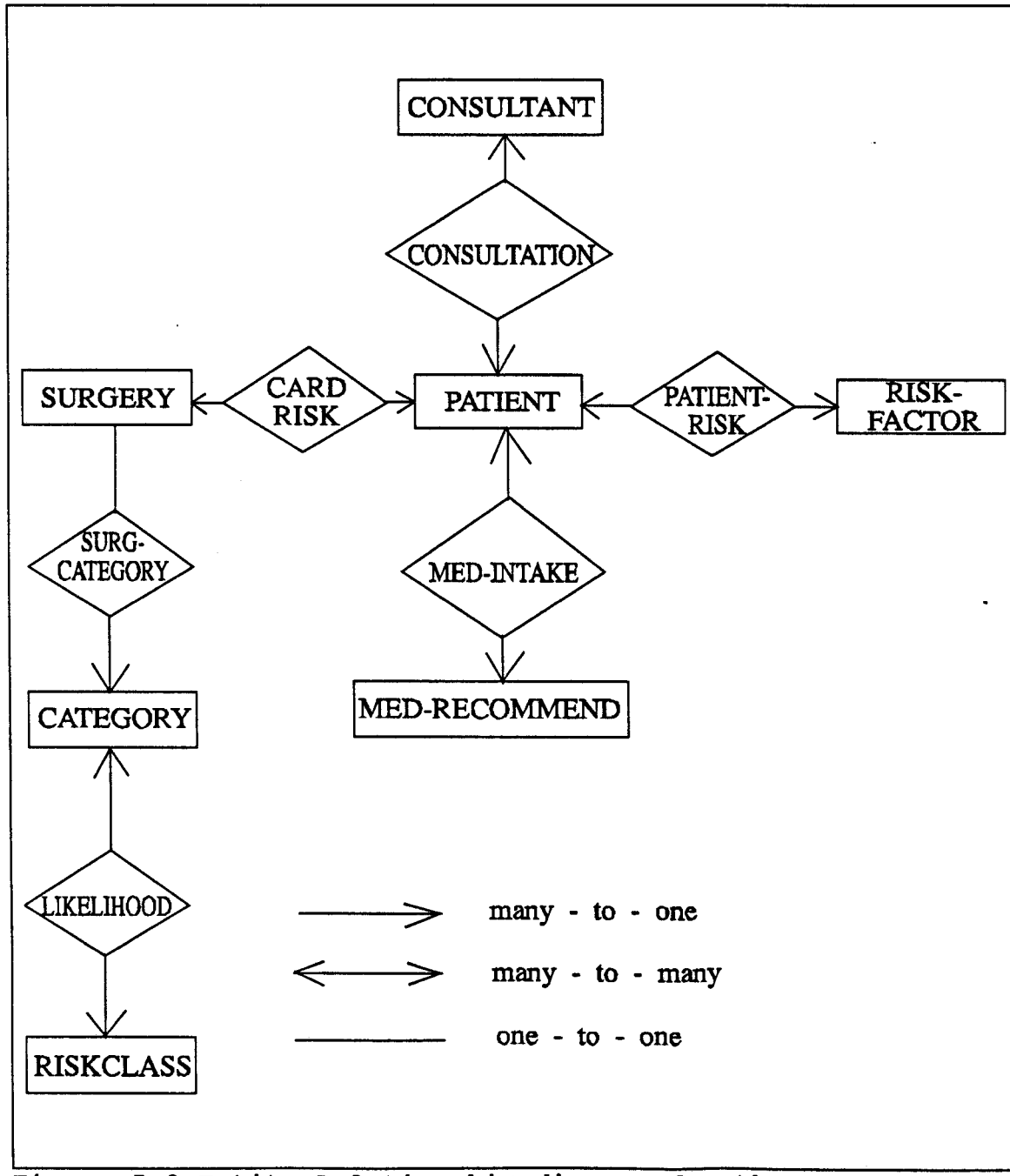


Figure 5.2 Entity-Relationship diagram for the PREOP database.

Table 5.3 Relation schemas for the PREOP database

PATIENT	(pat-id, pat-name, pat-sex, pat-age)
CONSULTANT	(consultant-id, consultant-name)
SURGERY	(surgery-type, date, time, severe, serious)
CATEGORY	(category-name)
RISKFACTOR	(risk-name, date, time, weight)
RISKCLASS	(class, date, time, lowweight, highweight)
MED-RECOMMEND	(medication,condition,date,time,recommend)
CONSULTATION	(pat-id, date, time, consultant-id, ref-id)
PATIENT-RISK	(pat-id, date, time, risk-name)
CARDRISK	(pat-id, date, time, surgery-type, mmi, severe, serious)
MED-INTAKE	(pat-id, date, time, medication, condition, recommend)
SURG-CATEGORY	(surgery-type, category)
LIKELIHOOD	(class, category, date, time, ratio)

The relation schema in Table 5.3 are all in Boyce-Codd normal form (see section 5.3.4). For all the relation schemas, all nonprime attributes are fully functionally dependent on

the primary key, and the only dependencies are those involving the primary key. The relation schemas are based on the following assumptions:

All transactions which update a relation whose primary key has a date or time attribute will not physically update the old record, but will instead create a new record. This new record will include effective date and time.

A consultant can assess one patient at a time, and only one consultant is in charge of an assessment even if several consultants may be involved.

A surgery category is either major or minor. The category associated with each surgery type will never change.

5.5.2 SYBASE Database Tables

The database is implemented on the SYBASE DBMS (see chapter 6), with a few deviations from the relation schema.

For the specific purposes of this prototype, all data used as input in the calculation of posttest probability (rates, ratios, weights, weight-ranges) is regarded as non-volatile. Therefore the date, time attributes are not implemented for database tables which keep this information. Only the output tables contain a time feature.

Because of the assumption that a surgery-type will never have its category changed, the CATEGORY, SURGERY-CATEGORY relations are not implemented; instead category is a field in

the SURGERY table. The system administrator will therefore be responsible for ensuring consistency between the SURGERY and LIKELIHOOD tables.

The RISKFACTOR relation is implemented as part of the PREOP object base. The relation is represented as a class with the risk factors as member objects and risk weight as an inheritable property.

Information from the pretest probability and likelihood ratio tables is duplicated in the assessment output tables to facilitate reporting. (This redundancy may be removed later when a reporting functionality is added to PREOP.) Only surgery category is not duplicated because it is assumed to be a permanent attribute of surgery type. For all output tables, records belonging to one assessment will have the same values for the fields: patient-id, date, and time.

Table 5.4 SYBASE tables for the PREOP database

PATIENT	pat_id 9(8) not null pat_name varchar(30) not null pat_sex char(1) not null pat_birthdate 9(2) not null
CONSULTANT	consultant_id varchar(30) not null consultant_name varchar(30)
SURGERY	surgery_type varchar(30) not null category varchar(15) not null severe_pretest float not null serious_pretest float not null
RISKCLASS	class varchar(9) not null lowweight int(3) not null highweight int(3) not null
MED_RECOMMEND	medication varchar(30) not null condition varchar(30) not null recommend text
LIKELIHOOD	class varchar(9) category varchar(15) ratio float not null
CONSULTATION	pat-id char(8) not null date char(8) not null time char(6) not null consultant_id varchar(30) not null ref-id varchar(30)
CARDRISK	pat_id char(8) not null date char(8) not null time char(6) not null surgery_type varchar(30) not null class varchar(9) not null mmi int not null likelihood_ratio float not null severe_pretest float not null serious_pretest float not null severe_posttest float not null serious_posttest float not null

Table 5.4 SYBASE table structures for the PREOP database (cont.)

PATIENT_RISK	pat_id char(8) not null date char(8) not null time char(6) not null risk varchar(255) not null
MED_INTAKE	pat_id char(8) not null date char(8) not null time char(6) not null medication varchar(30) not null condition varchar(30) not null recommend text not null

Chapter 6

PREOP-DIS IMPLEMENTATION

PREOP-DIS is implemented as a C program that communicates data between PREOP and a SYBASE database by dynamically making calls to NEXPERT's Application Programming Interface (see sections 3.3) and to the SYBASE dataserver.

This chapter discusses the choice of hardware, programming language, and DBMS for the implementation of PREOP-DIS. Section 6.2 describes the communication between PREOP-DIS and PREOP. Section 6.3 describes the communication between PREOP-DIS and the SYBASE dataserver. Section 6.4 gives an overview of the application flow.

6.1 Hardware Environment

PREOP-DIS is implemented on the VAX VMS mainframe. This choice over a PC environment or a mixed PC/mainframe environment was made because:

The VAX is available to a large, distributed user community including project team members and PC owners. A PC or workstation can be connected to the mainframe via a network.

A large part of the hospital information systems is

implemented on the VAX; therefore using it for PREOP-DIS will facilitate its integration with these systems. Also, patient demographic information, which is part of PREOP's domain, is stored on the VAX.

It is not unreasonable to assume that the final PREOP system will be implemented on the VAX. Therefore chances of system migration problems are greatly decreased if the development environment is the same as the production environment.

The use of PC's would require that there be as many copies of PREOP as there are PC-users. This would make it difficult to maintain these copies and to maintain consistency amongst them.

It is easier to enforce security control on the system if there is one copy of the system and one access path to data.

6.2 Programming Language

Amongst all the programming languages that are supported by both VAX VMS and NEXPERT, viz. C, FORTRAN and Pascal, C was the best choice for PREOP-DIS because it is highly portable across operating systems and application systems, it can access several of the major DBMS's, and it can access NEXPERT with ease since NEXPERT is just a collection of C functions.

The use of one programming language for all the components of a system makes it easy to develop, debug and

maintain the system.

The expert system PREOP is embedded in PREOP-DIS by means of the C version of the NEXPERT Application Programming Interface. PREOP accesses PREOP-DIS by means of the EXECUTE statement in the rule base. PREOP-DIS accesses PREOP by executing a set of NEXPERT functions which are summarized below. A full description of these functions is found in section 3.3.

1. **NXP_Control(NXP_CTRL_INIT)** initializes NEXPERT's working memory.

2. **NXP_SetHandler(NXP_PROC_EXECUTE, ext-name,"nxp-name")** identifies to NEXPERT a PREOP-DIS function that will be executed from NEXPERT by means of the EXECUTE statement. There are six NXP_SetHandler() calls, each corresponding with one of the six PREOP-DIS functions that are called by PREOP.

3. **NXP_SetHandler(NXP_PROC_QUESTION, QuestioHandler, 0)** identifies to NEXPERT the PREOP-DIS function which NEXPERT will use to prompt a PREOP user for input. A sample of PREOP's dialogue with the user is shown in appendix B.

4. **NXP_SetHandler(NXP_PROC_ALERT, AlertHandler, 0)** identifies to NEXPERT the PREOP-DIS function which NEXPERT will use to notify a PREOP user of any NEXPERT system errors.

5. **NXP_LoadKB("KB-name", KB-address)** loads the knowledge base with filename *KB-name* at the address pointed to by *KB-*

address. There are three such NXP_LoadKB() calls, each corresponding with one of PREOP's three knowledge base files.

6. **NXP_Control(NXP_CTRL_KNOWCESS)** starts the inference engine. Execution of this function passes program control to NEXPERT until the inference session terminates.

7. **NXP_Volunteer()** is used to update PREOP's object base.

8. **NXP_GetAtomInfo()** interrogates PREOP's object base.

9. **NXP_Control(NXP_CTRL_EXIT)** exits NEXPERT and de-allocates its working memory.

6.3 Database Environment

PREOP's database is implemented as SYBASE tables. SYBASE is an SQL-based relational DBMS which is written in C [SYB89]. Transact-SQL™, a version of the SQL database language, is used to send commands to the SYBASE server to interrogate or manipulate the database.

Although SYBASE has an integrated development environment, users are not restricted to this environment for access to the server. DB-Library/C™ is a set of C routines and macros that allow an application to access the server. There is a DB-Library for each programming language that is supported by SYBASE. SYBASE therefore has an open architecture.

Unlike other SQL-interfaces such as the Ingres-C interface, DB-Library is not embedded in C and therefore does

not require a pre-compiler. DB-Library routines can be called from an application like any C routines.

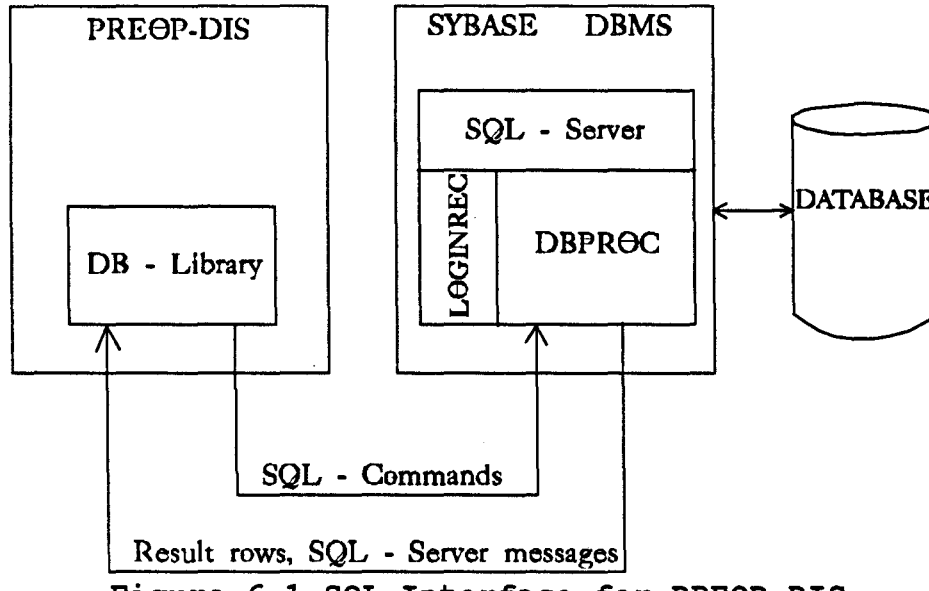


Figure 6.1 SQL-Interface for PREOP-DIS

A special data structure, the DBPROCESS, is the vehicle for communication between DB-Library routines and the server. It is a set of pointers pointing to buffers for data such as Transact-SQL queries, result rows of a query, server error messages, etc. These data can only be accessed via DB-Library routines.

Another important structure is the LOGINREC, which contains typical login information such as username and

password. This information is used by DB-Library routines when sending commands to the server.

PREOP-DIS accesses the SYBASE dataserver by means of the following DB-Library routines:

1. The routine **dbinit()** is used to initialize DB-Library. The routine **dblogin()** sets up the LOGINREC and logs the application to the server. It returns a pointer to LOGINREC, which is used as a parameter in **dbopen()** to allocate a DBPROCESS to the LOGINREC.

2. User-defined procedures to handle DB-Library error messages and server messages can be declared to DB-Library with the routines **dberrhandle()** and **dbmsghandle()**. This step is not essential since DB-Library will use its own error handling procedures if it is bypassed.

3. At this stage the application is ready to send Transact-SQL commands to the server. The **dbcmd()** routine is used to send commands to the command buffer. Several commands can be accumulated in the buffer before being sent to the server by the routine **dbsqlexec()**. The commands are executed in the order in which they were sent to the buffer.

4. To process the results of a command the routines dbresults(), dbbind() and dbnextrow() are executed in order. **Dbresults()** prepares the result buffer in DBPROCESS for processing. **Dbind()** associates a column in the result row with

a program variable. **Dbnextrow()** allocates the values of the columns to their associated program variables for processing by the application. Dbnextrow() can be executed repeatedly until no more rows are found in the buffer.

5. The last routine is **dbexit()**. It closes the connection to the server.

6.4 Application Flow

PREOP is executed in the PREOP-DIS process environment (see sections 3.3, 6.2). For all its communication with the user PREOP uses the PREOP-DIS QuestionHandler function which displays a PREOP prompt string on the screen and volunteers the user's response into the PREOP object base.

A database access is initiated when the inference engine executes a PREOP-DIS function that is specified in an EXECUTE statement in PREOP's rule base. The PREOP-DIS function uses local variables to read or update the database, based on object base parameters specified in the EXECUTE statement. After a successful database read PREOP-DIS updates the object base by executing the NEXPERT NXP_Volunteer() function. At the end of a PREOP inference session PREOP-DIS stores the inference results in SYBASE tables.

The following steps give an overview of a PREOP application flow:

1. PREOPD-DIS initializes NEXPERT, identifies to NEXPERT all the PREOP-DIS functions that will be executed by PREOP, and initializes the SYBASE dataserver. It then reads the CONSULTANT table using the VMS system-user parameter as a key to check if the user is a valid user. The full name of a valid user will be used for output purposes. On an unsuccessful read the system terminates. The three knowledge base files that PREOP requires are then loaded into NEXPERT.

2. PREOP-DIS prompts the user for the patient-id, which is used to read patient name, sex and birthdate from the PATIENT table. Current patient age is calculated from birthdate. Patient information is then volunteered into the PREOP object base. If the patient-id is not found the user is prompted to input patient name, sex, and age.

At this point PREOP is ready to be executed. PREOP-DIS suggests the start hypothesis to PREOP and passes control to NEXPERT by executing the NXP_Control(NXP_CTRL_KNOWCESS() function.

3. PREOP prompts the user for the type of surgery that the patient is about to undergo. PREOP then passes this surgery type as a parameter to a PREOP-DIS function which uses it to read surgery category and pretest probabilities from the SURGERY table.

PREOP prompts the user for patient information which it requires to determine risk factors associated with cardiac

complications. In the object base a score is associated with each risk factor. All the scores for risk factors which PREOP establishes for the patient are accumulated into a single score, the mmi-score (see section 5.5.1). This mmi-score is passed as a parameter to a PREOP-DIS function which uses it to read an associated riskclass from the RISKCLASS table. Riskclass and surgery category are then passed as parameters to a PREOP-DIS function which reads the LIKELIHOOD table for the likelihood ratio associated with these parameters. PREOP then uses the formulas in Figure 5.1 to determine the probability of cardiac complications. A PREOP-DIS function saves the posttest probability and all the data which was used in its calculation in local variables. This data will later be saved in the CARDRISK output table.

4. PREOP prompts the user for all the medication that the patient is taking. For each medication, the user is prompted to supply further information which PREOP uses to determine associated condition codes (see section 6.5.1). The medications and their condition codes are passed to a PREOP-DIS function which reads the MED_RECOMMEND table for the corresponding recommendations.

5. At this point the PREOP inference session terminates and control is passed back to PREOP-DIS, which then exits NEXPERT. PREOP-DIS saves all output data to the CARDRISK, CONSULTATION, PATIENT_RISK, and MED_INTAKE tables. A PREOP

report is displayed on the screen and a copy of this report is appended to a text file (see Appendix B). PREOP-DIS exits the SYBSASE dataserver and execution stops.

Chapter 7

PROJECT DISCUSSION

Chapter 4 discussed the advantages of choosing PREOP-DIS over NEXPERT's Database Links utility to implement a database interface for the PREOP expert system. These advantages are: flexibility of the database interface, enhanced integration with other information systems, system modularity, and ease of maintenance. This chapter will discuss modifications done to the original PREOP knowledge base in the process of implementing PREOP-DIS, the implication of these modifications for knowledge engineering, consistency between the database and the knowledge base, and possible future changes to PREOP-DIS.

7.1 Impact on Knowledge Base Design

The development of PREOP-DIS and the resultant changes to the PREOP knowledge base (KB) highlighted aspects of database and KB design which suggest that the task of database specification and design is a sub-task of knowledge acquisition and representation, and that during the knowledge engineering process database design must precede KB design.

This view of the database as a form of knowledge base is

defined by Brachman [BRA86a] as a knowledge level view of the database. Of importance at the knowledge level is the information content of the system, what it can tell the user about the domain, and not how this information is represented.

A database specification document, as part of a systems requirements document, is based on interviews with domain experts and information from relevant literature. Database specification is therefore a knowledge acquisition task and database design a knowledge representation task. The database describes the structure of the domain, whilst the rule base describes the behaviour of this structure. Database design must therefore precede rule base design. The development of the rule base may require changes to the database design, which may in turn cause changes to some parts of the rule base. This means that there is a feedback loop between database design and rule base design.

For systems which have an object base or an equivalent internal database, the structure of the object base must follow that of the database, and not vice versa. Unlike object bases, databases have well-established design methodologies and serve a large user community. Databases are therefore generally subject to site standards which the object base is not guaranteed to observe. The requirement that the object base follow the database is therefore necessary for the maintenance of site standards and for system integration. This

requirement is also reasonable since the object base is primarily a representation of the database.

This project shows that designing the object base after the database makes the object base and rule base more understandable, facilitates communication with the database, and facilitates knowledge processing. To illustrate this, two of the major changes to the original PREOP knowledge base will be described.

In the old KB, risk factors are represented as boolean properties of the object risk. These properties are then used as conditions in rules whose purpose is to accumulate the mmi scores associated with each risk factor into a total mmi score. An example of such a rule is

```
IF risk.poor_general_status    THEN mmi_assignment
                                 do  mmi = mmi + 5
```

The representation of a risk factor as a property not only hides its structure as an object with properties of its own, but burdens the affected rules with unnecessary data processing by storing the mmi score of each risk factor in a rule. An important disadvantage of this representation is that it is impossible to save the risk factors which have been established in an assessment without checking and then saving them one by one. NEXPERT has no way of processing properties of an object as a group as it does with member objects of a class.

With the implementation of PREOP-DIS the risk factors are represented as objects of the class risk_factors. Each member object has the properties Value, flag, weight. Both Value and flag are boolean properties which are set to true when the risk factor has been established. Weight represents the mmi score associated with the risk factor. The class risk_factors is an implementation of the relation RISK_FACTORS from the database design. Flag is an artificial property which is used to process risk factors as a group. The invention of the property flag is necessary because NEXPERT does not allow group processing on the property Value, i.e., a term such as <|risk_factors|>.Value is invalid. An example of the modified rules is:

```
IF poor_general_status THEN mmi_assignment
```

The following rule accumulates the mmi scores:

```
IF <|risk_factors|>.flag

THEN mmi_assignment
  do assess.mmi = SUM(<|risk_factors|>.weight)
```

This rule means that if there exists at least one object in the class risk_factors whose flag is true, then accumulate the weights of all such objects into the slot assess.mmi.

The advantages of this new structure for risk factors are the following: it facilitates a comprehension of the domain

structure; it decreases the size of affected rules; a change in the weight value is restricted to the object base and does not impact the rule base; it facilitates knowledge processing.

A second major change is in the elimination of the classes likelihood_ratios and surgery_types, which are used to duplicate the entire LIKELIHOOD and SURGERY database tables in the object base. These tables are used in the determination of the likelihood ratio associated with a surgery category, mmi class pair. An example of such a rule is:

```
IF surgery_determined
and mmi > 15
and mmi <= 30
and surgery.category = "minor"

THEN likelihood_ratio_determination
  do likelihood_ratio.class = "class_II"
   do   likelihood_ratio.ratio   =
                  \likelihood_ratio.class\.minor
```

There are 6 such rules, to cater for each combination of 2 surgery categories and 3 mmi classes. These rules are difficult to understand and are vulnerable to changes in the database. In the modified knowledge base the classes likelihood_ratios and surgery_types are replaced by the objects assess and surgery respectively. The 6 rules are replaced by the following 2 rules:

```
IF surgery_determined
and   Execute   "GetClass"   @ATOMID=assess.mmi,
                          assess.mmi_class
```

```
THEN likelihood_class_determined

IF likelihood_class_determined
and Execute "GetProbRatio" @ATOMID=assess.mmi_class,
          surgery.category, assess.likelihood_ratio

THEN likelihood_ratio_determined
```

The advantages of this modification of the knowledge base are that: There is a decrease in the size of the rule base, the affected rules are easy to read, and changes to the database do not impact the rule base.

7.2 Consistency Between the Database and Knowledge Base

A data communication mechanism between two independent systems must address the question of data consistency between the two systems. In the case of a database interface for an expert system the maintenance of consistency is difficult, if not impossible, because the two systems use different technologies to manage data. Several DBMS/AI researchers are investigating a class of future systems known as Knowledge Based Management Systems (KBMS), which will provide a complete environment for the construction and access of both the database and knowledge base [KER89] [BRO86b]. KBMS's are regarded as a long-term solution to interface problems such as inconsistency between the database and knowledge base.

Both static and dynamic database interfaces have a

potential for data inconsistency because the database can be updated at any time during an inference session. For a static interface, the potential for using outdated data is higher than that for a dynamic interface, but there is no risk of inconsistent analysis because all the required data is loaded before the inference session begins. While the use of a dynamic interface eliminates the problem of changing the KB with every change in the database, it increases the risk of inconsistent analysis. For example, suppose that several related database tables or records of the same table have to be accessed to complete a computation. If data that has already been accessed is updated before the computation is complete the KB might end up with wrong information.

As with all dynamic database interfaces, PREOP-DIS has to address inconsistencies that may be caused by database updates during the process of an inference session. PREOP-DIS attempts to control such inconsistencies at the level of database administration. All users except the database administrator (DBA) are denied update permission on all tables; the only tables where users have insert permissions are the output tables. The onus lies with the DBA to enforce these database access permissions, and to ensure that database updates do not overlap with the use of PREOP.

The PREOP KB has two instances of a static database coupling which PREOP-DIS cannot resolve. The first instance is

in the use of the IS statement to literally specify valid choices for surgery type, as in the statement:

```
IS surgery.type "vascular", "intrathoracic", "orthopedic"
```

NEXPERT will suggest the list of choices whenever it prompts the user for the value of surgery.type. The list automatically becomes part of the object base at compile time, and therefore cannot be dynamically specified. The knowledge engineer must ensure that the list represents all the surgery types in the database. The structure of an IS statement therefore increases the risk of inconsistency and frequent maintenance.

The second instance of static coupling is where a medication code is literally specified in order to determine appropriate recommendations for medication intake. The medication name and code together constitute the key in the database table MED_RECOMMENDS. A code is used to identify different recommendations for the same medication, based on the health conditions of a patient. For example, the following two rules will result in different medication recommendations:

```
IF patient_medications.name is "antidiabetic"
and diabetic.name = "chlorpropamide"

THEN recommendation_done
  do patient_medications.condition = "diachl"


IF patient_medications.name = "antidiabetic"
and diabetic.oral_intake
```

```
THEN recommendation_done
  do patient_medication.condition = "diainsm"
```

Factors that determine a medication code are so numerous and potentially infinite that it is impossible to represent the relationship between a medication code and factors that determine it in a database. Such a relationship is statically specified in the rule base. Therefore the knowledge engineer must ensure that the medication codes in the rule base correspond to those in the MED_RECOMMENDS database table.

7.3 Possible Future Changes to PREOP-DIS

PREOP-DIS is implemented as a control module which has a database interface and user interface functionality. As PREOP is further developed, a separate module might take over the control and user interface functionality from PREOP-DIS without affecting the database interface functionality.

Some of the database tables, such as the LIKELIHOOD table, are local to PREOP and are not used by the hospital information systems. Since HIRU and Chedoke-McMaster hospital are separate institutions, it might not be possible in the early life of PREOP to keep its local tables on the hospital SYBASE database. In that event flat files can be substituted for SYBASE tables in the affected procedures.

Conclusion

The design objectives of PREOP-DIS are to facilitate overall system maintenance, integration, modularity, and flexibility for the PREOP expert system. PREOP-DIS has shown that these objectives are best achieved by an independent, loosely coupled database interface, instead of one that is embedded in the knowledge base.

Although the investigation of knowledge engineering methodologies was not part of this project's mandate, the experience gained from the project has implications for knowledge engineering. Work on PREOP-DIS has shown that database specification and design is a knowledge engineering process, database and knowledge base specification derive from the same knowledge acquisition process, rule base design must be based on database design and may in turn cause modifications in the database design, and the object base structure must follow the database structure. This means that the knowledge engineer's view of the domain must proceed from the knowledge level during the knowledge acquisition phase, to the system engineer level during the knowledge representation phase.

REFERENCES

[BLA90] Blackman, M.J. "CASE For Expert Systems" in *AI EXPERT*, No. 2 (Feb. 1990), pp. 27-31.

[BRA86a] Brachman, R. Levesque, H. "The Knowledge Level of a KMBS"in *On Knowledge Base Management Systems.Topics in Information Systems*, Springer-Verlag New York Inc.: New York, NY (1986), pp. 9-12.

[BRA86b] Brachman, R.J. Levesque, H.J. "What Makes a Knowledge Base Knowledgeable? A View of Databases From the Knowledge Level" in *Expert Database Systems: Proceedings from the First International Workshop*, The Benjamin/Cummings Publishing Company Inc.: Menlo Park, CA (1986), pp. 69-78.

[BRO86a] Brodie, M.L. Mylopoulos, J. "Knowledge Base vs Databases" in *On Knowledge Base Management Systems. Topics in Information Systems*, Springer-Verlag New York Inc.: New York, NY (1986), pp. 83-86.

[BRO86b] Brodie, M. et al. "Knowledge Base Management Systems: Discussions from the Working Group" in *Expert Database Systems: Proceedings from the First International Workshop*, The Benjamin/Cummings Publishing Company Inc. (1986), pp. 19-33.

[BRU90] Bruha, I. *Introduction to Artificial Intelligence*, DCSS Course Notes, McMaster University (1990).

[BRW90] Brown, D.E. "Inference Engines for the Mainstream" in *AI EXPERT*, No. 2 (Feb. 1990), pp. 32-37.

[BUC84] Buchanan, B.G. Shortliffe, E.H. (Editors). *Rule-Based Expert Sytems. The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Publishing Company Inc. (1984).

[CAM84a] Campbell, J.A. "Three Uncertainties of AI" in *Artificial Intelligence: Human Effects*, Ellis Horwood Limited: Chichester (1984), pp. 249-273.

[CAM84b] Campbell, J.A. "The Expert Computer and Professional Negligence" in *Artificial Intelligence: Human Effects*, Ellis Horwood Limited: Chichester (1984), pp. 37-51.

[CPS88] The College of Physicians and Surgeons of Ontario. "Computerised Medical Records" in *College Notices*, No. 16 (Nov. 1988), pp. 2-3.

[DAT86] Date, C.J. *An Introduction to Database Systems, Volume I*, Addison-Wesley Publishing Company Inc. (1986).

[DEC89] Digital Equipment Corporation manual. *Guide to File Applications, VMS Version 5.0*, Digital Equipment Corporation: Maynard, Mass. (1989).

[DET86] Detsky, A.S. Abrams, H.B. McLaughline, J.R. et al. "Predicting Cardiac Complications in Patients Undergoing Non-Cardiac Surgery," in *Journal of General Internal Medicine*, (Jul./Aug. 1986), pp. 211-219.

[ELI90] Eliot, L.B. "Best Bets for Future Innovations" in *AI EXPERT*, No. 3 (Mar. 1990), pp. 9-11.

[GAR89] Gardarin, G. Valduriez, P. *Relational Databases and Knowledge Bases*, Addison-Wesley Publishing Company inc. (1989).

[GUI83] Guimaraes, T. "Managing Application Program Maintenance" in *Communications of the ACM*, No. 10 (Oct. 1983), pp. 739-746.

[HUN84] Hunter, J.R.W. "The Expert Medical Computer" in *Artificial Intelligence: Human Effects*, Ellis Horwood Limited: Chichester (1984), pp. 26-36.

[IOA89] Ioannidis, Y.E. Chen, J. Friedman, M.A. Tsenganis, M.M. "BERMUDA - An Architectural Perspective on Interfacing Prolog to a Database Machine" *Proceedings from the Second International Conference on Expert Database Systems*, The Benjamin/Cummings Publishing Company, Inc.: Redwood City, CA (1989), pp. 229-255.

[ISR86] Israel, D. "AI Knowledge Bases and Databases" in *On Knowledge Base Management Systems. Topics in Information Systems*, Springer-Verlag New York Inc.:

New York NY (1986), pp 71-75.

[KER89] Kerschberg, L. (Editor). *Proceedings from the Second International Conference on Expert Database Systems*, The Benjamin/Cummings Publishing Company Inc.: Redwood City, CA (1989).

[KEY90] Keyes, J. "Where's the 'Expert' in Expert Systems?" in *AI EXPERT*, No. 3 (Mar. 1990), pp. 61-64.

[KUZ88] Kuzmak, P.M. Walters, R.F. Penrod, G. "Technical Aspects of Interfacing MUMPS to an External SQL-based DBMS" in *Proceedings of the 12th Annual Symposium on Computer Applications in Medical Care*, (Nov. 1988), pp. 616-624.

[LEV86] Levesque, H.J. Brachman, R.J. "Knowledge Level Interfaces to Information Systems" in *On Knowledge-Based Management Systems. Topics in Information Systems*, Springer-Verlag New York Inc.: New York, NY (1986), pp. 13-17.

[LIE81] Lientz, B.P Swanson, E.B. "Problems in Application Software Maintenance" in *Communications of the ACM*, No. 11 (Nov. 1981) pp. 763-769.

[MIN68] Minsky, M. (Editor). *Semantic Information Processing*. Massachusetts Institute of Technology (1968).

[NAR84] Narayanan, A. "AI, Medicine and Law" in *Artificial Intelligence: Human Effects*, Ellis Horwood Limited: Chichester (1984), pp. 15-25.

[NEU88] NEXPERT manual. *NEXPERT OBJECT Fundamentals IBM Ver. 1.1*, Neuron Data Inc.: Palo Alto, CA (1988).

[NEU89] NEXPERT manual. *NEXPERT OBJECT Callable Interface IBM Ver. 1.1*, Neuron Data Inc.: Palo Alto, CA (1989).

[NIL80] Nilsson, N.J. *Principles of Artificial Intelligence*, Tioga Publishing Company: Palo Alto, CA (1980).

[PHI90] Phillip, G.C. Schultz, H.K. "What's Happening With Expert Systems?" in *AI EXPERT*, No. 11 (Nov. 1990), pp. 57-59.

[RAM89] Ramsden, M.F. "Development of Medical Expert Systems - Proposal Main Body," HIRU, Dept. of Clinical Epidemiology and Biostatistics, Faculty of Health Sciences, McMaster University (1989).

[ROB90] Robert, J. Jacob, K. Froscher, J.N. "A Software Engineering Methodology for Rule-Based Systems" in *IEEE Transactions on Knowledge and Data Engineering*, No. 2 (Jun. 1990), pp. 173-189.

[SAC85] Sackett, D.L Haynes, R.B Tugwell, P. *Clinical Epidemiology. A Basic Science for Clinical Medicine*, Little, Brown and Company: Boston (1985).

[SOL90] Solntseff, N. "A Framework for the Preoperative Assessment Project", DCSS Report, McMaster University (1990).

[SOM89] Sommerville, I. *Software Engineering*, Addison-Wesley Publishing Company (1989).

[STO89] Stonebreaker, M. Hearst, M. "Future Trends in Expert Database Systems," *Proceedings from the Second International Conference on Expert Database Systems*, The Benjamin/Cummings Publishing Company Inc.: Redwood City, CA (1989), pp. 3-20.

[SYB89] SYBASE Manual. *Open Client DB-Library/C Reference Manual*, Sybase, Inc.: Emeryville, CA (1989).

[ULL88] Ullman, J.D. *Principles of Database and Knowledge-Based Systems Volume I*, Computer Science Press, Inc.: Rockville, Maryland (1988).

[VAS86] Vassiliou, Y. "Knowledge-Based and Database Systems: Enhancement, Coupling or Integration?" in *On Knowledge Base Management Systems. Topics in Information Systems*, Springer-Verlag New York Inc.: New York, NY (1986), pp. 87-92.

[WIE86] Wiederhold, G. "Knowledge versus Data," in *On Knowledge Base Management Systems. Topics in Information Systems*, Springer-Verlag: New York, NY (1986), pp. 77-82.

[WIN84] Winston, P.H. *Artificial Intelligence*, Addison-Wesley Publishing Company Inc. (1984).

[YAZ84] Yazdani, M. "Introduction: What is Artificial Intelligence?" in *Artificial Intelligence: Human Effects*, Ellis Horwood Limited: Chichester (1984), pp. 9-13.

APPENDIX A

```
/*
 *                    PREOP-DIS Program Code
 *
 *   This program serves as a database interface for the PREOP
 *    expert system. It makes calls to the SYBASE dataserver
 *    and to NEXPERT's Application Programming Interface.
 */

#include "sys$common:[000000.syslib]stdio.h"
#include "sys$common:[000000.syslib]string.h"
#include "sys$common:[000000.syslib]ctype.h"
#include "sys$common:[000000.syslib]stdlib.h"
#include "sys$common:[000000.syslib]unixlib.h"
#include "sys$common:[000000.syslib]time.h"
#include "sys$common:[000000.nexpert.decwindows]nxpdef.h"
#include "dka300:[000000.pharasi.sybase]sybfront.h"
#include "dka300:[000000.pharasi.sybase]sybdb.h"
#include "dka300:[000000.pharasi.sybase]syberror.h"


#define RetCode char
#define FALSE 0
#define TRUE  1
#define MAXCHAR 31
#define MAXFLOAT 8
#define MAXINT 10
#define PATID_LEN 9

FILE *fopen(), *fp;
extern char *getenv();

static DBPROCESS *dbproc;                          // ptr to
SYBASE-process
static LOGINREC *login;                            // ptr to
SYBASE-login rec
static char *id, *name, *sex;                  // patient details
static char *userid, *username, *refname;   // consultant
details
static int   age;                              // patient details
static int mmi;                                    // modified
multfact.index
static char *real_surgery, *test_surgery;  // surgery type
```

```
static char *class;                          // surgery class
static double ser_pretest, sev_pretest,      // pre-/post-test
prob.
              ser_posttest, sev_posttest;
static struct medstruct
       {
         char medname[MAXCHAR];             // recommendations
by key
         char medcode[MAXCHAR];
         char medtext[255];
       } **medhead;
static struct riskstruct
       {
         char riskname[50];                    // cardiac risk
factors
       } **riskhead;
static char riskprobsaved, riskfactsaved, medrecsaved;

/*************************************************************
*****/
/* Procedure to change a string to lower case.
  */
/* Used for sybase objects.
  */
/*************************************************************
*****/
char *LowerCase(instring)
char *instring;
{
    char *c;
     if (instring == NULL) return NULL;
     for (c=instring; *c != '\0'; c++)
     {
       *c = tolower(*c);
     }
     return instring;
}
/*************************************************************
*****/
/* Procedure to check if a string value is an integer value
  */
/*************************************************************
*****/
  RetCode CheckInteger(instring)
  char *instring;
  {
      char *c;

      if (instring[0] == '\0') return FALSE;
```

```
      for (c=instring; *c != '\0'; c++)
      {
         if (!isdigit(*c)) return FALSE;
      }
      return TRUE;
  }

/*****************************************************************
******/
/* Procedure to check if a string value is a real value
   */
/*****************************************************************
******/
  RetCode CheckReal(instring)
  char *instring;
  {
      char *c;

      if (instring[0] == '\0') return FALSE;
      for (c=instring; *c != '\0'; c++)
      {
         if (!isdigit(*c) && *c != '.') return FALSE;
      }
      return TRUE;
  }

/*****************************************************************
******/
/* This procedure converts a date string with format yyyymmdd
to */
/* current age.It is used for getting patient age from patient
 */
/* birthdate
   */
/*****************************************************************
******/
  RetCode GetAge(datestring,howold)
  char *datestring;
  int  *howold;
  {
     char *c;
     struct tm *current_time;
     time_t     total_secs;
     char yyyy[5], mm[3];
     int current_year, current_mon, year, mon;

     if (datestring == NULL) return FALSE;
     if (!CheckInteger(datestring)) return FALSE;
```

```
    if (strlen(datestring) < 8) return FALSE;

    time(&total_secs);
    current_time = localtime(&total_secs);
    current_year = current_time->tm_year + 1900;
    current_mon  = current_time->tm_mon + 1;

    strncpy(yyyy, datestring, 4); yyyy[4] = '\0';
    strncpy(mm, (datestring + 4), 2);   mm[2] = '\0';
    year = atoi(yyyy);
    mon  = atoi(mm);

    *howold = current_year - year;
    if (current_mon < mon) *howold-= 1;

    return TRUE;
  }

/*****************************************************************
******/
/* Returns the current date as a string with format yyyymmdd
   */
/*****************************************************************
******/
  char *CurrentDate()
  {
    struct tm *current_time;
    time_t     total_secs;
    char       date_string[9];

    time(&total_secs);
    current_time = localtime(&total_secs);

    if (!sprintf(date_string,"%4d%02d%02d",
                             (1900 + current_time->tm_year),
                             (   1 + current_time->tm_mon),
                              current_time->tm_mday) )
       return NULL;
    date_string[8] = '\0';

    return date_string;
  }

/*****************************************************************
******/
/* Returns the current time as a string with format hhmmss
   */
/*****************************************************************
******/
```

```
  char *CurrentTime()
  {
     struct tm *current_time;
     time_t     total_secs;
     char       time_string[7];

     time(&total_secs);
     current_time = localtime(&total_secs);

                                                          i          f
(!sprintf(time_string,"%02d%02d%02d",current_time->tm_hour,

current_time->tm_min,

current_time->tm_sec) )
          return NULL;
     time_string[6] = '\0';

     return time_string;
  }



/*************************************************************
******/
/* General procedure for all screen inputs
   */
/*************************************************************
******/
  RetCode GetResponse(instring,n)
  char *instring;
  int n;
  {
      char c;
      int  cnt;

      fflush(stdin);
      cnt=0;
      while (TRUE)
      {
        c = getchar();
        if (c=='\r' || c=='\n') break;
     if (cnt > n - 2) continue;
        instring[cnt] = c;
     cnt++;
      }
      instring[cnt] = '\0';
      if (instring[0] == '\0') return FALSE;
      return TRUE;
```

```
  }

RetCode   hello()
{
     printf("\n ");
        printf("                Welcome to the PREOP Expert
System\n");
        printf("
---------------------------------\n\n");
        fflush(stdout);
     return TRUE;
}

/*************************************************************
******/
/* General error-handling procedure for the SQL-server
                                                    *        /
/*************************************************************
******/
 int   SybErrHandler(dbprocess,   severity,   dberr,   oserr,
dberrstr,
                     oserrstr)
  DBPROCESS *dbprocess;
  int        severity;
  int dberr;
  int oserr;
  char *dberrstr;
  char *oserrstr;
  {
     if  (dbprocess  ==  NULL  ||  DBDEAD(dbprocess))  return
INT_EXIT;
     printf("DB-library error is %s\n",dberrstr);
     if (oserr != DBNOERR)
        printf("Operating system error is %s\n",oserrstr);
     return INT_CANCEL;
  }

/*************************************************************
******/
/* General message handling procedure for sybase
   */
/*************************************************************
******/
  int  SybMsgHandler(dbprocess,  msgno,  msgstate,  severity,
msgtext,
                     servname, procname, line)
  DBPROCESS  *dbprocess;
  DBINT       msgno;
  int         msgstate;
```

```
  int          severity;
  char        *msgtext;
  char        *servname;
  char        *procname;
  DBUSMALLINT line;
  {
     if (severity == 0) return 0;
     printf("Msg %ld, Level %d, State %d\n",msgno, severity,
                                                    msgstate);
     if  (strlen(servname)  >  0)  printf("Server  '%s',",
servname);
     if  (strlen(procname)  >  0)  printf("Procedure  '%s',",
procname);
     if (line > 0) printf("Line %d", line);
     printf("\n\t%s\n", msgtext);
     return 0;
  }


/*****************************************************************
******/
/* Initialize sybase, set up the login record, open the sybase
 */
/* server using the login record, specify database to be used
  */
/*****************************************************************
******/
   RetCode InitSybase()
   {
       userid = (char *) malloc(strlen(getenv("USER")));
       if (!strcpy(userid, getenv("USER")) ) return FALSE;
       if (!LowerCase(userid)) return FALSE;

       if (dbinit() == FAIL) return FALSE;

       dbmsghandle(SybErrHandler);
       dbmsghandle(SybMsgHandler);

       if ((login=dblogin()) == NULL)
       {
         puts("dblogin failed");
         return FALSE;
       }

       DBSETLUSER(login,"preopuser");

       if ((dbproc=dbopen(login,NULL)) == NULL)
       {
         puts("dbopenfailed");
```

```
        return FALSE;
      }

      dbuse(dbproc,"preopdb");
      return TRUE;
   }

/*************************************************************
******/
/*    Check user's logon-id against user-table for authority
to  */
/*    use the PREOP expert system
   */
/*************************************************************
******/
   RetCode CheckUser()
   {
      if ((username = (char *) malloc(MAXCHAR)) == NULL)
         return FALSE;

      dbcmd(dbproc," select cons_name from consultant");
      dbfcmd(dbproc," where cons_id = '%s' ", userid);
      if (dbsqlexec(dbproc) == FAIL) return FALSE;
      dbresults(dbproc);
      dbbind(dbproc, 1, STRINGBIND, 0, username);
      if (dbnextrow(dbproc) == NO_MORE_ROWS)
      {
         printf("user  %s  is  not  authorized  to  use  this
system\n",
                  userid);
         return FALSE;
      }
      return TRUE;
   }

/*************************************************************
******/
/*    Promt user to input full name of the referring physician
 */
/*************************************************************
******/
   RetCode GetRef()
   {
      if ((refname = (char *) malloc(MAXCHAR)) == NULL)
         return FALSE;
      while(TRUE)
      {
        printf("\n\nEnter name of referring physician,
                  '?' for notknown: ");
```

```
        if (GetResponse(refname,MAXCHAR) == TRUE) break;
      }
      return TRUE;
   }

/*************************************************************
******/
/*    load the knowledge bases - preop,card,medrec,medexp in
order*/
/*************************************************************
******/
   RetCode LoadBase()
   {
        KBId preopkb, cardkb, medexpkb, medreckb;

        if(!NXP_LoadKB("mppreop4.tkb", &preopkb) )
        {
            printf("load preop-kb failed with error %d",
                        NXP_Error());
            return FALSE;
        }
        if(!NXP_LoadKB("mpcard4.tkb", &cardkb) )
        {
            printf("load card-kb failed with error %d",
                        NXP_Error());
            return FALSE;
        }
        if(!NXP_LoadKB("mpexp2.tkb", &medexpkb) )
        {
            printf("load medexp-kb failed with error %d",
                        NXP_Error());
            return FALSE;
        }
        if(!NXP_LoadKB("mprec2.tkb", &medreckb) )
        {
            printf("load medrec-kb failed with error %d",
                        NXP_Error());
            return FALSE;
        }
        return TRUE;
   }

/*************************************************************
*****/
/* Promt user for the patient-id and get the corresponding
  */
/* patient details from the patient-table.
  */
/* If patient-id is not known, prompt the user for the other
```

```
  */
/* patient details.
  */
/*****************************************************************
*****/
   RetCode GetPatientFile()
   {
        char input_id[PATID_LEN];
        char birthdate[9];

          id = (char *) malloc(PATID_LEN);
        name = (char *) malloc(MAXCHAR);
        sex  = (char *) malloc(2);

        while(TRUE)
        {

           printf("enter patient id (9(8) or '?' for notknown):
");
           if (!GetResponse(input_id,PATID_LEN)) continue;

           if (input_id[0] == '?') return FALSE;

           if (!CheckInteger(input_id)) continue;

           dbcmd(dbproc, "select * from patient");
           dbfcmd(dbproc, " where pat_id = '%s'",input_id);
           if (dbsqlexec(dbproc) == FAIL) return FALSE;
           dbresults(dbproc);
           dbbind(dbproc, 1, STRINGBIND, 0, id);
           dbbind(dbproc, 2, STRINGBIND, 0, name);
           dbbind(dbproc, 3, STRINGBIND, 0, sex);
           dbbind(dbproc, 4, STRINGBIND, 0, birthdate);

           if (dbnextrow(dbproc) == REG_ROW) break;
            printf("patient-id    %s    not    found    on
file\n",input_id);
        }
        birthdate[8] = '\0';

        if ( !(GetAge(birthdate,&age)) )
        {
           puts("error in reading patient birthdate");
           return FALSE;
        }
        return TRUE;
   }
```

```
/*************************************************************
******/
/*          if  patient-id  is  unknown  or  not  on  the
patient-table,prompt*/
/*    user for patient details
   */
/*************************************************************
******/
   RetCode GetPatientScreen()
   {
      char c, input_age[4];
      int  cnt;

      strcpy(id, "NOTKNOWN");

      while (TRUE)
      {
         printf("patient name: ");fflush(stdout);
         if (GetResponse(name,MAXCHAR)) break;
      }

      while (TRUE)
      {
         printf("patient sex (f/m): ");fflush(stdout);
         if (!GetResponse(sex,2)) continue;
         if (sex[0] == 'f' || sex[0] == 'm') break;
      }

      while (TRUE)
      {
         printf("patient age in years: ");fflush(stdout);
         if (!GetResponse(input_age,4)) continue;
         if (CheckInteger(input_age)) break;
      }
      age = atoi(input_age);

      return TRUE;
   }


/*************************************************************
******/
/*  volunteer patient details into the knowledge base
                                                    *          /
/*************************************************************
******/
   RetCode IdentifyPatient()
   {
```

```
      AtomId  idslot, nameslot, sexslot, ageslot;

       if    (!NXP_GetAtomId("patient.id",    &idslot,
NXP_ATYPE_SLOT))
      {
         puts("patient.id slot does not exist");
         return FALSE;
      }

      if (!NXP_GetAtomId("patient.name", &nameslot,
                              NXP_ATYPE_SLOT))
      {
         puts("patient.name slot does not exist");
         return FALSE;
      }


       if    (!NXP_GetAtomId("patient.sex",    &sexslot,
NXP_ATYPE_SLOT))
      {
         puts("patient.sex slot does not exist");
         return FALSE;
      }

       if    (!NXP_GetAtomId("patient.age",    &ageslot,
NXP_ATYPE_SLOT))
      {
         puts("patient.age slot does not exist");
         return FALSE;
      }

      if (!NXP_Volunteer(idslot, NXP_DESC_STR, id,
                         NXP_VSTRAT_SETQUEUE))
         return FALSE;

      if (!NXP_Volunteer(nameslot, NXP_DESC_STR, name,
                         NXP_VSTRAT_SETQUEUE))
         return FALSE;

      if (!NXP_Volunteer(sexslot, NXP_DESC_STR, sex,
                         NXP_VSTRAT_SETQUEUE))
         return FALSE;

      if (!NXP_Volunteer(ageslot, NXP_DESC_INT, &age,
                         NXP_VSTRAT_SETQUEUE))
         return FALSE;

      return TRUE;
   }
```

```
/*************************************************************
******/
/*  check the surgery-type from the knowledge base against the
 */
/*  surgery-table and volunteer its corresponding category and
 */
/*  risk probabilities into the knowledge base
   */
/*************************************************************
******/
   RetCode GetSurgeryType(atomstr, atomcnt, atomlist)
   char   *atomstr;
   int     atomcnt;
   AtomId *atomlist;
   {
        AtomId    *typeslot,    *categoryslot,    *serppslot,
*sevppslot;
        char *nxptype;
        char tbltype[MAXCHAR];
        char tblcategory[6];
        double tblsevpp, tblserpp;

        if (atomcnt<4 || !atomlist)
        {
           puts("invalid NXP parameters - GetSurgeryType");
           return FALSE;
        }
     typeslot     = atomlist;
     categoryslot = atomlist + 1;
     serppslot    = atomlist + 2;
     sevppslot    = atomlist + 3;

        nxptype = (char *) malloc(MAXCHAR);
        if (!NXP_GetAtomInfo(*typeslot, NXP_AINFO_VALUE, 0, 0,
                             NXP_DESC_STR, nxptype, MAXCHAR)
)
        {
           puts("NXP_GetAtomInfo failed - GetSurgeryType");
           return FALSE;
        }


        dbcmd(dbproc, "select * from surgery");
        dbfcmd(dbproc, " where type='%s'", nxptype);
        if (dbsqlexec(dbproc) == FAIL)
        {
           puts("Failure in reading sybase table surgery -
                 GetSurgeryType");
```

```
            return FALSE;
         }
         dbresults(dbproc);

         dbbind(dbproc, 1, STRINGBIND, 0, tbltype);
         dbbind(dbproc, 2, STRINGBIND, 0, tblcategory);
         dbbind(dbproc, 3, FLT8BIND,   0, &tblsevpp);
         dbbind(dbproc, 4, FLT8BIND,   0, &tblserpp);

         if (dbnextrow(dbproc) == NO_MORE_ROWS)
         {
            printf("surgery type %s undefined to this
system\n",
                    nxptype);
            return FALSE;
         }

         if (!NXP_Volunteer(*categoryslot, NXP_DESC_STR,
                                              tblcategory,
NXP_VSTRAT_SETQUEUE))
               return FALSE;

         if    (!NXP_Volunteer(*serppslot,    NXP_DESC_FLOAT,
&tblsevpp,
                               NXP_VSTRAT_SETQUEUE))
               return FALSE;

         if    (!NXP_Volunteer(*sevppslot,    NXP_DESC_FLOAT,
&tblserpp,
                               NXP_VSTRAT_SETQUEUE))
               return FALSE;

         free(nxptype);

         return TRUE;
   }


/*****************************************************************
******/
/*  use the mmi value from the knowledge base to determine the
 */
/*  corresponding likelihood class from the database
   */
/*****************************************************************
******/
   RetCode GetClass(atomstr, atomcnt, atomlist)
   char   *atomstr;
   int     atomcnt;
```

```
AtomId *atomlist;
{
     AtomId *mmislot, *classslot;
     int nxpmmi;
     char tblclass[10];

     if (atomcnt<2 || !atomlist)
     {
        puts("invalid NXP parameters - GetClass");
        return FALSE;
     }
   mmislot   = atomlist;
   classslot = atomlist + 1;

     if (!NXP_GetAtomInfo(*mmislot, NXP_AINFO_VALUE, 0, 0,
                          NXP_DESC_INT, nxpmmi, 3) )
        return FALSE;

     dbfcmd(dbproc, "select class from classrange");
     dbfcmd(dbproc, " where lowweight <= %d", nxpmmi);
     dbfcmd(dbproc, " and  highweight >= %d", nxpmmi);
     if (dbsqlexec(dbproc) == FAIL)
     {
        puts("Failure in reading sybase table riskclass
            - GetRatioProb");
        return FALSE;
     }
     dbresults(dbproc);

     dbbind(dbproc, 1, STRINGBIND, 0, tblclass);

     if (dbnextrow(dbproc) == NO_MORE_ROWS)
     {
        printf("no matching class found for mmi = %d\n",
nxpmmi);
        return FALSE;
     }

     if (!NXP_Volunteer(*classslot, NXP_DESC_STR, tblclass,
                        NXP_VSTRAT_SETQUEUE))
        return FALSE;

     return TRUE;
}

/*****************************************************************
******/
/*   use the surgery-class and surgery-category from the
knowledge*/
```

```
/*  base to get the corresponding likelihood ratio from the
table*/
/**********************************************************
******/
   RetCode GetProbRatio(atomstr, atomcnt, atomlist)
   char   *atomstr;
   int     atomcnt;
   AtomId *atomlist;
   {
        AtomId *classslot, *categoryslot, *ratioslot;
        char *nxpclass, *nxpcategory;
        double tblratio;

        if (atomcnt<3 || !atomlist)
        {
           puts("null NXP parameters - GetProbRatio");
           return FALSE;
        }
     classslot    = atomlist;
     categoryslot = atomlist + 1;
     ratioslot    = atomlist + 2;

        nxpclass = (char *) malloc(10);
        nxpcategory = (char *) malloc(6);

        if (!NXP_GetAtomInfo(*classslot, NXP_AINFO_VALUE, 0,
0,
                             NXP_DESC_STR, nxpclass, 10) )
           return FALSE;

        if (!NXP_GetAtomInfo(*categoryslot, NXP_AINFO_VALUE,
0, 0,
                             NXP_DESC_STR, nxpcategory, 6) )
           return FALSE;

        dbfcmd(dbproc, "select ratio from classratio");
        dbfcmd(dbproc, " where class = '%s'", nxpclass);
        dbfcmd(dbproc, " and category = '%s'", nxpcategory);
        if (dbsqlexec(dbproc) == FAIL)
        {
           puts("Failure in reading sybase table riskclass
               - GetRatioProb");
           return FALSE;
        }
        dbresults(dbproc);

        dbbind(dbproc, 1, FLT8BIND, 0, &tblratio);

        if (dbnextrow(dbproc) == NO_MORE_ROWS)
```

```
        {
           printf("class %s and category %s pair undefined to
this
                   system\n", nxpclass, nxpcategory);
           return FALSE;
        }

        if    (!NXP_Volunteer(*ratioslot,    NXP_DESC_FLOAT,
&tblratio,
                              NXP_VSTRAT_SETQUEUE))
            return FALSE;

        free(nxpclass);
        free(nxpcategory);

        return TRUE;
   }


/*************************************************************
******/
/*   suggest the 'start' hypothesis into the knowledge base
   */
/*************************************************************
******/
   RetCode Start()
   {
        AtomId starthypo;

         if    (!NXP_GetAtomId("start",    &starthypo,
NXP_ATYPE_HYPO) )
        {
           puts("hypothesis start does not exist");
           return FALSE;
        }
        if (!NXP_Suggest(starthypo,NXP_SPRIO_SUG) )
        {
          puts("suggest start failed");
          return FALSE;
        }
        return TRUE;
   }


/*************************************************************
******/
/* Get recommendation identifiers (medication & condition
name)  */
```

```
/* from NEXPERT and store corresponding texts in one string
   */
/*****************************************************
******/
    RetCode SaveRecommends(atomstr, atomcnt, atomlist)
    char   *atomstr;
    int     atomcnt;
    AtomId *atomlist;
    {
        AtomId *atomptr;
        struct medstruct **headptr, *medptr;
        int maxmed, cnt, reclen, totlen;

        if (!atomcnt || !atomlist)
        {
           puts("null NXP parameters - GetRecommends");
           return FALSE;
        }

        maxmed = atomcnt/2;
        totlen = 0;

        if ( (medhead = (struct medstruct **)
                        calloc(maxmed + 1,sizeof(*medhead)) )
==
   NULL)
        {
           puts("memory allocation failed - GetRecommends");
           return FALSE;
        }

        for (cnt=0,atomptr=atomlist; cnt<maxmed && atomptr;
cnt++,
                                                atomptr++)
        {
            if ((*(medhead + cnt) = (struct medstruct *)
                   malloc(sizeof(struct medstruct)) ) == NULL)
            {
                puts("memory    allocation    failed    -
GetRecommends");
               return FALSE;
            }
            medptr = *(medhead + cnt);

            if (!NXP_GetAtomInfo(*atomptr, NXP_AINFO_VALUE, 0,
0,
                                 NXP_DESC_STR,  medptr->medname,
30))
            return FALSE;
```

```
            if (!NXP_GetAtomInfo(*(atomptr + maxmed),
                        NXP_AINFO_VALUE, 0, 0, NXP_DESC_STR,
                            medptr->medcode, 30))
                return FALSE;
        }
        *(medhead + cnt) = NULL;

        for   (cnt=0,headptr=medhead;   cnt<maxmed,*headptr;
cnt++,

headptr++)
        {
            dbfcmd(dbproc,    "select    rec_text    from
recommendation");
            dbfcmd(dbproc, " where rec_med =  '%s'",
                        (**headptr).medname);
            dbfcmd(dbproc, " and   rec_cond = '%s'",
                        (**headptr).medcode);
            if (dbsqlexec(dbproc) == FAIL) continue;
            dbresults(dbproc);
              dbbind(dbproc,    1,    STRINGBIND,    0,
(**headptr).medtext);
            if (dbnextrow(dbproc) == NO_MORE_ROWS) continue;
            totlen+= dbdatlen(dbproc, 1);

            if ( (**headptr).medtext == NULL )
               strcpy( (**headptr).medtext, (**headptr).medcode
);
        }

        medrecsaved = TRUE;
        return TRUE;
    }


/*****************************************************************
******/
/* Keep the cardiac-rik-determination results in pgm variables
 */
/* so they can be stored in the database at the end of
   */
/* knowledge processing
   */
/*****************************************************************
******/
   RetCode SaveRiskProb(atomstr, atomcnt, atomlist)
   char   *atomstr;
   int     atomcnt;
```

```
   AtomId *atomlist;
   {
      int cnt;
      int atomtype;
      AtomId *atomptr;

      if (!atomcnt || !atomlist)
      {
         puts("null NXP parameter(s) - SaveRiskProb");
         return FALSE;
      }

      if ( !(real_surgery = (char *) malloc(MAXCHAR)) )
         return FALSE;
      if ( !(test_surgery = (char *) malloc(MAXCHAR)) )
         return FALSE;
      if ( !(class    = (char *) malloc(MAXCHAR)) ) return
FALSE;

      for (cnt=0,atomptr=atomlist; cnt<atomcnt && atomptr;
cnt++,
                                                   atomptr++)
      {
         switch(cnt)
         {
                                  case 0 : { if
(!NXP_GetAtomInfo(*atomptr,NXP_AINFO_VALUE,
                                    0,   0,   NXP_DESC_STR,
real_surgery,

MAXCHAR))
                        return FALSE;
                     break;
                    }
                                  case 1 : { if
(!NXP_GetAtomInfo(*atomptr,NXP_AINFO_VALUE,
                                    0,   0,   NXP_DESC_STR,
test_surgery,

MAXCHAR))
                        return FALSE;
                     break;
                    }
                                  case 2 : { if
(!NXP_GetAtomInfo(*atomptr,NXP_AINFO_VALUE,
                                    0,   0,   NXP_DESC_FLOAT,
&sev_pretest,
                                    MAXFLOAT))
                        return FALSE;
```

```
                    break;
                   }
                                c a s e   3 :   { i f
(!NXP_GetAtomInfo(*atomptr,NXP_AINFO_VALUE,
                        0, 0, NXP_DESC_FLOAT, &ser_pretest,
                                                  MAXFLOAT))
                       return FALSE;
                    break;
                   }
                                c a s e   4 :   { i f
(!NXP_GetAtomInfo(*atomptr,NXP_AINFO_VALUE,
                        0, 0, NXP_DESC_FLOAT, &sev_posttest,
                        MAXFLOAT))
                       return FALSE;
                    break;
                   }
                                c a s e   5 :   { i f
(!NXP_GetAtomInfo(*atomptr,NXP_AINFO_VALUE,
                        0, 0, NXP_DESC_FLOAT, &ser_posttest,
                        MAXFLOAT))
                       return FALSE;
                    break;
                   }
                                c a s e   6 :   { i f
(!NXP_GetAtomInfo(*atomptr,NXP_AINFO_VALUE,
                        0, 0, NXP_DESC_STR, class, MAXCHAR))
                       return FALSE;
                    break;
                   }
                                c a s e   7 :   { i f
(!NXP_GetAtomInfo(*atomptr,NXP_AINFO_VALUE,
                                 0,  0,  NXP_DESC_INT,  &mmi,
MAXINT))
                       return FALSE;
                    break;
                   }
            default : break;
           }
        }

        riskprobsaved = TRUE;
        return TRUE;
   }


/*************************************************************
******/  /*  Get  recommendation  identifiers  (medication  &
condition name)  */
/* from NEXPERT and store corresponding texts in one string
```

```
   */
/*************************************************************
******/
    RetCode SaveRiskFact(atomstr, atomcnt, atomlist)
    char   *atomstr;
    int     atomcnt;
    AtomId *atomlist;
    {
        AtomId *atomptr;
        AtomId  slotid;
        struct riskstruct *riskptr;
        int maxrisk, cnt;

        if (!atomcnt || !atomlist)
        {
           puts("null NXP parameters - SaveRiskFactor");
           return FALSE;
        }

        maxrisk = atomcnt;

        if ( (riskhead = (struct riskstruct **)
                        calloc(maxrisk + 1,sizeof(*riskhead))
) ==
    NULL)
        {
           puts("memory allocation failed - SaveRiskFactor");
           return FALSE;
        }

        for (cnt=0,atomptr=atomlist; cnt<maxrisk && atomptr;
cnt++,

atomptr++)
        {
            if ((*(riskhead + cnt) = (struct riskstruct *)
                   malloc(sizeof(struct  riskstruct))  )  ==
NULL)
            {
                puts("memory    allocation    failed    -
SaveRiskFactor");
               return FALSE;
            }
            riskptr = *(riskhead + cnt);

            if (!NXP_GetAtomInfo(*atomptr, NXP_AINFO_NAME, 0,
0,
                                 NXP_DESC_STR, riskptr->riskname,
50))
```

```
                return FALSE;

        }
        *(riskhead + cnt) = NULL;

        riskfactsaved = TRUE;
        return TRUE;
    }

/*************************************************************
******/
/* Write the assessment results to file/screen
   */
/*************************************************************
******/
   RetCode WriteResults()
   {
      int cmdlen;
      char *cmdstr;
      char curr_date[9], curr_time[7];
      struct medstruct **medptr;
      struct riskstruct **riskptr;

      if (!(strcpy(curr_date,CurrentDate()) )) return FALSE;
      if (!(strcpy(curr_time,CurrentTime()) )) return FALSE;

      if (!riskprobsaved) return FALSE;

      cmdlen = 100 + strlen(id) + strlen(curr_date)
                        + strlen(curr_time)
                  + strlen(test_surgery) + strlen(class)
                  + strlen(userid) + strlen(refname);

      if ((cmdstr = (char *) malloc(cmdlen)) == NULL)
      {
           puts("memory   allocation   failed   -
WriteResults,cmdstr");
         return FALSE;
      }

      if (!sprintf(cmdstr, " values

('%s','%s','%s','%s','%s','%s','%s',%d,%f,%f,%f,%f)",
           id,   curr_date,   curr_time,   userid,   refname,
test_surgery,
                class,   mmi,   sev_pretest,   ser_pretest,
sev_posttest,
                ser_posttest) )
      {
```

```
            puts("sprintf failed - WriteResults");
            return FALSE;
        }

        dbcmd(dbproc, " insert into assess_riskprob");
        dbcmd(dbproc,   "   (patid,   date,   time,   consultant,
refid,");
        dbcmd(dbproc, " surgery_type, risk_class, mmi,");
        dbcmd(dbproc, " severe_pretest, serious_pretest,");
        dbcmd(dbproc, " severe_posttest, serious_posttest) ");
        dbcmd(dbproc, cmdstr);

        if (dbsqlexec(dbproc) == FAIL)
        {
            printf("failure in writting to assess_cardrisk\n");

            return FALSE;
        }

        dbresults(dbproc);

        fp = fopen("preop.txt", "a");

        fprintf(fp,"\n ");
        fprintf(fp,"                    PREOP Assessment Results
\n");
        fprintf(fp,"                    ------------------------
\n\n");
        fprintf(fp," CONSULTANT           : %s\n", username);
        fprintf(fp," REFERRING PHYSICIAN : %s\n", refname);
        fprintf(fp," PATIENT             : %s  %s\n", id, name);
        fprintf(fp," PATIENT SEX          : %s\n", sex);
        fprintf(fp," PATIENT AGE          : %d\n", age);
        fprintf(fp," SURGERY             : %s\n", real_surgery);

        if ( strcmp(real_surgery,test_surgery) != 0)
        {
        fprintf(fp," ASSESSMENT SURGERY  : %s\n", test_surgery);
        }

        fprintf(fp," SURGERY CLASS        : %s\n", class);
        fprintf(fp," MMI                  : %d\n\n", mmi);
        fprintf(fp,"                       SEVERE       SERIOUS
\n");
        fprintf(fp,"                       ------       -------
\n");
        fprintf(fp," PRE_TEST PROB.       : %f   %f\n",

sev_pretest,ser_pretest);
```

```
      fprintf(fp," POST_TEST PROB.      : %f   %f\n",
sev_posttest,ser_posttest);
      fprintf(fp," \n\n");

      printf("\n ");
      printf("                  PREOP Assessment Results \n");
      printf("                        ----------------------
\n\n");
      printf(" CONSULTANT           : %s\n", username);
      printf(" REFERRING PHYSICIAN  : %s\n", refname);
      printf(" PATIENT              : %s  %s\n", id, name);
      printf(" PATIENT SEX          : %s\n", sex);
      printf(" PATIENT AGE          : %d\n", age);
      printf(" SURGERY              : %s\n", real_surgery);

      if ( strcmp(real_surgery,test_surgery) != 0)
      {
      printf(" ASSESSMENT SURGERY   : %s\n", test_surgery);
      }

      printf(" SURGERY CLASS        : %s\n", class);
      printf(" MMI                  : %d\n\n", mmi);
      printf("                          SEVERE        SERIOUS
\n");
      printf("                          ------        -------
\n");
      printf(" PRE_TEST PROB.       : %f     %f\n",
                                     sev_pretest,ser_pretest);

      printf(" POST_TEST PROB.      : %f     %f\n",
sev_posttest,ser_posttest);
      printf(" \n\n");

      if (riskfactsaved && riskhead)
      {
         fprintf(fp," IDENTIFIED RISK FACTORS\n");
         fprintf(fp," ----------------------\n\n");

         printf(" IDENTIFIED RISK FACTORS\n");
         printf(" ----------------------\n\n");

         for (riskptr=riskhead; *riskptr; riskptr++)
         {
           {
               fprintf(fp,"   %s\n", (**riskptr).riskname);

               printf("   %s\n", (**riskptr).riskname);
```

```
                dbfcmd(dbproc, "insert into assess_riskfact");
                dbfcmd(dbproc,    "    (patid,    date,    time,
risk_factor)");
                dbfcmd(dbproc, " values('%s','%s','%s','%s')",
                                id, curr_date, curr_time,
                                (**riskptr).riskname);
                if (dbsqlexec(dbproc) == FAIL)
                {
                  fprintf(fp,"Failure    in    writting    to
assess_riskfact\n");
                   printf("Failure    in    writting    to
assess_riskfact\n");
                  fclose(fp);
                  return FALSE;
                }
                dbresults(dbproc);
             }
          }
       }
       fprintf(fp," \n\n");
       printf(" \n\n");

       if (medrecsaved && medhead)
       {
          fprintf(fp,"    RECOMMENDATIONS    FOR    MEDICATION
INTAKE\n");
                          f p r i n t f ( f p , "
----------------------------------\n\n");

          printf(" RECOMMENDATIONS FOR MEDICATION INTAKE\n");
          printf(" ----------------------------------\n\n");
          for (medptr=medhead; *medptr; medptr++)
          {
             fprintf(fp," %s:\n", (**medptr).medname);
             fprintf(fp,"       %s\n\n", (**medptr).medtext);

             printf(" %s:\n", (**medptr).medname);
             printf("        %s\n\n", (**medptr).medtext);

             dbfcmd(dbproc, "insert into assess_medrec");
             dbfcmd(dbproc, " (patid, date, time, medication,
                                                recommend)");
                    d b f c m d ( d b p r o c ,    "    v a l u e s
('%s','%s','%s','%s','%s')",
                                id, curr_date, curr_time,
                                     (**medptr).medname,
(**medptr).medtext );
             if (dbsqlexec(dbproc) == FAIL)
             {
```

```
                fprintf(fp,"Failure    in    writting    to
assess_medrec");
                printf("Failure in writting to assess_medrec");
                fclose(fp);
                return FALSE;
             }
             dbresults(dbproc);
          }
       }

       return TRUE;
    }

/*****************************************************************
******/
/*  a general procedure to get questions from the knowledge
base */
/*  to the user and to feed back the response to the KB
    */
/*****************************************************************
******/
    RetCode QuestionHandler(slot, prompt)
    AtomId slot;
    char  *prompt;
    {
       char c, response[MAXCHAR], numresponse[MAXCHAR];
       struct table
       {
          char choicestr[MAXCHAR];
       } *tableptr, *ptr;
       int  cnt, maxchoices;
       int  atomtype;
       int  intvalue;
       double dblvalue;

       printf("\n\n %s ",prompt);
       fflush(stdin);

       if (!NXP_GetAtomInfo(slot, NXP_AINFO_VALUETYPE, 0, 0,
                                  NXP_DESC_INT, &atomtype, 0))
            return FALSE;

       maxchoices = 0;
       if (NXP_GetAtomInfo(slot, NXP_AINFO_CHOICE, 0, -1,
                NXP_DESC_INT, &maxchoices, 0) && maxchoices>0)
       {
          if ( !(tableptr = (struct table *)
                        malloc(maxchoices    *    sizeof(struct
table))) )
```

```
            return FALSE;
         printf("maxchoices is %d\n",maxchoices);
         for (cnt=0,ptr=tableptr; cnt<maxchoices; cnt++,ptr++)
         {
            printf("\n          %d.",(1 + cnt));
            if  (NXP_GetAtomInfo(slot,  NXP_AINFO_CHOICE,  0,
cnt,
                  NXP_DESC_STR, ptr->choicestr, MAXCHAR))
            {
                printf(" %s", ptr->choicestr);
            }
         }

         printf("\n");
         while (TRUE)
         {
            printf(" enter a number between 1 and %d: ",

maxchoices);
            if (!GetResponse(numresponse,MAXCHAR)) continue;
            if (!CheckInteger(numresponse)) continue;
            if (atoi(numresponse) < 1
            || atoi(numresponse) > maxchoices)
               continue;
            ptr = tableptr + atoi(numresponse) - 1;
            strcpy(response, ptr->choicestr);
            break;
         }
      }
      else
      {
         while(TRUE)
         {

            if (!GetResponse(response,MAXCHAR))
            {
               printf("\n\n %s ",prompt);
               continue;
            }

            if (response[0] == '?')
            {
              if (!NXP_Volunteer(slot, NXP_DESC_NOTKNOWN, 0,
                                 NXP_VSTRAT_SETQUEUE))
              {
                 puts("notknown response not volunteered");
                 return FALSE;
              }
              return TRUE;
```

```
            }

            if (atomtype == NXP_VTYPE_STR) break;

            if (atomtype == NXP_VTYPE_LONG)
            {
               if (CheckInteger(response)) break;
               printf(" enter integer value\n");
            }

            if (atomtype == NXP_VTYPE_DOUBLE)
            {
               if (CheckReal(response)) break;
               printf(" enter integer+decimals value\n");
            }

            if (atomtype == NXP_VTYPE_BOOL)
            {
               if (response[0] == 'y' || response[0] == 'n')
break;
               printf(" enter 'y' or 'n' \n");
            }

            printf("\n\n %s ",prompt);
         }
      }

      switch (atomtype)
      {
           case NXP_VTYPE_STR    : { if (!NXP_Volunteer(slot,
                                        NXP_DESC_STR,
response,

NXP_VSTRAT_SETQUEUE))
                                           return FALSE;
                                     break;
                                   }
           case  NXP_VTYPE_LONG          :  {  intvalue  =
atoi(response);
                                    if (!NXP_Volunteer(slot,
                                        NXP_DESC_INT,
&intvalue,

NXP_VSTRAT_SETQUEUE))
                                           return FALSE;
                                     break;
                                   }
           case   NXP_VTYPE_DOUBLE      :   {   dblvalue   =
atof(response);
```

```
                                        if (!NXP_Volunteer(slot,
                                              NXP_DESC_DOUBLE,
                                                   &dblvalue,
NXP_VSTRAT_SETQUEUE))
                                                 return FALSE;
                                          break;
                                        }
            case NXP_VTYPE_BOOL    : { if (response[0] == 'y')
                                              intvalue = TRUE;
                                         else intvalue = FALSE;
                                        if (!NXP_Volunteer(slot,
                                                   NXP_DESC_INT,
&intvalue,
NXP_VSTRAT_SETQUEUE))
                                                 return FALSE;
                                          break;
                                        }
            default                        :{printf("atomtype %d
invalid\n",
                                                   atomtype);
                                         return FALSE;
                                        }
       }
       return TRUE;
    }

/*****************************************************************
******/
/*   a general procedure to alert the user of any NEXPERT
system  */
/*  errors.
    */
/*****************************************************************
******/
    RetCode AlertHandler(code, thestr, ret)
    int   code;
    char *thestr;
    int  *ret;
    {
       printf("\n %s \n", thestr);
       switch(code)
       {
       case NXP_ALRT_OK: { ret = NXP_RET_OK; break;}

       case NXP_ALRT_OKCANCEL: {ret = NXP_RET_CANCEL; break;}

       case  NXP_ALRT_YESNOCANCEL:  {ret  =  NXP_RET_CANCEL;
```

```
break;}

      default: {ret = NXP_RET_CANCEL; break;}
    }

    return TRUE;
}

main()
{
    NXP_Control(NXP_CTRL_INIT);
    NXP_SetHandler(NXP_PROC_EXECUTE, GetSurgeryType,
                                        "GetSurgeryType");
    NXP_SetHandler(NXP_PROC_EXECUTE, GetClass,
                                        "GetClass");
    NXP_SetHandler(NXP_PROC_EXECUTE, GetProbRatio,
                                        "GetProbRatio");
    NXP_SetHandler(NXP_PROC_EXECUTE, SaveRecommends,
                                        "SaveRecommends");
    NXP_SetHandler(NXP_PROC_EXECUTE, SaveRiskProb,
                                        "SaveRiskProb");
    NXP_SetHandler(NXP_PROC_EXECUTE, SaveRiskFact,
                                        "SaveRiskFact");
    NXP_SetHandler(NXP_PROC_ALERT,  AlertHandler, (char *)0);
    NXP_SetHandler(NXP_PROC_QUESTION, QuestionHandler,
                                          (char *)0);

    if (!InitSybase()) exit(1);

    if (!CheckUser(userid)) { dbexit(); exit(1);}

    if (!LoadBase()) { dbexit(); exit(1);}

    hello();

    if (!GetPatientFile() && !GetPatientScreen() )
    { dbexit(); exit(1);}

    GetRef();

    if (!IdentifyPatient()) { dbexit(); exit(1);}

    if (!Start()) { dbexit(); exit(1);}

    riskprobsaved = riskfactsaved = medrecsaved = FALSE;


    NXP_Control(NXP_CTRL_KNOWCESS);
```

```
    NXP_Control(NXP_CTRL_EXIT);
    WriteResults();
    dbexit();
}
```

APPENDIX B

Welcome to the PREOP Expert System

enter patient id (9(8) or '?' for notknown): 12345678

Enter name of referring physician,'?' for notknown: T. Testconsultant

What type of surgery is Smith John,M about to undergo ? maxchoices is 12

1. aortic_vascular
2. carotid_vascular
3. cataract
4. head
5. intraperitoneal
6. intrathoracic
7. neck
8. orthopedic
9. other
10. peripheral_vascular
11. TURP
12. vascular

enter a number between 1 and 12: 4

How urgent is Smith John,M's operation ? maxchoices is 2

1. elective
2. emergency

enter a number between 1 and 2: 2

Does Smith John,M have a poor general medical status? (y/n) y

Has Smith John,M had myocardial infarction within the past 6 months? (y/n) y

Has Smith John,M had myocardial infarction more than 6 months ago? (y/n) y

Has Smith John,M experienced any signs of angina?(y/n) y

Is Smith John,M able to perform physical activity without the development of angina?(y/n) n

Does Smith John,M have unstable angina? (y/n) n

Does Smith John,M have a history of syncope on exertion?(y/n) n

Has Smith John,M had more than 5 premature ventricular contractions at any time prior to surgery? (y/n) n

Was there a rythm other than sinus or sinus+APB on Smith John,M's last preopera tive ECG? (y/n) n

Has Smith John,M ever had alveolar pulmonary edema? (y/n) y

Has Smith John,M had alveolar pulmonary edema within the past week? (y/n) y

Does Smith John,M experience angina while climbing less than one flight of stairs at a normal pace?(y/n) n

Does Smith John,M experience angina while walking a couple of blocks on level g round?(y/n) n
saveriskprob successful

Enter the types of medications Smith John,M is using (choose "finished" when done). maxchoices is 11

1. anti_epileptic
2. anti_inflammatory
3. anticoagulant
4. antidiabetic
5. antihypertensive
6. corticosteroid
7. estrogen
8. finished
9. psychotropic
10. pulmonary
11. thyroid

enter a number between 1 and 11: ~r{_1
enter a number between 1 and 11: 1

Enter the types of medications Smith John,M is using (choose "finished" when done). maxchoices is 11

1. anti_epileptic
2. anti_inflammatory

```
          3. anticoagulant
          4. antidiabetic
          5. antihypertensive
          6. corticosteroid
          7. estrogen
          8. finished
          9. psychotropic
          10. pulmonary
          11. thyroid
 enter a number between 1 and 11: 3

 Enter the types of medications Smith John,M is using (choose
"finished" when done). maxchoices is 11

          1. anti_epileptic
          2. anti_inflammatory
          3. anticoagulant
          4. antidiabetic
          5. antihypertensive
          6. corticosteroid
          7. estrogen
          8. finished
          9. psychotropic
          10. pulmonary
          11. thyroid
 enter a number between 1 and 11: 8

 What is the name of the anti-epileptic drug Smith John,M is
using? maxchoices is 8

          1. carbamazepine
          2. clonazepam
          3. mephobarbital
          4. mesantoin
          5. phenobarbital
          6. phenytoin
          7. primidone
          8. valproic acid
 enter a number between 1 and 8: 5

 What type of epileptic seizure does the patient experience?
maxchoices is 3

          1. focal
          2. grand mal
          3. petit
 enter a number between 1 and 3: 2

 How well have the epileptic seizures been controlled in the
```

```
past year? maxchoices is 2

         1. adequate
         2. poor
 enter a number between 1 and 2: 1

 What is the risk of thrombosis for Smith John,M? maxchoices
is 3

         1. high
         2. intermediate
         3. low
 enter a number between 1 and 3: 3


                    PREOP Assessment Results
                   ------------------------

 CONSULTANT           : Pharasi Mathabo,M
 REFERRING PHYSICIAN  : T. Testconsultant
 PATIENT              : 12345678  Smith John,M
 PATIENT SEX          : m
 PATIENT AGE          : 68
 SURGERY              : head
 SURGERY CLASS        : class_III
 MMI                  : 70

                        SEVERE        SERIOUS
                        ------        -------
 PRE_TEST PROB.       : 0.078000      0.026000
 POST_TEST PROB.      : 0.558120      0.284970


 IDENTIFIED RISK FACTORS
 -----------------------

   Class_IV_angina
   emergency_operation
   myocardial_infarction_more_than_6_months
   myocardial_infarction_within_6_months
   alveolar_pulmonary_edema_within_1_week
   alveolar_pulmonary_edema_ever
   poor_general_status


 RECOMMENDATIONS FOR MEDICATION INTAKE
 -------------------------------------

 anticoagulant:
```

For intermediate or low risk patients, management is similar to that of a high risk patient with anticoagulation reversed with intravenous vitamin K or fresh frozen plasma until prothrombin time is normal or within 2-3 seconds of control

anti_epileptic:

Although grand mal-seizures can increase the risk of surgery substantially, it is generally unnecessary to measure blood levels of anti-epileptic medications because control of seizures has been adequate for the year before surgery