NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

A User-Definable Office Information System

David A. Walther

March 1990, Revised December 1990

A thesis presented to the University of Ottawa in partial fulfilment of the requirements for the degree of **Master of Computer Science** in The Department of Computer Science.

ISBN 0-315-68012-1

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

CONTENTS

EXAMPLES

FIGURES

TABLES

Preface

Work on this thesis started in 1985 while the author was working for Digital Equipment of Canada in Ottawa. The author worked in Hong Kong for Digital Equipment of Hong Kong Ltd, from October 1986 to July 1989. He returned to Canada in July 1989 to continue work with Digital Equipment of Canada in Ottawa. The thesis was completed in March 1990.

Abstract

This thesis investigates the design and implementation of an Office Information System that allows user input in defining a system for the office environment. This system has a limited knowledge of its domain, as an expert system. This knowledge is provided by users, and is specified in terms of production rules. The system allows users to tailoring of the system to best meet their own requirements. The system has been implemented in Common Lisp on a VAX computer.

Acknowledgements

I would like to express my thanks to those people who have helped me start, continue and finally finish this thesis. This includes Dr. George White , my thesis advisor, my various managers at Digital Equipment of Canada, Mike Erdelyi, Helen Gordon, and Judy Knowles, and my managers at Digital Equipment of Hong Kong, Derek Tse and Kevin Ng. I would also like to acknowledge the support of my wife Megan, who endured my work on this thesis.

CHAPTER 1
INTRODUCTION

> *"Office information systems can be viewed as critically dependent on their ability to support individual users in their personal decision making and problem solving.* 'An office information system can be generally defined as a system which assists office workers in a variety of tasks' [ELL87]. *Such an emphasis makes it especially important that the system is tuned to the situation and needs as experienced by the user. From the systems design point of view this means that while corporate information systems rely on a careful design of a common database and standard transactions, an OIS will not be effective unless it also provides appropriate services for local users"* [Hägglund Aug88a].

Systems development and specifically Office Information System development has commonly followed the traditional software development process (to be discussed shortly). The result of this process is a system which is generally thrust upon its users. The resultant system does not fully meet the user's expectations and is usually not current with office practices. It can actually slow the various functions the users are to perform, and the users quickly resist and reject the system [Suchman Apr88]. The system analysts are often viewed by the users as not being in touch with the users needs and as being too technical [Suchman Apr88], [Good et al. Oct84]. The author having been both system developer and system user agrees. Having to use Information Systems which are inflexible to customisations, cumbersome to use, and

unable to provide users with required information, has provided the motivation for this thesis. There is a dualism which exists in literature and practice regarding the human-computer interface [Good et al. Oct84]. The two opposing views are:

1. Adapt the user to the system
2. Adapt the system to the user.

In this dualism, some believe that analysts and designer can create "ideal" systems for users with limited user input. When users had problems with the system, the solution was to provide additional training so that the user's understanding of the system improved. This solution unfortunately avoids the real problem, that the system does not meet the user's requirements. Others believe that all systems must develop from genuine user requirements, and have a large amount of user input. This process was used in the development of the Apple Lisa computer system [Good et al. Oct84], which produced a system that was "user friendly" and had much user derived functionality. The author believes in the second view.

Many Information Systems being developed today still do not meet the goal of improving the efficiency of work within the organisation by the introduction of new tools (the system). This seems difficult to comprehend with the recent development of the tools and products, especially in the Computer Aided Software Engineering (CASE) environment. Users generally have limited input into the systems development process. They are permitted input for specifications and in final acceptance testing, but are excluded from all other processes. Designers, analysts and programmers will often make decisions which simplify the coding of the system at the expense of "user friendliness" and ease of use. Systems are often too inflexible for local variations required by different individuals, different offices, different regions in a country or for different countries [Suchman Apr88], which is of great importance to the user. Systems must be designed to give all the users some degree of local tailoring, especially in the area of Office Information Systems.

Systems are frequently developed as tools for users (the domain experts) to perform tasks. Domain knowledge is coded within the system through the flow of program statements. This is contrasted with *expert systems* where domain knowledge can be embedded within the system through production rules. Systems serve their users by performing work related tasks. The level of work is limited by the ability of users to productively use the system. Should users be unable to effectively use the system, they are viewed as poor users. A user who can use the system to perform the many different required tasks is viewed as a good user even if he has little domain knowledge.

The author proposes an Office Information System that allows users to design parts of, and provide direct input for domain specific tasks. It allows users to tailor the system to specific user needs, and incorporates user knowledge as do expert systems. The system has the following general goals

- Tasks are defined by the user with support from a knowledge engineer and a database designer
- Easy individual user customisation
- Flexibility
  - — to be utilised in a number of different office environments
  - — to be changed quickly to match changes in office tasks
- Provide expert system like processing to the office environment

The system is a tool that could be used in the office environment. A short historical review of the Traditional Systems Development Process follows, to highlight the features and benefits of this new system and its environment.

1.1 Traditional System Development Process

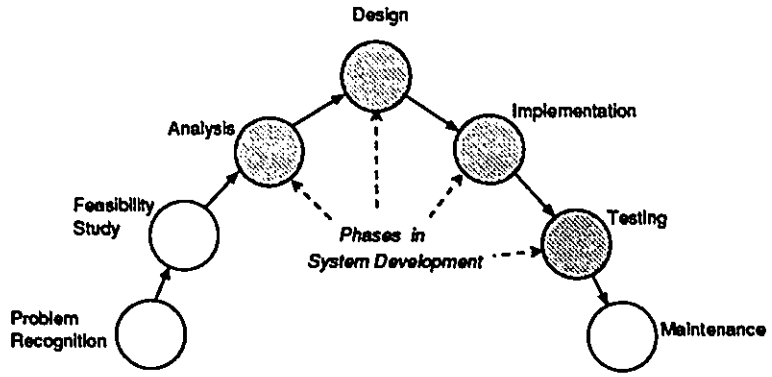
The traditional system life cycle [Page-Jones 80] consists of the following phases:

- Problem Recognition

- Feasibility Study
- Analysis
- Design
- Implementation
- Testing
- Maintenance

The system life cycle of is illustrated in figure 1–1.

Figure 1–1: Traditional System Life Cycle



For typical development of a system, the amount of work (time) spent on each phase is illustrated in table 1–1 [Page-Jones 80].

Table 1–1: Time spent per phase of Development

Development Phase	Life Cycle Phase	Percentage of Time
Requirements	Analysis	10%
Specification	Analysis	10%
Design	Design	15%
Coding	Implementation	20%

Table 1–1 (Cont.): Time spent per phase of Development

Development Phase	Life Cycle Phase	Percentage of Time
Module Testing	Testing	25%
Integration Testing	Testing	20%

The process of system development appeared in the early 1960's as large computer application development started. Some common problems appeared in many of the systems that were developed which were initially the result of the people performing the work. Early computer development experienced a lack of analysts and designers. The people involved in system development were computer programmers who were often ill prepared to design large systems. Frequently there were no existing methodologies or systems for the developers to follow or work with. As a result, common problems occurred in the resultant systems [Page-Jones 80], including:

1.1.1 Unmanageable Systems

Many systems were evaluated based upon the amount of code that is used to produce them. The more code, the better the system was supposed to be. This philosophy produced systems that were composed of enormous volume of code which could not be completely understood.

1.1.2 Unreliable Systems

Many systems did not always function correctly. Software bugs were constantly being located and patches applied. As noted in table 1–1, about 45 percent of the development time is devoted to testing and debugging.

1.1.3 Inflexible Systems

Most projects are under great pressure to deliver the "working system" within the specified time frame and within the specified budget. The priority of designing a "flexible" system, one in which future changes can be made for a variety of reasons, seldom occurs. Most system developers design and code system to work and not to have any flexibility for any future changes that could occur.

After a system is completed, it then enters the maintenance phase. History has shown that this is the most costly part of system development [Page-Jones 80].

Table 1–2: Lifetime System Phase Cost

Phase	Percentage of Total System Cost
Requirements	3%
Specifications	3%
Design	5%
Coding	7%
Module Testing	8%
Integration Testing	7%
Maintenance	67%

Boehm has stated that maintenance costs are as low as they are because some systems are so unmaintainable early in their lifetime that they have to be scrapped [Boehm Dec76]. Maintenance includes fixing existing bugs in the software, and implementing enhancements or modifications as they are required.

Clearly an excess of resources was spent in the process of maintaining software. In the mid 1960's, these problems were recognised and solutions to them were being developed. The first approach to better software design was through **Stepwise refinement** as proposed by Niklaus Wirth [Pressman 82]. A procedure is developed by decomposing it into smaller and

smaller tasks, until all the tasks can be expressed directly in terms of some programming language.

Structured Programming was proposed and formalised by Edsgar Dijkstra starting in 1968 [McGowan and Kelly 75]. One part of this was that all programs could be expressed through use of very basic program constructs. One of the benefits was that programs were inherently simple in structure.

The next development was **Structured Analysis**. This appeared in the early 1970's as a series of methods to specify a problem. The large traditional functional specification was replaced with diagrams, charts and tables, all designed to make the specification easier to understand and use. A variety of methods developed [Page-Jones 80] [Pressman 82], these included:

- Data Flow Diagrams (DFD)
- Hierarchical Block Diagrams
- Warnier Diagrams

The next development was **Structured Design**. One of the main design methodologies was proposed by Yourdon, Constantine and DeMarco [Page-Jones 80] [Stevens et al. May74]. The system is represented by a series of DFDs and is then transformed based on a methodology to arrive at a solution. This design was based on the data flow of the system and is called data flow-oriented design. Another design method called data structure-oriented design, causes the structure of the data to form the software design. Both methods have strengths and areas of where best used.

These developments have greatly improved the process of developing systems. They address the issues of analysis, design, coding and maintenance, but one area they have not

influenced greatly is the ability of the system to respond in a timely fashion to the modifications of the users. Using the various structured approaches, greater flexibility for software changes can be made, but changes must still go through the system development process. This means that corrections or improvements to a system can not occur as quickly as needed.

Application development tools have been developed and allowed for faster system development. These tools include such items as Code Generators, Graphic Specification Facilities, Forms Facilities for screen displays, Code and Module Libraries. Special tools called Fourth Generation Languages (4GLs) has allowed faster application development through the integration of common required facilities into tools, specifically a user interface and non-procedural programming [Misra and Jalics Jul88]. Generally 4GLs use databases as the data platform, and allow user access to them. Fourth generation languages originally appeared on the larger mainframe system, but have since moved to mini-computers and now to Personal Computers (PCs). Some current commercial 4GLs include PowerHouse, SQL*Forms, VAX Rally and Focus. 4GLs have decreased the development time throught prototyping but have had some inherent problems [Barker 87],[Misra and Jalics Jul88]. One basic and fundamental question remains for any system, *"Does the system do what the user wants ?"*. The answer should be yes when the system is initially implemented, but can the system change along with the user or with business needs? The answer is generally no. With current application development tools and the traditional development processes, systems generally can not remain current with user needs, because they can not change with sufficient speed.

Recently a new system has gained popularity in selected systems use, this being the **Expert System**. Many expert systems are rule-based systems, and allow direct input of user knowledge or domain expert knowledge into the system. These types of systems are more flexible to changes in the knowledge than the traditionally developed and coded systems. Expert systems can be driven by the rules, which are easily updated and easily expandable. This allows changes to be make quickly and the results seen almost immediately. Instead

of having programmers and analysts who have little understanding and knowledge of the subject area of the system being designed, develop a system, the users develop the system. The users are the subject matter experts who, through the knowledge engineer, translate their subject knowledge into rules. Most expert systems have a narrow knowledge domain, and are for very specific domain areas. Some examples of current expert systems are included in table 1–3 [Harmon and King 85]

Table 1–3: Well Known Expert Systems

System Name	Function
MICIN	Developed by Stanford University for use in diagnosis and treatment of meningitis and bacteremia infections.
DENDRAL	Used to examine a spectroscopic analysis of an unknown molecule and predict the molecular structures that could account for that particular analysis.
MACSYMA	Used to assist mathematicians, scientists and engineers in solving complex mathematical problems.
HEARSAY I, II	Developed at Carnegie-Mellon University to demonstrate the possibility of speech-understanding.
PROSPECTOR	Used to provide consultation to geologists in the early stages of investigating a site for ore-grade deposits.
PUFF	Used to interpret measurements from respiratory tests administered to patients in a pulmonary (lung) function laboratory.
XCON	Developed by Digital Equipment Corporation for use in configuring hardware in computer systems.
XSEL	Developed by Digital Equipment Corporation for use by sales people to analyse customer system specifications.
DELTA\CATS-1	Developed by General Electric Company for use to help maintain diesel-electric locomotives.

Unfortunately expert systems generally lack any useful form of database access facilities, which makes their direct use for an Office Information System virtually impossible.

The "office environment" is dynamic. It is constantly changing as new people enter the office, as new policies or procedures are implemented and as business changes are required. As a result of these facts, any office information system that is developed and implemented will consequently require changes through maintenance and modification of the software [Hägglund Aug88]. In the past and currently when using the traditional software development process, these changes seldom occurred, never occurred or caused problems in their implementation. Often when the changes in the system were finally implemented, new changes were pending [Page-Jones 80].

To provide user responsiveness, a User-Definable Office Information System has been developed. This system would provide an environment that would allow users to develop tasks directly or through direct input. It must provide a simple interface to access and manipulate data from a database. The interface of the User-Definable Office Information System is similar to fourth generation languages (4gls). It is window or screen oriented and does not require users to learn a query language such as SQL. The system would have features of an expert system, including production rules and use an inference engine. It would allow domain knowledge to be embedded in the system as production rules, from user input.

It is towards this end that this thesis examines the development of such an office information system. A small sample system is given by the author as an example of the use of this user-definable Office Information System.

CHAPTER 2
SYSTEM DESCRIPTION

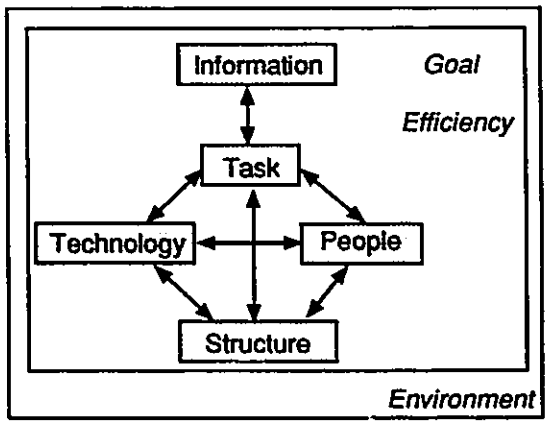
2.1 The Office Environment

> *"The conceptual design of an office information system constitutes a complex task. In fact, in designing an OIS, it is first necessary to define office and business goals, which are not always evident, in order to understand what the work globally performed in the office is, and how the system to be designed will affect the work. The analysis of office work that must be carried out in order to gain such knowledge is complex, due to the nature of office work itself, which consists of a large number of operational and decision support activities - in general, loosely structured - with many possible anomalies."* [Dachouffe and Lesuisse Aug88a]

The first step is to obtain a description of the office. This unfortunately is not a simple task. A complete and formal description of an office is not practical due to the large number of exceptions and special cases [Dachouffe and Lesuisse Aug88]. A possible cause for the exceptions in the office, is that the work tasks in the office are not standardised. Groups within the same office will have different methods of performing the same task, and thus complicating a formal description. A standard for processes, the *process standard*, could be developed to define the requirements of a process including input, and output. As a process is standardised, the required tasks are known and can be standardised as well. This would allow the office to be formalised and model with greater ease. Unfortunately very few offices have any formal process standards.

The office is the environment that many people work in. They have different functions and activities that they perform. An organisational model of the office as described by Leavitt [Leavitt 65] is seen in figure 2–1

Figure 2–1: A Model of the Office



"According to this model, an office is thus a social entity, composed of people , *joined together for a certain time to achieve the same* goal. *For this purpose, these persons perform different* tasks *(in particular, information processing tasks), by eventually making use of some* technologies. *People, tasks and technologies are integrated in a* structure *which determines the relationships between the tasks and the role of different persons, and which must also normally allow the social entity to work with* efficiency.

As every organisation, every office belongs to an environment *composed of other offices which it exchanges information with."* [Dachouffe and Lesuisse Aug88]

In the office environment, a structure exists which defines the individuals within the office and the tasks that perform and how they interact. The relationships between the individuals can be based on

- Who the individual reports to
- The task each performs
- The inputs and outputs of the tasks each performs

With an office structure that is so complex and variable, any Office Information System (OIS) must allow incorporation of this complex and variable structure within its design.

2.2 Office Information System Models

Models of Office Information Systems can be classified into four categories [Bracchi and Pernici Apr84]. These are listed in table 2–1.

Table 2–1: Office Information System Models

Model Type	Description
Data-based	Generally group data into *forms*, which are similar to paper forms. The office is represented from the view of object manipulated by the office workers.
Process-based	The office is described by the activities performed concurrently by the users. The office is represented as the integration of all the activities perform in an office in order to perform certain tasks.
Agent-based	An office is modelled from the viewpoint of the functions performed by active elements (users). This includes users tasks, the data to be accessed and the relationships with other users. It bases the data and tasks on the organisational structure.
Mixed	Assume more that one type of element as the basis of the system specification.

The User-Definable Office Information system is in the *mixed* model category. It does not have any specific bias for developing tasks, but could allow tasks to be defined in multiple methods.

The tasks in the office can be categorised differently. One category is based on the task being placed in one of two areas depending on whether it is a

- General Office Activity
- Job Specific Activity

The **general office activities** would include Word Processing or Document Processing, Electronic Messaging, Time/Calendar Management. These are activities that are common to all people in the office. Individuals may have great or little need of the services depending upon their job. Managers generally use Electronic Messaging for daily communication with other managers and staff to obtain information and direct work activities. Secretaries use Word or Document Processing in the daily generation of letters and memos. There has been research in this area recently, and some articles on the subject include [Yang 88], [Güting 89] and [Witten and Bramwell 85]. This system does not incorporate any general office activities. However, they could be added to the base system using the same facilities as the job specific.

The **job specific activity** area differs from the general office activity area in that it consists of tasks specific to a job or function. The amount of job specific activities varies for different individuals in an office, and for different offices. These applications are traditionally developed by MIS or EDP organisations, written by outside consultants or purchased as products. These applications include functions such as:

- Customer Order systems
- Invoicing systems
- Manufacturing systems
- Enrolment systems
- Forecasting systems
- Spreadsheet systems
- Accounts Receivable

- Accounts Payable

These applications are developed using the traditional application/system development process, which is common to most MIS and EDP organisations and software consultants. The application will be generally coded in a third generation programming language (3GL) such as COBOL or PL/1 [Martin 85]. These systems will frequently develop and experience the problems as noted in Chapter 1.

The User-Definable Office Information System developed by the author would provide an environment to allow users to develop their own applications. The system would allow user domain knowledge to be embedded in the system, as in an expert system and allow users to access data with little knowledge of the physical data format. In order to achieve this, the author has implemented the system in the Artificial Intelligent (AI) programming language of COMMON LISP. The system must provide facilities for

- Task Selection
- User Definition
- Data Interface
- Data Access System
- Task Definition

The *Task Selection* facility allows the user to select and perform different work tasks. It generally takes the form of either a command interface (similar to the DCL environment in VMS or MS-DOS or a UNIX shell) or a menu interface (either simple or graphical). For the office environment the menu interface is frequently used since it is simple and structured for users. This type of interface is used in Digital Equipment Corporation's ALL-IN-1 Office system, in LOTUS 1-2-3, or the MACINTOSH computer from Apple Corporation.

The *User Definition* facility allows users to be defined for the system. Each user will have characteristics and functionality that should be similar to their position and work in the office.

The *Data Interface* facility allows users to enter, view and manipulate data. Generally this is a screen or window oriented display as opposed to a line-by-line. Evidence of the importance of this area in applications is in extensions to certain third generation programming languages (3GLs). Digital Equipment Corporation's version of COBOL includes extensions to the WRITE and ACCEPT statements to include direct cursor addressing of the screen. In the C programming language, there are the "cursor" routines for screen input and output. The recent development of X-Windows [Scheifler et al. 88] at the Massachusetts Institute of Technology, and its acceptance as a defacto standard in windowing by major computer vendors including Digital Equipment Corporation and Hewlett-Packard Corporation indicates the importance of the window environment in future software.

The *Data Access System* facility is used to perform data accessing. The data platform could be files supported by the file system, or it could be a database management system. The data access system is hidden from the user, but does affect the programming and design of the proposed system. Access methods for users may not be directly supported by the data access system, and may require extra programming to develop a required access method.

The *Task Definition* facility allows job related tasks to be created and performed by the system. This is perhaps the most important part of an information system.

With COMMON LISP as the language of implementation, all of the previous facilities had to be developed, as they are not native parts of COMMON LISP. The User-Definable Office Information System is composed of the components in table 2–2.

Table 2–2: System Components

Required Component	System Component
Task Selection	Menu System, Windowing System
User Definition	Flavour System
Data Interface	Database System Interface
Data Access System	User-Database Interface System, Windowing System
Task Definition	Production Rules and Inference Engine, Windowing System

2.3 Implementation

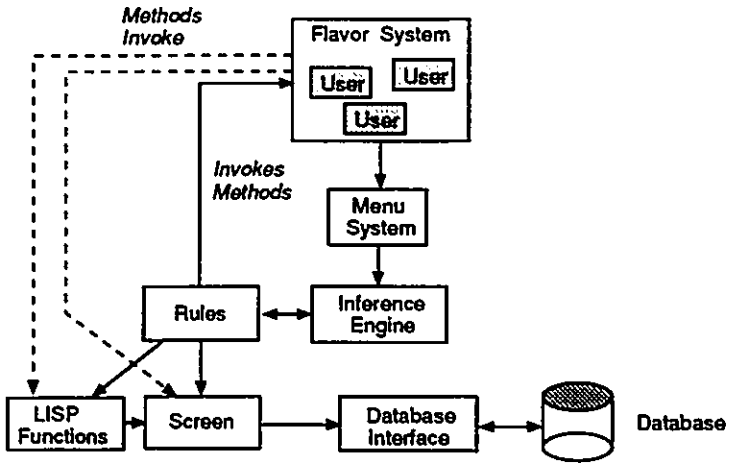
The proposed system is implemented on a VAX computer system running the VMS V5.3 time-sharing multi-user operating system. In this environment each user on the VAX is a "process" and can run different executable images. The User-Definable Office Information System is stored as a suspended LISP session that is restarted when the user invokes the system. Each user of the system loads and runs a separate copy of the same suspended LISP session. Portions of the LISP software can be shared between users for improved performance and reduced memory usage. There is no direct communication between different users of the system when running concurrently.

Data for the system is read from a series of text files stored in a common directory and in the user's own directory. This information includes

- User Definition
- System Menus
- System Rules
- User Menus
- User Rules

2.4 System Components

Figure 2–2: System Components



2.4.1 Flavour System

A flavour system is used to define the system users. The flavour system which is used is based on a flavour system previously developed by the author. A flavour system allows objects (users) to be defined and various methods (tasks) to be associated with each. The flavour system allows the definition of the users for the system to model the actual office structure in terms of individual interrelationships and functionality.

2.4.2 Windowing System

A system is required for perform input and output. A window environment is in common use today in program develop and application, this is seen through products like MS-Windows, X-Windows , and Hewlett-Packard's New Wave products. The system was designed to run using non-graphic type terminals. A window system was developed to emulated the windowing environment for a non-graphic terminal.

2.4.3 Menu System

A simple non-graphical menu system is used to define the menus that are used to provide application selection. Many office systems are generally menu oriented. The menu system is designed to allow the users to create their own menus for access to the system.

2.4.4 Production Rules and Inference Engine

Production rules and an inference engine are used for task definition and execution. The rules define the activities that the system will perform. The rules are defined from user input to form the tasks in the system. Tasks can be executed by many users, or by just one. User specific rules can also be defined to supplement and customise an otherwise common task.

2.4.5 User-Database Access

The user-database interface is a "*screen*". The screen defines fields in which the user will see information displayed. Each screen has a specific function associated with it that allows the user to perform only that function on the data displayed.

2.4.6 The Database

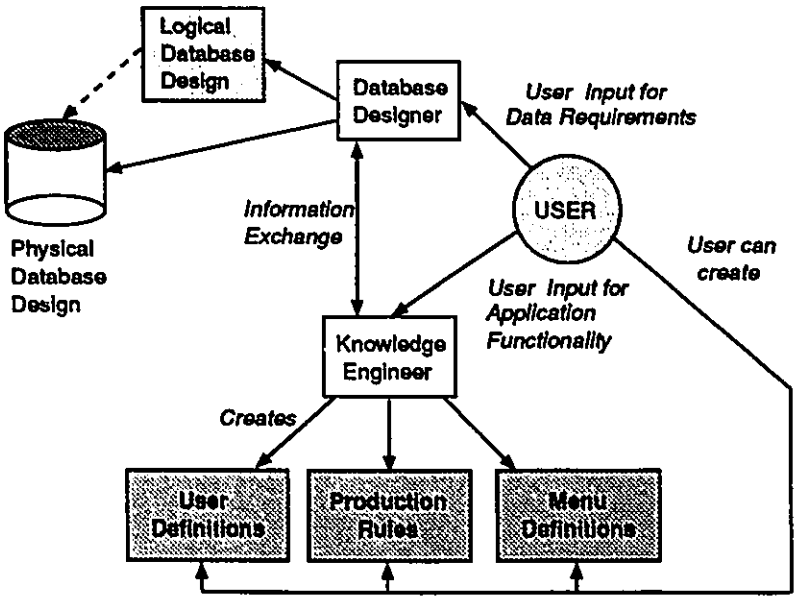
A relational database system was used for data storage. The database system used is Rdb/VMS V3.0 which is a Digital Equipment Corporation software product that runs on VAX machines under VMS/VMS. A relational database was chosen, since it provides a flexible and simple environment to define, access and model data. It provides users with a great amount of flexibility in linking data together which is often required for user tasks.

2.4.7 User Support

The author believes that users of this system, with the necessary user level training and additional technical support, can develop an Office Information System. The additional support personnel as seen in figure 2–3, is required by the users to help perform the following technical functions of the system.

- Logical Database Design and Normalisation
- Physical Database Design
- User definitions
- System rule definitions for work tasks

Figure 2–3: Support Personnel

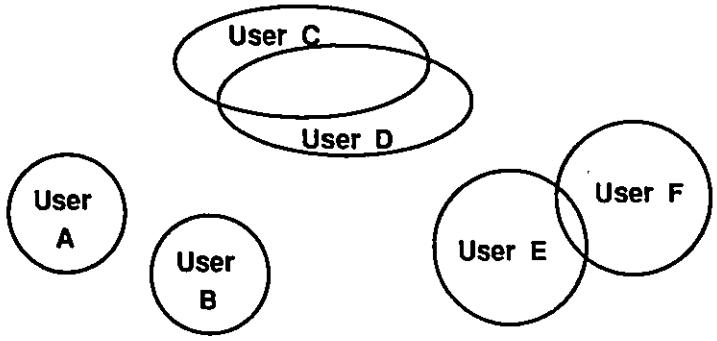


The role of the designer, analyst and programmer in the traditional system development process are replaced by the knowledge engineer and the database designer.

CHAPTER 3
SYSTEM USER DEFINITION

The office structure which includes the individuals and the tasks they perform, is difficult to generalise [Bracchi and Pernici Apr84]. Different office environments involved in similar work activities may have very different structures. The tasks that individuals perform vary with their job in the office. Interrelationships exist between different workers with regards to the data they can access and the tasks they can perform. These relationships could be described in figure 3–1

Figure 3–1: Office Relationships

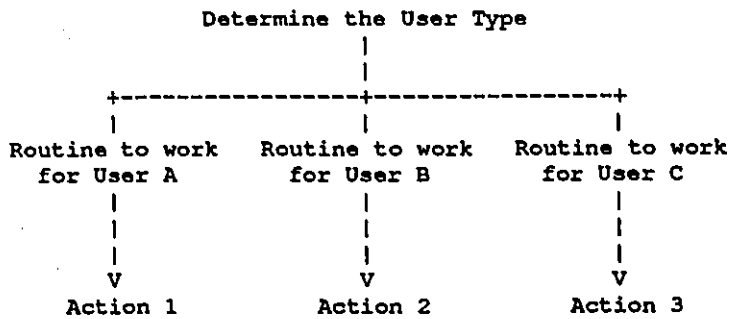


Workers A and B have no interrelation at all, other than they work in the same office. Workers C and D have a large interrelationship, in contrast to worker E and F who have a small interrelationship.

In order to have flexibility in the definition of system users, a **flavour system** was chosen to be used to define each of the different users of the system. A flavour system is an object-oriented programming system involving message passing, with the addition of inheritance between objects. A flavour system allows object classes to be defined, and have associated with each object class a set of methods (tasks) and attributes. Each user appears to the system as an instance of an object class. Object-oriented programming can be used when programming in LISP and is an integral feature of SMALLTALK.

A benefit of the flavour system, resulting from object oriented programming, is the ability to define operations that are independent of their operands. The traditional "operator/operand" model [Cox Jan84] has a separate operator to handle each different type of operand. For an application, this could involve having a separate function or subroutine for each user that is to perform a function. This is illustrated in figure 3–2.

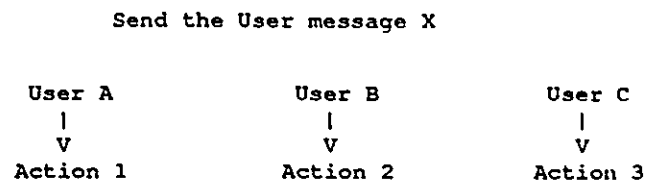
Figure 3–2: Non Object oriented Programming



In object-oriented programming, the operators are called *methods* and the operands are *instances* of object-classes (a user). Programming becomes sending a message to an instance of an object. The message is interpreted as a method to be executed for the instance. This allows the same message to be handled differently for different classes of objects (users) as seen in figure 3–3. The origin of the method request does not need to change for different

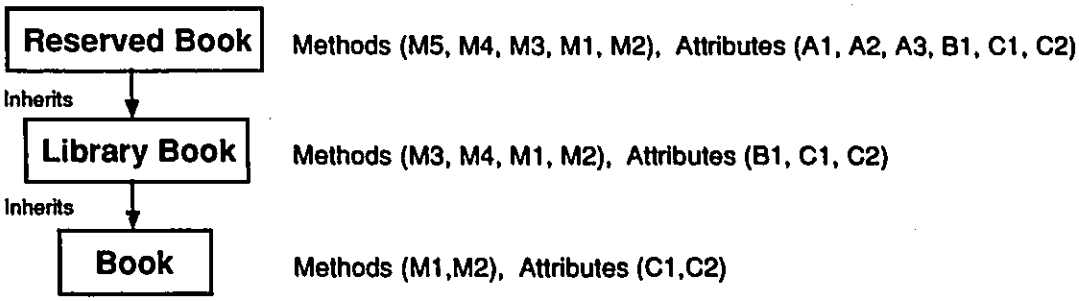
object classes, it can be the same just applied to different objects. Tasks specified in terms of methods can then be defined generically for many different users (object-classes), which simplies task implementation.

Figure 3–3: An Object-Oriented System



A feature of flavour systems is the ability for object classes to inherit methods and attributes from other object classes. A simple example of this is seen in figure 3–4.

Figure 3–4: Flavour Concepts



In this case there are three object classes (BOOK, LIBRARY BOOK and RESERVED BOOK). The object class RESERVED BOOK is a superset of the class LIBRARY BOOK which is a superset of the class BOOK. The class RESERVED BOOK has inherited the methods and attributes of the other classes.

Each occurrence of a class object is called an instance (or element of a class of objects). A class of objects X may be generalised as

$$X(a, m)$$

where a is the set of attributes of the class $a_1, \ldots, a_x$ and m is a set of methods of the class $m_1, \ldots, m_y$. An instance F of an object is said to belong to the class of objects

$$F \in X$$

With the flavour system, the various classes of object can be related through inheritance. If A is a class of objects that is a subclass of the class of objects B, then this can be expressed as

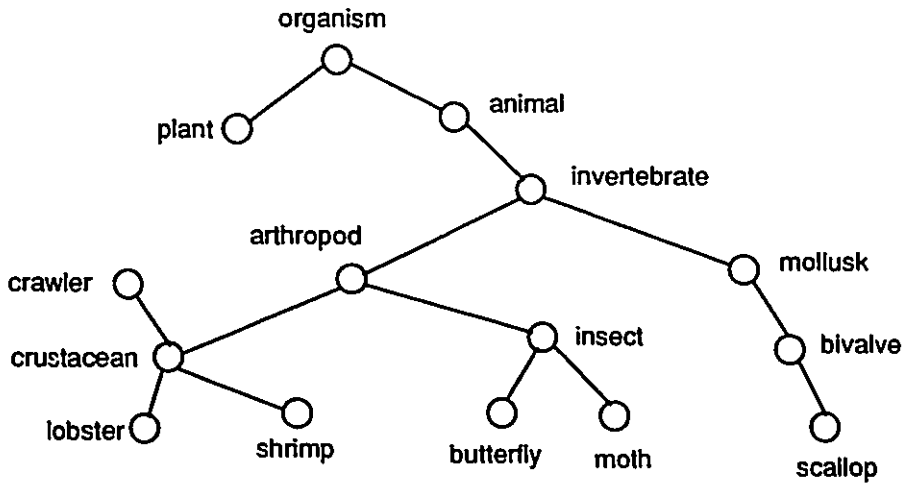
$$A \subseteq B$$

This is commonly referred to as the *IS-A* property. For figure 3–4, the relationship between the three classes could be stated as

$$BOOK \subseteq LIBRARY\ BOOK \subseteq RESERVED\ BOOK$$

In a general form the inheritance hierarchy can be seen in figure 3–5.

Figure 3–5: Sample Inheritance



In this figure the most basic class of objects is *organism*, called the *root*. All other classes of objects inherit its methods and attributes. The highest class of objects are *crawler, lobster, shrimp, butterfly, moth, scallop* which are called *leaves*. Between the root and the leaves there can be a number of intermediate classes of objects. The class *shrimp* can be stated as

$$organism \subseteq animal \subseteq invertebrate \subseteq arthropod \subseteq crustacean \subseteq shrimp$$

A benefit of using the flavour system and defining classes of users, is to help avoid separately implementing common features for each different user. Instead the methods associated with one class of object can be inherited from many other classes.

With the ability to inherit attributes and methods, a user can be defined in terms of other users (or object-classes). This is a common reality for tasks and characitistics of individuals in the office environment. The methods inherited can be overridden by defining a new method with the same method-name at a high level in the inheritance tree structure. The

methods defined for different users may be the same across the different user thus simplifying implementation and maintenance.

3.1 Flavour System Details

The flavour system in use was one developed by the author earlier and was modified for use in this system. It consists of several LISP functions that can be used to define flavours, methods, perform the message passing and create instances of a flavour. Users can use these LISP functions to define their office structure. The functions are listed in table Table 3–1. When the user invokes the system, user definition files are accessed that contain function calls that define the user to the system.

Table 3–1: Flavour Functions

Function Name	Description
DEFLAVOR	Create a flavour, including attributes and any flavour inheritances
DEFMETHOD	Creates a method and associates it with a flavour
MAKE-INSTANCE	Creates an instance of a particular flavour
SEND	Sends a message to an instance of a flavour

3.1.1 DEFLAVOR Function

This function is used to define a flavour. Its calling format is seen in figure Figure 3–6.

Figure 3–6: DEFLAVOR function arguments

```
(deflavor   'flavour-name
            '(instance-variables)
            ( 'inheritance-flavour | nil )
            '( options ) )
```

The *flavour-name* is the name of the new flavour to be created. Should a flavour with that name already exist, a message is displayed informing the user and the deflavour is not defined.

The *instance-variables* represent the attributes that are associated with the flavour and exist for each instance of the flavour. Each attribute can take on a value as variables in a program.

The *inheritance-flavour* is the name of a previously defined flavour. The current flavour will inherit all the attributes and methods of inherited flavour.

The *options* are used to specify characteristics of the attributes. Three possible options exist as listed in table Table 3–2.

Table 3–2: Attribute Options

Option	Description
GETTABLE	This option will create an automatic method for each specified attribute that returns the current value of the attribute. The name of the method is the attribute name.
INITABLE	This option allows the value for an attribute to be specified when it is created with the MAKE-INSTANCE function.
SETTABLE	This option will create an automatic method for each specified attribute that allows the value of the attribute to be set. The name of the method is SET-*attribute-name*.

If only the option keyword is specified then it applies to all attributes. An option may apply to a subset of attributes, but each attribute must be explicitly specified. This is seen in example 3–1.

Example 3–1: Attribute options

```
(deflavor 'box
          '(length width height weight)
          nil
          '(gettable (initable length width)
                     (settable weight) )
          )
```

In this example the all the attributes are GETTABLE, only *length* and *width* are INITABLE, and only *weight* is SETTABLE.

3.1.2 DEFMETHOD Function

This function is used to define a method for a flavour. Its calling format is seen in figure Figure 3–7.

Figure 3–7: DEFMETHOD function arguments

```
(defmethod   flavour-name
             'method-name
             '( method-variables )
             '( method-body ) )
```

The *method-name* is used to identify the method being defined. The *method-name* must be unique. If it duplicates a method already defined for this flavour, the last definition is the one that is used. If the *method-name* is the same as a method defined in any of the inherited methods from other flavours, the new method supersedes the previous.

The *method-variables* represent a list of LISP variables used to accept information from the SEND function. These variables take on the values associated with the SEND function and can be referenced by the statements in the *method-body*. This list can contain anything found in the parameter definitions in the LISP function DEFUN.

The *method-body* represents a series of LISP functions that will be executed when the method is invoked. The functions can be any LISP function of COMMON LISP, or any of the various support pre-defined LISP function of the system.

3.1.3 MAKE-INSTANCE Function

This function is used to create an instance of a flavour. Its called format is seen in figure Figure 3–8.

Figure 3–8: MAKE-INSTANCE function arguments

```
(make-instance   'flavour-name
                 'attribute1 value
                 'attribute2 value ... )
```

The final form of this function is the instance of the specified flavour. This should be assigned to a variable in LISP through the SETQ or SETF function as illustrated in example Example 3–2.

Example 3–2: Using MAKE-INSTANCE function

```
(setf a (make-instance 'box 'length 5 'width 5 'height 6))
```

If an *attribute* has been defined as INITABLE, then it can be specified after the flavour name. Following the *attribute*, a form can be specified, which will be evaluated, and the resultant value assigned to the *attribute*.

SEND Function

This function is used to send an instance a method to be invoked. Its calling format is seen in figure Figure 3–9.

Figure 3–9: SEND function arguments

```
(send    instance-name
         'method-name
         arg1 ... argn)
```

This function will send the *method-name* to the instance of the object class for processing. If the *method-name* is found then the method is executed with the arguments *arg1 ... argn* matched to the *method-variables* associated with the method. If the method specified is not defined in the current flavour, its inherited flavour is then searched, and this process continues until all flavours in the inheritance structure have been searched. If no method can be defined an error method is invoked which displays a message indicating the requested method does not exist.

Additional Features

Different flavour systems have a number of common features. The system in use has the following features.

Base Flavour

The first flavour definition, inherits a flavour called VANILLA. The flavour VANILLA has a number of methods associated with it, as listed in table Table 3–3. The user does not have to explicitly specify these methods in any definition.

Table 3–3: VANILLA Flavour Methods

Method Name	Description
INIT	The method is only used when an instance of a flavour is created.
WHICH-OPERATIONS	This method accepts no arguments, and returns a list of all the methods accessible to this instance. This includes BEFORE and AFTER flavours. These are in the form "flavour-AFTER" and "flavour-BEFORE".
EVAL-INSIDE-YOURSELF	This method accepts one argument which is a form that will be evaluated within the context of the instance.
FUNCALL-INSIDE-YOURSELF	This method accepts two arguments, the first is a LISP function name and the second is a series of arguments to the function. The function will be executed with the specified arguments in the context of the instance.

Before and After Methods

A flavour can have a BEFORE and AFTER methods. These are methods that are executed either before or after the execution of any other method specifically associated with that flavour.

In defining a BEFORE and AFTER methods, the method name specified is either BEFORE or AFTER. If the method does not require any parameters, then the following must be specified for the method-variables

```
'( &REST JUNK)
```

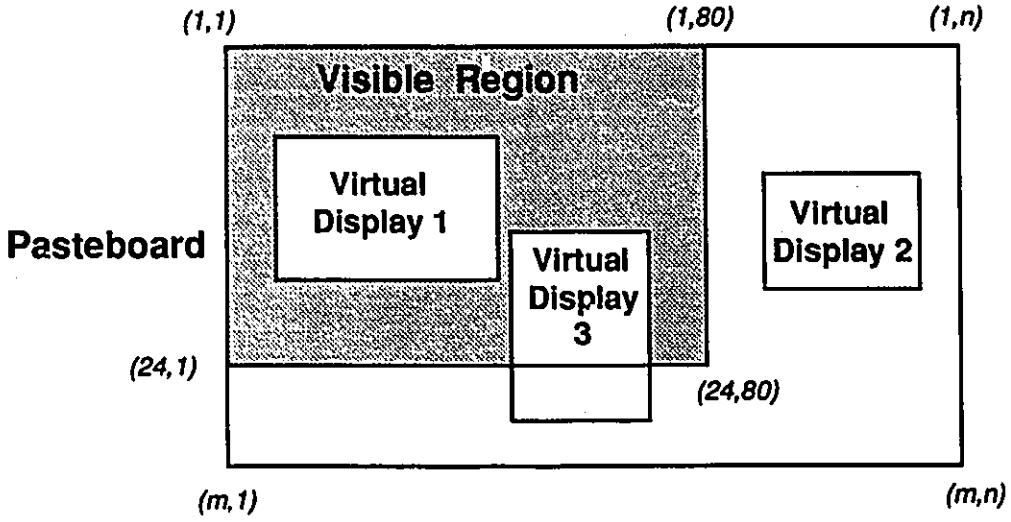
Returning Values from Methods

If a method is to return a value, then this must appear as the final form of the *method-body*. This value will then be returned from the calling of the SEND function, and can be assigned to a variable with the SETQ or SETF function.

CHAPTER 4
WINDOWING SYSTEM

The VMS Operating system (V5) has a series of Screen Management Routines (SMG). These routines provide terminal independent input and output. A feature of the SMG routines is that of providing a "window environment". SMG allows windows (called **virtual displays**) to be established and manipulated. Windows can be read from, written to, moved and manipulated. These SMG routines provide the facilities used to develop the menu system and the user-database interface.

Figure 4–1: SMG Concepts



4.1 Virtual Displays

This is the "window" that can be created and displayed on the screen. All input and output is performed to a virtual display. Each virtual display has a unique identification number associated with it which is used to identify it in input and output operations. Virtual displays are defined in terms of rows and columns, and have a (x,y) coordinate system to identify each location. The base coordinates are (1,1) and identify the upper left position of each virtual display.

4.2 Pasteboard

The **pasteboard** is a virtual structure, whose function is similar to a notice board. All virtual displays must be placed on the pasteboard in order to be used for input and output. The user's terminal is mapped to the a portion of the pasteboard. The pasteboard can vary in size, and is measured in terms of a number of rows and columns. If its size is 24 rows and 80 columns, then the pasteboard maps the full terminal, and all of the virtual displays mapped to this pasteboard are visible. However, a pasteboard can be created larger than the terminal. When this happens, the terminal maps the upper left region of the pasteboard as rows 1 to 24 and columns 1 to 80. Any virtual displays placed entirely in the visible region are completely visible, as Virtual Display 1 in figure 4–1. All pasteboard locations outside the visible region are invisible and any virtual display placed in this area is not visible, as in Virtual Display 2 in figure 4–1. Non-visible virtual displays are maintained by the SMG routines and output operations can be performed to them. A virtual display can be placed on the **pasteboard** in such a manner that it is both partially visible and invisible as in Virtual Display 3 in figure 4–1.

In the menu system each MENU is implemented as a virtual display. The MENUs are then placed on the invisible portion of the pasteboard. As requested by the user, they are then moved into the visible portion. One feature of the SMG routines is that virtual displays can

be stacked with the most recent being visible (if they are all directly on top of each other and the same size). By removing the top virtual display, the one immediately below it becomes visible. This feature is managed entirely by the SMG routines and is used as an integral part of the menu system, since it means no redisplaying of the previous MENUS is required by the system.

4.3 Window Functions

A series of LISP functions were created to map to the various SMG routines and perform the basic SMG functions and to create higher level functions by combining SMG routines together. Table 4–1 lists the various LISP functions.

Table 4–1: LISP functions

LISP function	Description
CREATE-AND-MAP-WINDOW	Create a virtual display and map it to a row and column position on the pasteboard
CLEAR-WINDOW	Erases the current contents of a virtual display
WRITE-WINDOW	Write to the current cursor position in a virtual display
WRITE-WINDOW-AT	Write text at a specific location in a virtual display
READ-FROM-WINDOW	Read from a virtual display
READ-FROM-WINDOW-TERM	Read from a virtual display and return the terminator character entered
REMOVE-WINDOW	Deletes a virtual display from the pasteboard

The LISP functions provide the user the facilities to create windows and perform input and output from a LISP function or flavour method. They can then be used in the development of tasks. The window functions are also used by the Menu System and the User-Database

Interface. The LISP functions are discussed in detail below. The term "window" is used as a generic term for the SMG virtual displays.

4.3.1 CREATE-AND-MAP-WINDOW

This function will create a window and it map to the pasteboard, usually in the visible portion. It is the first function to be called. The pasteboard is created transparently by the system and is always available for use in mapping virtual displays. The return value and parameters are listed in table 4–2.

Table 4–2: CREATE-AND-MAP-WINDOW Function

Parameter Name	Data Type	Description
Return Value		
	Integer	Virtual Display ID
Parameters		
Rows	Integer	Number of rows in the window
Columns	Integer	Number of columns in the window
X	Integer	Pasteboard row location for window
Y	Integer	Pasteboard column location for window
Window Title	String	Title to centre on top border

4.3.2 CLEAR-WINDOW

This function clears the contents of a window. The parameters are listed in table 4–3.

Table 4–3: CLEAR-WINDOW Function

Parameter Name	Data Type	Description
Window ID	Integer	Window ID of the window to clear

4.3.3 WRITE-WINDOW

This function writes a string to a window. The function writes to the next line on the window, so that no row or column for output can be specified. The parameters are listed in table 4–4.

Table 4–4: WRITE-WINDOW Function

Parameter Name	Data Type	Description
Window ID	Integer	Window ID of the window to write to
Text	String	Text to write to the window

4.3.4 WRITE-WINDOW-AT

This function writes a string to a window at a specific row and column. The parameters are listed in table 4–5.

Table 4–5: WRITE-AT-WINDOW Function

Parameter Name	Data Type	Description
Window ID	Integer	Window ID of the window to write to
Text	String	Text to write to the window
Row	Integer	Row of the virtual display to write the text
Col	Integer	Column of the virtual display to write the text

4.3.5 READ-FROM-WINDOW

This function reads a string of characters from a window. An optional prompt may be specified. The return value and parameters are listed in table 4–6.

Table 4–6: READ-FROM-WINDOW Function		
Parameter Name	Data Type	Description
Return Value		
	String	Input string
Parameters		
Window ID	Integer	Window ID to read from
Maximum Length	Integer	Maximum length of input string
Prompt	String	Prompt string (in double quotes) to be displayed

4.3.6 READ-FROM-WINDOW-TERM

This function reads a string of characters from a window. An optional prompt may be specified. The return result is a list of two items, the first being the terminator character, and the second is the user entered data. The return value and parameters are listed in table 4–7.

Table 4–7: READ-FROM-WINDOW-TERM Function		
Parameter Name	Data Type	Description
Return Value		
	List	Input string
Parameters		
Window ID	Integer	Window ID to read from
Maximum Length	Integer	Maximum length of input string
Prompt	String	Prompt string (in double quotes) to be displayed

4.3.7 REMOVE-WINDOW

This function deletes a window from the pasteboard. This is the final function called once use of a window is completely finished. The parameters are listed in table 4–8.

Table 4–8: REMOVE-WINDOW Function

Parameter Name	Data Type	Description
Window ID	Integer	Window ID of the window to delete

Example 4–1 is a a fragment of LISP code illustrating some of these functions. The example creates a window and display some data to it and then reads some input from the user.

Example 4–1: LISP window functions

```
(setq window-id (create-and-map-window 10 55 2 2 "Main Window"))
(write-window window-id "Here is some text")
(write-window window-id "Here is more text")
(setq input (read-from-window window-id 25 "Enter your name: "))
```

CHAPTER 5
MENU SYSTEM

5.1 Menu System

The task selection system is required to allow users to select and execute tasks. Many business/office systems use a menu driven system. This means that there is a series of menus to allow users to choose tasks to perform. The menus provide a hierarchical structure of functionality. This is generally the preferred interface for information systems. The alternative is a command language interface, which is generally more complex to use since the user must learn the appropriate commands.

Many information systems provide a multi-level menu interface for users. Feedback from users usually includes comments that they do not like the particular hierarchy that is established for them. Certain functions that the user performs are on different segments of the hierarchy, and require timely menu jumping. One goal of the User Definable Office Information System is to allow users to define their own menus and thus tailor their user interface. As users first start using the system, the menu structure may be structured based on data entities (ie orders, customers, salesmen). But as users becomes more proficient with the system, they may wish to change the menus to better suit their specific needs. They could group frequently use options together on the same menu, even though the options perform non-related tasks.

5.2 Implementation

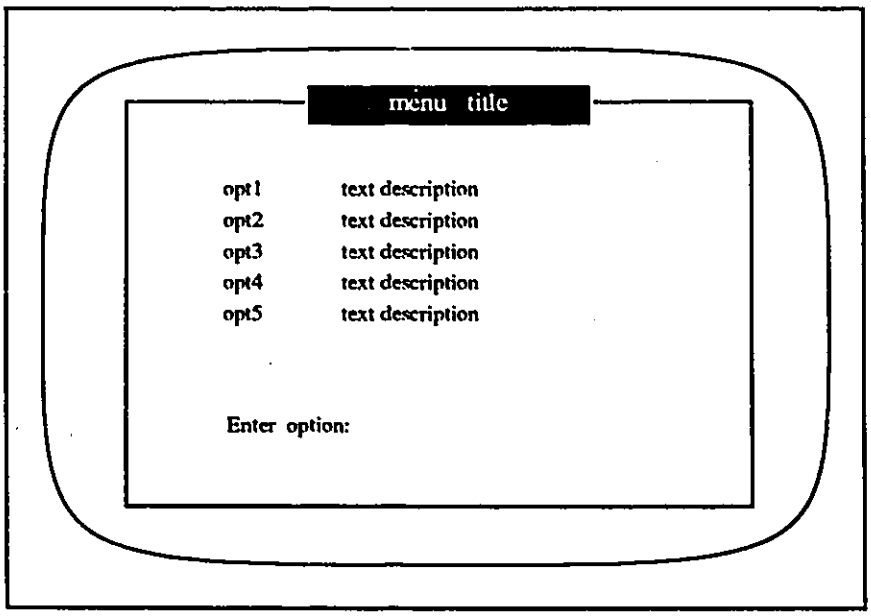
The menu system consists of the following components

- Menu Specification Language
- VMS Screen Management Routines
- Menu Building Routines

5.3 Menu Concepts

A typical menu from the system is a window is illustrated in figure 5–1.

Figure 5–1: A menu



The window has a border around it, with the menu name centred at the top. Each option in the list of options has a corresponding text description.

5.4 Specification Language

The MENU specification language allows MENU structures to be specified in an easy manner. Menu definition is missing from most high level languages. If required, they must be developed by the designers/programmers. There are generally no standards used in developing menus. The MENU specification provides the facility to easily define a MENU and the various OPTIONS of that menu. When a menu is invoked, the user is displayed a list of possible OPTIONS to choose from. The user enters a choice, and it is validated against the possible options. Once validated the menu option will perform one of two possible activities:

- Invoking another MENU
- Invoking some task

The MENU specification language used is keyword specification language. The language consists of two commands and a series of keywords associated with each. A command keyword must start the line, followed by the remaining keyword and value pairs. The command is then terminated with a semicolon (;). One menu command is for defining the MENU and its characteristics, and the second is for defining each OPTION in a MENU.

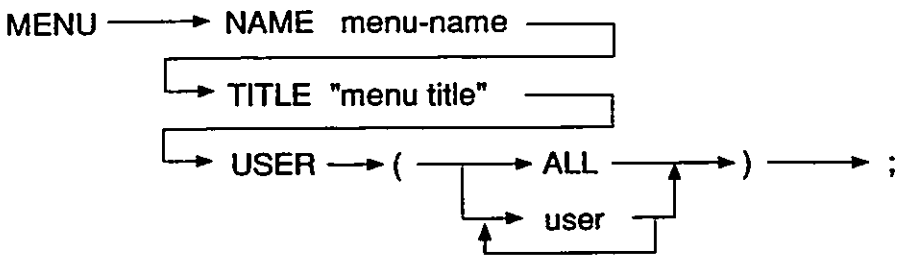
5.4.1 MENU Definition

The MENU command syntax is seen in figure 5–2.

NAME keyword

This specifies the name of the MENU. The name usually should give some indication of what the MENU is used for.

Figure 5–2: MENU definition syntax



TITLE keyword

This specifies a title that will be centred and displayed on the top border of the MENU. It should contain a description of the MENU function. The title must be enclosed within double quotes.

USER keyword

This specifies which class(s) of system users can have access to the MENU. If a system user does not have access to the MENU, then it is not created for when the system is initially invoked. The user is the value of the attribute I-AM-A for the user's instance.

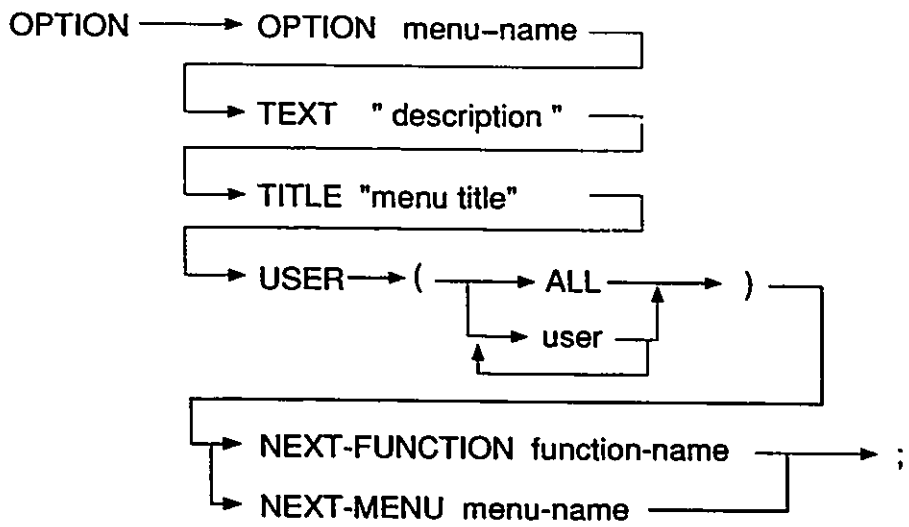
5.4.2 OPTION Definition

The OPTION command syntax is seen in figure 5–3.

OPTION keyword

This is the actual menu option, the characters that represent the choice. Generally it is 1 or 2 characters that is used to identify the function. For example D could be used for a delete function, E for an Edit function, etc...

Figure 5–3: OPTION Syntax



MENU keyword

This is the name of the menu for which this OPTION applies. The MENU must have been previously defined. However the OPTION command need not follow directly the corresponding MENU command.

USER keyword

This specifies which class(s) of system users can have access to this OPTION. If a system user does not have access to the OPTION, then it is not created on the menu for when the system is initially invoked. A system user may have access to a menu but not all the OPTIONS on the menu. The user is the value of the attribute I-AM-A for the user's instance.

TEXT keyword

This specifies the text description of the MENU option. It is restricted to 50 characters in length. It is displayed several spaces to the right of the option characters in the MENU.

NEXT-FUNCTION keyword

This represents the name of the LISP function to invoke when the user chooses this menu option.

NEXT-MENU keyword

This represents the name of the next menu to display when the user chooses this menu option. A check is made only at execution time as to whether the menu actually exists.

5.4.3 Implemenation

The menu system is implemented using the Window system to create and manipulate windows. Each MENU is implemented as a window (virtual display). The MENUs are then placed on the invisible portion of the pasteboard. As requested by the user, they are then moved into the visible portion. One feature of the SMG routines is that virtual displays can be stacked with the most recent being visible (if they are all directly on top of each other and the same size). By removing the top virtual display, the one immediately below it becomes visible. This feature is managed entirely by the SMG routines and is used as an integral part of the menu system, since it means no redisplaying of the previous MENUS is required by the system.

5.5 Menu Building Routines

A series of LISP functions were created to map to the various SMG routines and perform the basic SMG functions and to create higher level functions by combining SMG routines together. Table 5–1 lists the various LISP functions.

Table 5–1: LISP functions

LISP function	Description
CREATE_MENUS	Creates menus from the specified menu file.
MENU	Envokes the menu system using the specified menu as the starting menu.

5.5.1 CREATE_MENUS Function

This function builds the menus from a menu specification file. The function has a single argument which is the name of the menu specification file. Example 5–1 illustrates calling this function.

Example 5–1: Calling the CREATE_MENUS function

```
(CREATE_MENUS "user.menu")
```

5.5.2 MENU Function

After the menus have been created, control must then be transfered to the menu system to allow processing. The MENU function is used to perform this. The function has a single argument which is the name of the menu to display. Example 5–2 illustrates calling this function.

Example 5–2: Starting the MENU system

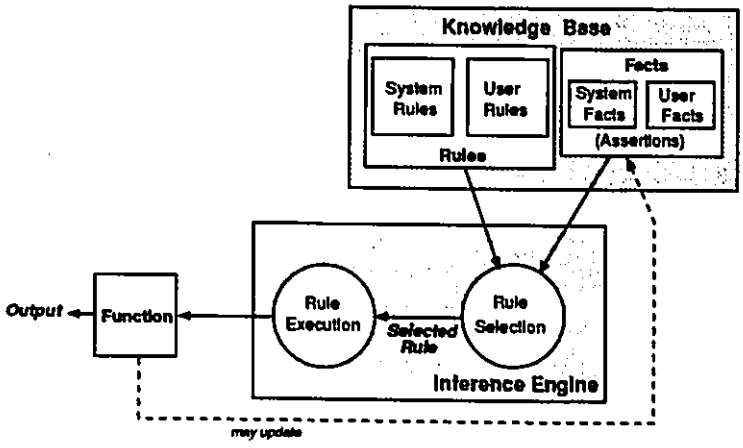
```
(CREATE_MENUS "my_menus.menu")
(MENU 'MAIN)
```

CHAPTER 6

THE KNOWLEDGE BASE AND KNOWLEDGE PROCESSING

A **knowledge base** exists for the system which contains two kinds of knowledge. First, it contains **productions rules** which when executed will produce a sequence of actions allowing individual parts of a task to be performed. The rules specify a set of conditions and the resultant actions to occur when all the conditions are met. Rules can be defined by the knowledge engineer with direct user input. These are called **System Rules** and they define most of the actions of a task. Additional rules can also be define for or by users, called **User Rules**. User Rules are designed to supplement but not override the System Rules. The knowledge base also contains **facts**.

Figure 6–1: Knowledge Base and Inference Engine

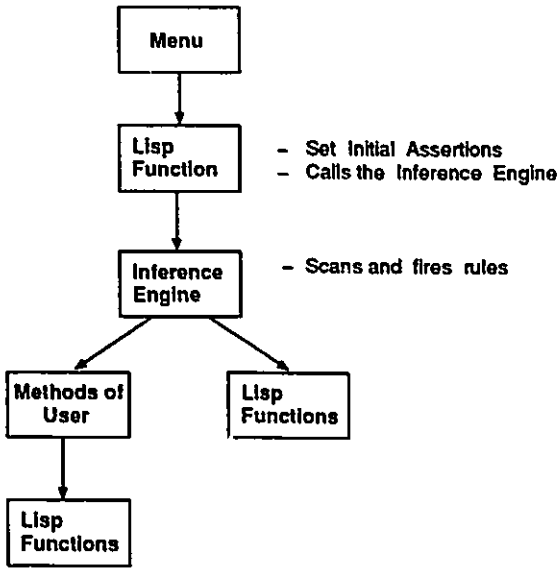


In the system, facts are called **assertions**. The conditions in the rules test the facts. The interaction of rules, assertions and the inference engine is illustrated in figure 6–1

6.1 Invoking the Inference Engine

When the system is invoked, the user selects a task through the menu system. Once the user has selected a menu option that performs a task, a LISP function is called that sets up the initial conditions for the inference engine as a series of one or more assertions. The inference engine is then invoked to perform rule processing. This sequence is illustrated in figure 6–2.

Figure 6–2: Control Flow



Users are not always under the control of the inference engine, rather only as tasks are actually selected. Once a task has completed through the executing of all the possible rules, control is returned to the menu system at the last menu invoked.

6.2 Rule Structure

A rule consists of a *premis-action* pair, for example

$$if\ P_1 \wedge \ldots \wedge P_n\ \ then\ \ Q_1\ \&\ \ldots\ \&\ Q_n.$$

and is read as "if premises P_1 and ... and P_n are true, then perform actions Q_1 and ... and Q_n. The P_i are called *conditions* and the Q_i are called *conclusions*. The *conclusions* in this system are either the execution of a specific function or the generation of a fact.

The facts exist as *assertions*, that link a working memory entity with a value. A fact such as "John is a man" can be expressed in a predicate format as $is\_a(john\ \ man)$ where *john* is the object, is_a is the attribute and *man* is the value. For this system, the entities in working memory only have a value. Therefore a specific naming convention is used to specify an attribute for the object which is appending the attribtute name to the object name as in

$$object - attribute$$

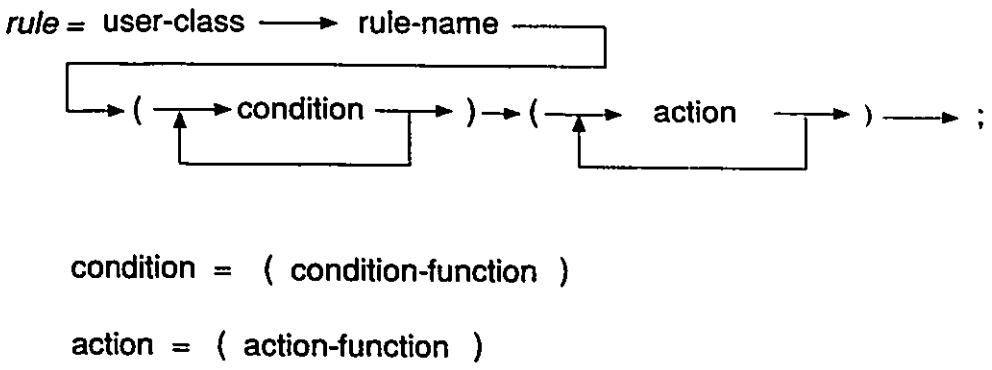
In the system the fact $is\_a(john\ \ man)$ is represented as the assertion $john - is\_a$ with the value *man*. The naming convention then allows an object to have a number of attributes associated with it. The naming convention must be followed throughout the definition of facts as well as the conditions that test facts. The conditions are designed to test assertions in the knowledge base. The test may include whether the assertion exists or the value of an assertion.

Each time the inference engine is invoked the assertions are cleared. This makes each invocation independent from the next or previous. There may be a requirement to have certain information maintain across inference invocations. This is accomplished through the creating of *permanent assertions*, which are never deleted. The permenent assertions are used by the user-database interface.

6.3 Rule Format

The **rules** have the following general syntax for their definition as seen in figure 6–3.

Figure 6–3: Rule Format



If more than one **condition** is specified in a rule, then all the condition must be true for the rule to "fire" and execute all the associcated actions.

The testing of the assertions is provided by several LISP condition functions as listed in table 6–1.

Table 6–1: Condition Functions

Function	Format	Description
String Equality	(STRING= a b)	Testing if string a is equal to string b
String Inequality	(NOT (STRING= a b))	Testing if string a is not equal to string b
Null Value	(IF-NULL assertion)	Test if the assertion is null
Existence of	(IF-EXIST assertion)	Test if the assertion is currently defined

Assertion values are always string values. A special function is exists that allows the current value of an assertion to be return, this being GET-ASSERTION. It can be used within some of the condition functions to specify one or both of the strings to test. Its use in conditions is illustrated in the rules in figure 6–4.

Figure 6–4: GET-ASSERTION Function

```
USERA  GET-DATA-A
       ((STRING= (GET-ASSERTION  'A) "FISH")
        (STRING= (GET-ASSERTION  'BOOK-RETURNED)
                 (GET-ASSERTION  'BOOK-STATUS))
       (  ... actions .... ) ;
```

The actions are a series of one or more LISP functions. This could include standard LISP functions or other system defined functions for creating or assigning values to assertions or other activities. The predefined LISP action functions are listed in table 6–2.

Table 6–2: Action functions

Function	Format	Description
Set Assertion	(SET-ASSERTION x y)	Set the assertion x to the value y
Access Database	(SCREEN ...)	Access the database through a screen
Enable User Rules	(ENABLE-USER-RULES)	Allow the inference to process user rules instead of system rules

6.4 Initial Conditions for Rule Firing

When control is transferred to the inference engine, the rules in the knowledge base are scanned. In order for a rule to "fire", some initial assertions must be set. A specific assertion could be used to determine which activities to perform. By assigning it different values, a

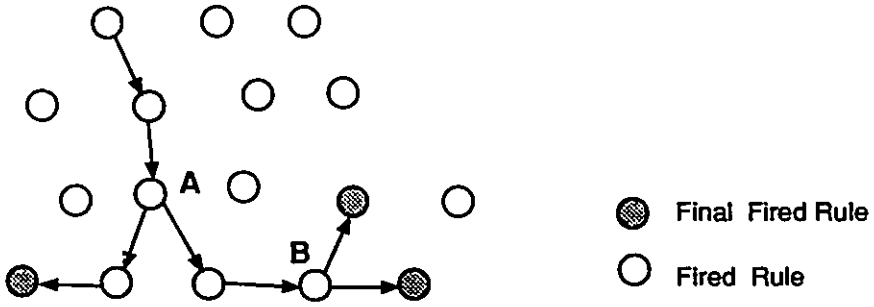
different sequence of rules will be "fired". Initial rules for each different task could test the value of this specific assertion as a trigger to execute the task.

6.5 Inference Engine

The inference engine is an important component of an expert system and also this system. Its basic function is to scan and identify rules to fire. Once the rules have been identified, then which rule or rules to fire must be determined. The domains of expert systems vary greatly, from medical to geology to mathematics. Each domain has unique knowledge and logic that is used in perform decision making. In the field of medicine for example, a common occurrence in knowledge is a degree of uncertainty. When tests results are returned, there may not be a single correct diagnosis based on the test. There may be several possible diagnoses, each with a different probability. This means that a probability or confidence factor (CF) must be associated with the rules and used by the inference engine in determining the0 rules to fire. This introduction of probability in rule firing has been termed *fuzzy logic* or *approximate logic* and is discussed in [Appelbaum and Ruspini 85], [Raman and Kerre 85], [Wierzchon 85], [Zadeh_85], [Zemankova-Leech 85] and [Summers Oct90]. This situation can cause several different rule streams to become active, each resulting in a different result as seen in figure Figure 6–5.

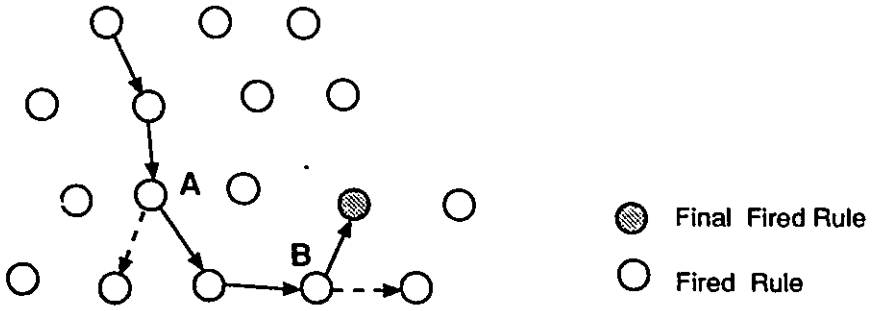
There are two points (A and B) at which multiple rules fire. One interesting result is that the order in which the multiple rules are fired, can have an effect on the final outcome. This means the outcome may be nondeterministic. This environment can contain *generic rules*, which may be involved in different rule streams for a different set of initial conditions, as opposed to *specific rules* which are involved in only one rule stream for only one set of initial conditions.

Figure 6–5: Multiple Rule Streams



In the User-Definable Office Information System, there is no specific facility for having a CF associated with rules. Although CFs are common in many expert systems, it is not an absolute requirement. The requirement of CFs is often based on the domain of knowledge for the expert system. In the User-Definable Office Information System, the rules have a specific single outcome, not several possible outcomes. This is due to the nature of the domain of knowledge, the office. The office environment does not have the same uncertainty as in the medical or geological environments. By removing CFs, also simplified the design and implementation of the inference engine. Generic rules can be created in the system but would not necessarily compose all the rules. There will only be one rule stream active in the knowledge base. If the inference engine locates multiple rules that can be fired, it must determine which one will be fired as seen in figure Figure 6–6.

Figure 6–6: Single Rule Stream



The approach of the system is to choose the rule to fire that has the greatest number of conditions. The greater the number of conditions, the greater the amount of knowledge the rule has referred to. This method of decision is often used in the decision making process. Additional conditions can be added to some initial condition(s) of a rule to form a new rule to handle specific exception situations. This allows the new specific rules to be chosen over generic rules. In figure Figure 6–6, at points A and B, only one rule is fired even though multiple rules are selected as being fireable. A situation may arise where several rules all have the same number of conditions and are all fireable. At this point, the inference engine will display this information to the user and terminate its searching. This is a deadlock situation which must be resolved to allow correct rule execution since a single rule stream is assumed for the office environment.

Another feature of the inference engine is, that once a rule has been identified to be possibly fired, it can not be fired a second time in the current rule stream. This avoids the possibility of a cycle occurring within rule firing.

The logic for the inference engine is listed in figure 6–7 in a pseudo-programming language.

Figure 6–7: Inference Engine Program

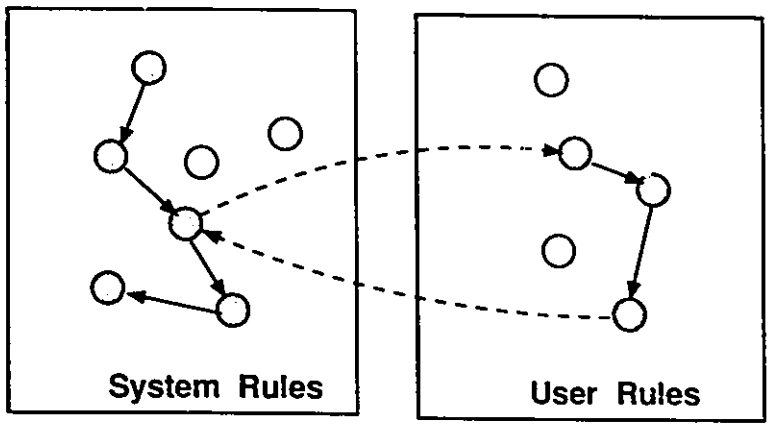
```
rule-stack <-- empty
exit-flag  <-- FALSE
WHILE (NOT exit-flag)
  rule-to-fire-stack <-- SCAN-RULES(knowledge-base, rule-context)
  IF (SIZE(rule-to-fire-stack) = 0) THEN
    exit-flag <-- TRUE
  ELSE
    IF (SIZE(rule-stack) = 1) THEN
      rule <-- TOP(rule-to-fire-stack)
      MARK-RULE(rule, IN-USE, ACTIVE)
      rule-stack <-- PUSH(rule, rule-stack)
      FIRE-RULE(rule)
    ELSE
      rule <-- SELECT-RULE(rule-to-fire-stack)
      MARK-RULE(rule, IN-USE, ACTIVE)
      rule-stack <-- PUSH(rule, rule-stack)
      FIRE-RULE(rule)
    END-IF
  END-IF
END-WHILE
```

6.6 SYSTEM and USER Rules

Two types of rules exist in the system, SYSTEM and USER. This allows users to perform some tailoring of functionality. The SYSTEM rules form the basis of the selected task functionality. Users can add additional features through their own USER rules. There are restrictions that must be placed on USER rules. For example, if through a series of SYSTEM rules, a record is added to a relation, the USER rules must not be allowed to delete this record. Also, USER rules should not be allowed to potentially fire at any time in relation to the SYSTEM rules. The current implementation uses a special rule-action function called ENABLE-USER-RULES to control USER rules. This function will allow USER rules to be searched and fired instead of SYSTEM rules. When the function is called as part of some SYSTEM rule action, it changes the current rule context from SYSTEM to USER and invokes

the inference engine. The inference engine will now search exclusively through the USER rules. This is illustrated in figure Figure 6–8.

Figure 6–8: System/User rule interaction



Once all the USER rules have fired, then current rule context is restored to SYSTEM and control returns to the next action after the call to the ENABLE-USER-RULES function. This feature can be used to allow user rule execution to be enabled only for specific tasks and at specific points in the task.

6.7 SYSTEM and USER Assertions

There are two types of assertion, SYSTEM and USER. This is seen in figure 6–1. The SET-ASSERTION action function creates all assertions and can assign values to an assertion. The type of assertion created depends on the current rule context. The current rule context is changed through calling the ENABLE-USER-RULES action function. When the current rule context is SYSTEM, only SYSTEM assertions are created or assigned values. When the current rule context is USER, only USER assertions are created or assigned values. USER

assertions are required so that USER rules do not modify SYSTEM assertions, which could then alter the SYSTEM rule execution. The purpose of USER rules is to supplement, but not override SYSTEM rules. Assertion names must be unique across both SYSTEM and USER assertions. This means USER assertions can not be created to override the corresponding SYSTEM assertion when reference by USER rules. The GET-ASSERTION function is used to access an assertion. When the current rule context is SYSTEM, only SYSTEM assertions are accessed. When the current rule context is USER, then both SYSTEM and USER assertions are accessed. This allows USER rules to test SYSTEM assertion.

CHAPTER 7
DATABASE SYSTEM INTERFACE

The storage of data for an application system is an important design decision. For most large applications, a database system is usually chosen. In the User-Definable Office Information System, a relational database system is used as the data storage platform. The specific product is VAX Rdb/VMS. This is a relational database product from Digital Equipment Corporation, that runs on a VAX computer.

A relational database is composed of many relations that can be accessed individually or joined together for providing information in queries. often used in the system. The ability to join different relations is often required by users who desire to see some information, but is not solely contained in a single relation.

7.1 VAX Rdb/VMS Interface

The VAX Rdb/VMS product has its own query language and interface known as RDO. Version 3.0 of VAX Rdb/VMS also supports an SQL interface as well. For using VAX Rdb/VMS from a program, there are several possible methods. One method involves using embedded query language statements inside a high level language program (ie COBOL, FORTRAN, PASCAL). These programs are precompiled and then compiled and linked. The precompiling replaces the embedded statements with the appropriate subroutine calls. The compiling of this new program then produces a executable program. This method generates relatively efficient code for execution. A second method involves using an interpretive interface. In this

method, the program makes calls to an RDO statement interpreter, which accepts RDO query language statements, and then executes them. This method provides a dynamic environment, since the statement evaluation and execution occurs at runtime. It does however produce less efficient execution of the RDO statements, since they must be interpreted each time. It was the second method, the interpretive interface, that was used in the Office Information System. This was directly a result of the fact that the first method would not work with LISP source code.

RDO is a query and data definition language. Its full complement of statements can be found in [Rdb/VMS Reference 88]. A summary of statements used by the system is in table 7–1.

Table 7–1: Utilised RDO statements

Statements	Function
INVOKE DATABASE	Open and allow access to the database
FINISH	Closes and terminates access to the database
START_TRANSACTION	Starts a new transaction
COMMIT	Commits the current transaction
ROLLBACK	Rolls back the current transaction
START_STREAM	Creates a stream of records from the database from a Record Selection Expression
FETCH	Retrieves the next record from a stream
MODIFY	Updates an existing record
ERASE	Deletes an existing record
STORE	Adds a new record

To access the database through RDO statements requires calling of the RDB$INTERPRET routine [Rdb/VMS Programming 88]. This routine resides in a shareable library once the Rdb/VMS database system has been installed on a VAX computer. This routine accepts a variable number of arguments. The first argument is the RDO statement to be executed, and the subsequent arguments represent the necessary data to be passed to the RDO or received from the RDO statement. The calling format in LISP is illustrated in figure 7–1.

Figure 7–1: RDB$INTERPRET routine

```
(RDB$INTERPRET  rdo-statement   host-language-variable
                                host-language-variable ... )
```

The host language variables, LISP variables, can not be included directly in the RDO statement. Instead a placeholder parameter !VAL is used to reserve a place for each of the specified host language variables. When RDB$INTERPRET interprets the RDO statement it will replace each placeholder with the value of the next host language variable from the original call. This means the number of placeholders must match the number of host language variables. These rules apply specifically to GET and MODIFY data manipulation statements.

To perform a query on the database through the RDB$INTERPRET interface, the following steps are necessary:

1. Start a transaction
2. Form a stream by specifying the Record Selection Expression
3. Loop Fetching each record from the steam

A sample series of RDO statements is seen in example 7–1.

In this example, RDB$INTERPRET would be called four times once for each statement. This would allow the first record from the relation to be retrieved. In the call to RDB$INTERPRET

Example 7–1: RDO statements to read

```
START_TRANSACTION READ_WRITE RESERVING EMPLOYEES FOR SHARED READ

START_STREAM  s  USING  e  IN  employees
  WITH  e.department = !val  SORTED BY e.last_name

FETCH s

GET  !val = e.first_name;
     !val = e.last_name;
     !val = e.department
END_GET;
```

for the START_STREAM statement, there would have to one (1) host language variable supplied that would provide the value for the relational operation. In the call to RDB$INTERPRET for the GET statement, there would have to be three (3) host language variables supplied to receive the data values from the relation's fields. The host language variables can be used for both input and output from RDB$INTERPRET.

As an optimisation to the RDB$INTERPRET interface, the system has eliminated the requirement for using host language variables on the START_STREAM statement used to form the Record Selection Expression (RSE). This has been accomplished through the dynamic building of the RSE by inserting literals for those values come from host language variables.

7.2 Basic Database Operations in callable RDO

The following examples list the sequence of callable RDO statements for use with RDB$INTERPRET for the four (4) basic database operations.

7.2.1 Reading Data

In the READ operation, all the records in the specified stream can be retrieved by looping with the FETCH and GET statements. The loop is terminated with an End-of-file condition.

Example 7–2: Database Read operation

```
                START_TRANSACTION  .....
                START_STREAM  x ... rse
        +--->   FETCH x
Loop    |
        +-----  GET .... END_GET;
```

Example 7–3: Database Update operation

```
                START_TRANSACTION  .....
                START_STREAM  x ... rse
        +--->   FETCH x
        |
Loop    |       GET .... END_GET;
        |
        +---    MODIFY ...  END_MODIFY;
```

7.2.2 Updating Data

In the UPDATE operation, the record to be updated can be optionally retrieved and displayed before it is updated. The loop is terminated with an End-of-file condition.

7.2.3 Deleting Data

In the DELETE operation, the record to be deleted can be optionally retrieved with the GET statement to have its contents displayed before it is deleted. The loop is terminated with an End-of-file condition.

Example 7–4: Database Delete operation

```
                START_TRANSACTION  .....
                START_STREAM  x ... rse
        +--->   FETCH x
        |
Loop    |       GET .... END_GET;
        |
        +---    DELETE
```

Example 7–5: Database Store operation

```
                START_TRANSACTION  .....
                STORE ... END_STORE;
```

7.2.4 Storing Data

In the STORE operation, any number can be performed within the current transaction.

7.3 Database Normalisation

The logical design of a relational database requires the data be normalised. Further information on database normalisation is available in [Date 87], [Cardenas 79]. This particular function generally can not be performed by the users, and must be performed by the database designer.

7.4 Query Optimisation

The Rdb/VMS database system has an advanced query optimiser that is invoked for all queries through RDO and SQL interfaces [Rdb/VMS Maint 88]. Its use is to determine the optimal data access to resolve the specified query. The optimal solution is based on the fewest number of I/Os to the database for the query. The system takes advantage of the optimiser, and as such performs no independent optimisation on the user's query. The functions and benefits are discussed in [Date 87].

7.5 Concurrency and Consistency in the Database

The issues of *concurrency* and *consistency* are very important in database systems. The features of the database with respect to these two issues vary from product to product.

7.5.1 Concurrency

Concurrency deals with the ability of the database system to allow multiple concurrent users access to the database. The *concurrency control mechanism* is required to ensure that the concurrent database transactions do not interfere with each other [Date 87] The interference between different transactions is essentially limited to three things that can go wrong [Date 87]. These are

- The Lost Update
- The Uncommitted Dependency
- The Inconsistent Analysis

If database systems can not provide a high level of *concurrency* then the number of simultaneous users will be few and the use of the system in the office environment will be greatly reduced. The solution to *concurrency* is a *concurrency control mechanism*. Several possible methods exist. The most commonly used method is known as *locking*. Rdb/VMS uses a locking system for *concurrency*. The Rdb/VMS locking system uses the VMS Distributed Lock Manager [Snaman and Thiel Sep87].

The Rdb/VMS database system as a facility to increase *concurrency* called *Snapshoting*. This method is used by users who start a READ ONLY transaction. In this transaction the user only reads data and can not update it. The *snapshoting* involves the writing of *before-images* of the updated records to a special SNAPSHOT file. The READ ONLY transactions can access the snapshot file and not wait for locks held on the updated record. The system attempts to take advantage of this feature through the user-database interface.

7.5.2 Consistency

Consistency deals with the integrity of the data in the database. The term *Referential Integrity* is also used. related to *concurrency* [Date 87]. Integrity in relational databases is of the utmost importance since, unlike other database models, the relation model requires that certain data is duplicated in the database. This is the key/foreign key required to join relations together. The basic concepts of *referential integrity* can be summarised in the following statements:

a. If two relations A and B are related through a key/foreign key, then if a record in A exists with a foreign key value of x, then there exists at least one record in B with the key of x.

b. If two relations A and B are related through a key/foreign key, then if a record in A with a foreign key value of x is deleted then all records in relation B with the key value of x are delete. This is some times called *cascading deletes*.

Rdb/VMS Version 3.0 currently supports the first item (a) of referential integrity, and currently has no support for (b) †

7.6 Predefined Functions

Several LISP functions for use with the database have been written.

Table 7–2: LISP Database functions

Functions	Description
RDB-DATABASE-OPEN	Opens a database for access
RDB-DATABASE-CLOSE	Closes a database, after committing any uncommitted transactions

Based on the previously defined LISP functions, some common methods have been defined for each user class to support database access. These methods are

† Rdb/VMS Version 3.1 currently supports both items (a) and (b) for referential integrity.

Method	Description
DATABASE-OPEN	Opens a database
DATABASE-CLOSE	Closes a database

7.7 Physical Database Design

The physical design of the database must be performed by a knowledgeable database expert. It is essential to have a optimally defined database so that possible performance problems may be eliminated. The author does not intend that this function be performed by users.

In the physical design of the database, specifically with Rdb/VMS, several items must be addressed. These include

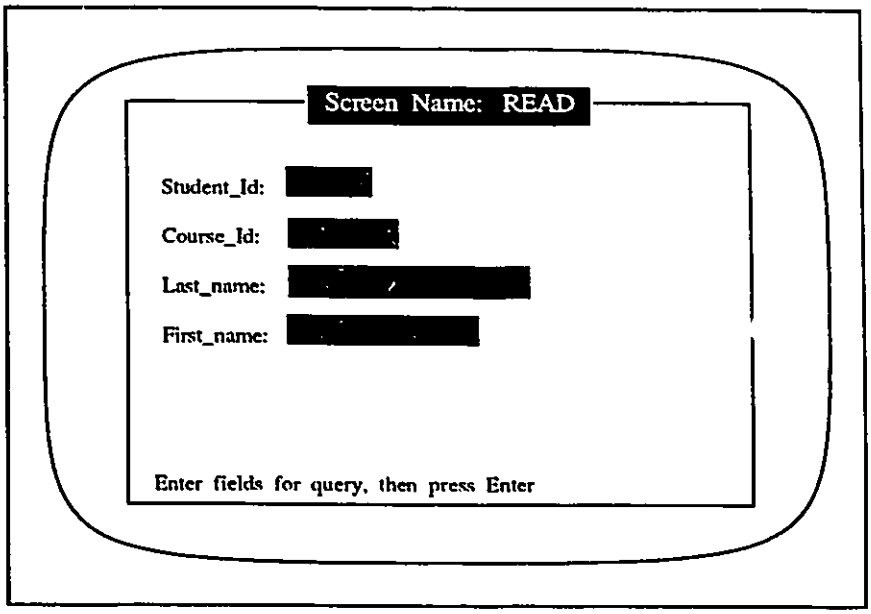
- Single file versus Multi-file Database
- Number of Indexes
- Type of Index (Sorted or Hash)
- Compressed records or non-compressed records

CHAPTER 8
USER-DATABASE INTERFACE SYSTEM

An important part of a system is the user interface provided for data access, specifically to a database. Since users will frequently perform queries and data entry some interface must be used that is simple and visually informative. For the system, a user-database interface is provided that attempts to remove much of the detailed technical requirements from the user in performing data manipulation on a database.

In the system, the user creates and uses **screens** that access the database. A **screen** is a "window" on one or more relations in the database. Each screen created has a specific functionality with regards to the database activity it can perform. The database functionality includes the ability to read, add, delete and update the database. The **screens** allow data to be displayed in a structured and consistent fashion for the user. This concept of a screen mapping to data is common among 4GLs including PowerHouse, SQL*Forms and VAX Rally. The screen provides the user a "fill in the blanks" process for performing database functions, especially queries. This method is an alternative to a structured query language for database access such as SQL, VAX DATATRIEVE or VAGUE [Motro Apr88], or a natural language query language such as INTELLECT. A screen as displayed on a terminal is illustrated in figure 8–1.

Figure 8–1: Sample Screen



A feature of this interface is that the user can specify multiple relations to be included on the same screen. This in itself is not new, but that is important is that the user does not need to know what the actual linking is between the relations in question and in SQL. The system will determine a linking and automatically join all the required relations for the requested relations and fields. A condition placed on the relations to be specified in a screen is that they form a single branched tree when joined. The linking for the join is based on knowledge in the system provided by the database designer. The linking is the shortest path from one relation to the next through the fewest number of linking relations.

8.1 SCREEN Function

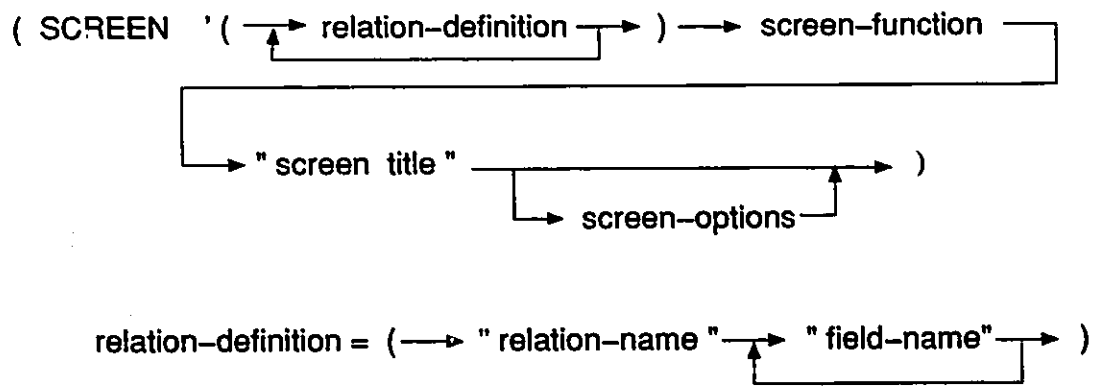
A LISP function called SCREEN exists that can be invoked from a rule or from a method associated with a particular user. A database DB consists of relations $R_a, \ldots, R_b$. Each relation R_x has fields $f_{xa}, \ldots, f_{xb}$ The screen function may be formally defined for a simple single relation query as

$$SCREEN\,(f_x, \ldots, f_{xb})$$

which is read as a database query displaying fields f_{xa} to f_{xb} from relation R_x.

The format of the SCREEN function shown in figure 8–2.

Figure 8–2: SCREEN function syntax



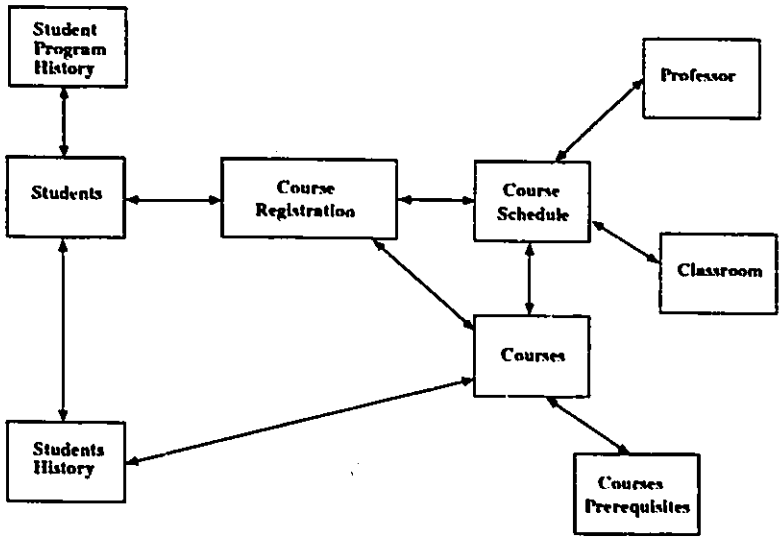
8.2 SCREEN Function Parameters

Certain parameters to the SCREEN function are required and others are optional.

8.2.1 Relations and Fields

The relations and fields to be displayed on in the screen must be listed. The first relation listed is the **primary relation**. All other relations that are listed must form a hierarchy or tree structure. The end-user does not need to know any intervening relations that are required to link the specified ones together. Fields from each relation must be listed individually. The field name displayed on the screen is the field name from the database. Determining the hierarchy is easily accomplished from the database relations diagram. What is happening is that the relations are being JOINed for the accessing.

Figure 8–3: Example Database Relations



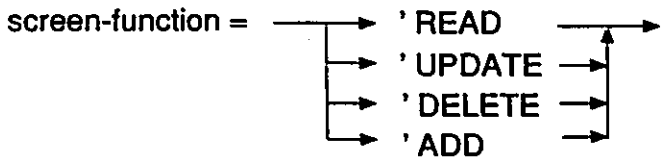
With reference to figure 8–3, if fields from the relations STUDENTS and COURSES were to be displayed. If STUDENTS were the primary relation, then the hierarchy is simply STUDENTS –> COURSES. The SCREEN function will perform the actual linking through the relations STUDENT_HISTORY since it links STUDENTS to COURSES.

If there were fields from STUDENTS, STUDENT_HISTORY and COURSES to be displayed, the relations MUST be listed in this sequence. If it is not done, then the SCREEN function will not successfully link all the relations.

8.2.2 SCREEN Functions

Four functions exist, as seen in figure 8–4. Only one function can be specified per screen, meaning that each screen performs a single basic function.

Figure 8–4: SCREEN functions



8.2.2.1 Read Function

When the Read function is used, the screen is displayed with all the various fields from the specified relations. Each field input and output area is highlighted, and data can be entered into any or all of the fields.

Once the input is complete, the database is then searched. The searching is based upon the input from the user. Every field with input is then used as part of the selecting expression in accessing the database. The performance of each query may vary based on whether any indexes exist. This optimisation of the query is left entirely to the database system.

The Rdb/VMS RDO query interface has a well developed optimizer that does the necessary optimisation of the query. The SCREEN function creates the appropriate RDO command to perform the query. If the query returns one or more data "records", each will be displayed in sequence.

The read function of the SCREEN function for a single relation can be formally described as the following. The set of input from the user for the SCREEN

$$I = \{i_{xa}, \ldots, i_{xb}\}$$

where i_{xa} is the input for field f_x and so on. The input set I may be described as the union of two disjoint sets of input I_E for the fields on the screen with no user input, and I_D for the fields with user input

$$I_E = \{\forall i_{xi} : i_{xi} = \phi\} = \{i_{xc}, \ldots, i_{xd}\}$$

$$I_D = \{\forall i_{xi} : i_{xi} \neq \phi\} = \{i_{xe}, \ldots, i_{xf}\}$$

$$I = I_E \cup I_D$$

A single relation query where $I_D \neq \phi$ may be described as

$$SCREEN\,(f_{xa}, \ldots, f_{xb}) \; : \; \exists R_x \left((f_{xe} = i_{xe}) \wedge \ldots \wedge (f_{xf} = i_{xf})\right)$$

which is read as, for this database query involving fields f_{xa} to f_{xb}, does there exists at least one record in the relation R_x such that field f_{xe} is equal to user input i_{xe} and ... and field f_{xf} is equal to user input i_{xf}. If $I_D = \phi$ then all records in the relation are accessed.

The SCREEN function also supports multi-relation queries. The first type of simple multi-relation query is where there is a direct key/foreign key relationship between the two relations where f_{xt} is the foreign key in relation R_x and f_{yu} is the key field in relation R_y. There are

no intervening relations that are required for the joining. The input I for this multi-relation query can now be defined as

$$I = \{i_{xa}, \ldots, i_{xb}, i_{yg}, \ldots, i_{yh}\}$$

where $i_{xa}, \ldots, i_{xb}$ is the input for fields $f_{xa}, \ldots, f_{xb}$ in relation R_x and $f_{ya}, \ldots, f_{yb}$ in relation R_y. The input set I may be described as the union of two disjoint sets of input now over two relations

$$I_E = \{\forall\, i_{xi} : i_{xi} = \phi\} \cup \{\forall\, i_{yj} : i_{yj} = \phi\} = \{i_{xc}, \ldots, i_{xd}, i_{ym}, \ldots, i_{ym}\}$$

$$I_D = \{\forall\, i_{xi} : i_{xi} \neq \phi\} \cup \{\forall\, i_{yj} : i_{yj} \neq \phi\} = \{i_{xe}, \ldots, i_{xr}, i_{yp}, \ldots, i_{yq}\}$$

$$I = I_E \cup I_D$$

This simple multi-relation query where $I_D \neq \phi$ may be described as

$$SCREEN\,(f_{xa}, \ldots, f_{xb}, f_{yg}, \ldots, _{yh}) :$$

$$\exists R_x\,((f_{xm} = i_{xm}) \wedge \ldots \wedge (f_{xn} = i_{xn}))$$

$$\wedge\, \exists R_y\,((f_{yp} = i_{yp}) \wedge \ldots \wedge (f_{yq} = i_{yq}))$$

$$\wedge\, \exists R_x \exists R_y\,(f_{xt} = f_{yu})$$

Should $I_D = \phi$ then the query may be described as

$$SCREEN\,(f_{xa}, \ldots, f_{xb}, f_{yg}, \ldots, f_{yh}) =$$

$$\exists R_x \exists R_y\,(f_{xt} = f_{yu})$$

which is simple equi-join of the two relations over the foreign key link between the relations.

The SCREEN function also supports complex multi-relation queries. These involve two or more relations that do not have a direct key/foreign key relationship. This means that one or more intervening relations are required to JOIN the required relations. For a query, relations R_x and R_y are involved and have no direct key/foreign key relationship. Therefore there is a linking between R_x and R_y consists of other relations LR_i, results in the following sequence of relations

$$R_x, LR_r, LR_s, \ldots, LR_t, LR_u, R_y$$

For each pair of relations, there exists a key/foreign key relationship that is used to JOIN the relations together

The complex multi-relation query where $I_D \neq \phi$ may be described as

$$SCREEN\,(f_{xa}, \ldots, f_{xb}, f_{yg}, \ldots, f_{yh}) \;:$$

$$\exists R_x\,((f_{xm} = i_{xm}) \wedge \ldots \wedge (f_{xn} = i_{xn})) \wedge$$

$$\exists R_y\,((f_{yp} = i_{yp}) \wedge \ldots \wedge (f_{yq} = i_{yq})) \wedge$$

$$\exists LR_r \exists R_x\,(f_{xa} = f_{rb}) \wedge$$

$$\exists LR_s \exists LR_r\,(f_{rc} = f_{sd}) \wedge$$

$$\vdots$$

$$\exists LR_u \exists LR_t\,(f_{te} = f_{uf}) \wedge$$

$$\exists R_y \exists LR_u\,(f_{ug} = f_{yh})$$

Should $I_D = \phi$ then the query may be described as

$$SCREEN\,(f_{xa}, \ldots, f_{xb}, f_{yg}, \ldots, f_{yh}) \;:$$

$$\exists R_x \exists LR_r\,(f_{xa} = f_{rb}) \wedge$$

$$\exists LR_r \exists LR_s (f_{rc} = f_{sd}) \wedge$$
$$\vdots$$
$$\exists LR_u \exists LR_t (f_{te} = f_{uf}) \wedge$$
$$\exists R_v \exists LR_u (f_{ug} = f_{vh})$$

which is an Equi-join over the relations in question.

Note that for all of the different types of queries, additional conditions may be added based on a protection scheme called *Query Modification*. This is discussed in greater detail in Chapter 9.

8.2.2.2 Update Function

The Update function starts with the Read function to accept user input for the database query and then reads each "record". After each record is displayed the user can enter new information into any or all of the fields. If there are multiple "records" retrieved by the query, the user has the opportunity to update each one.

8.2.2.3 Delete Function

The Delete function starts with the Read function to accept user input for the database query and then reads each "record". After each record is displayed the user can delete each record by responding to the delete prompt with a Y. If there are multiple "records" retrieved by the query, the user has the opportunity to delete each one.

8.2.2.4 Add Function

The Add function allows the user to enter data into each field on the screen. The record is added once all the fields have data entered in them, or the user presses the Enter key after filling only selected fields.

For this function, only single relations can be used. No joins of relations or complex database views are supported for adding data. This is a limitation of the database system Rdb/VMS.

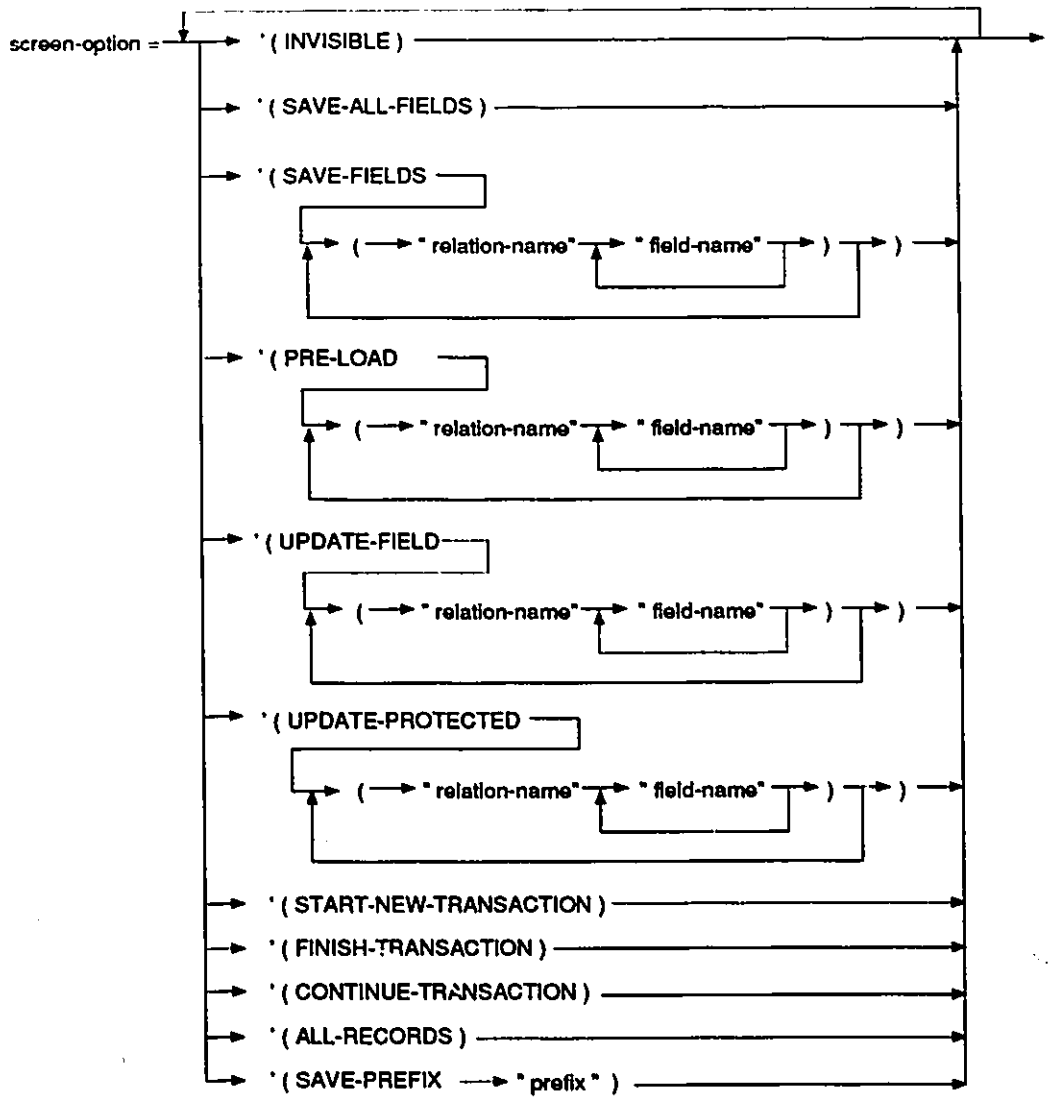
8.3 Screen Options

There are several screen options that can be specified for each SCREEN function call. These functions provide additional capabilities to the basic data manipulation functionality of the screen. The options are listed in figure 8–5.

8.3.1 Transactions

Database systems allow work to be performed in units called *transactions*. Any amount of work, data reads, updates, deletes and additions can be performed. A *transaction* is terminated with one of two possible results. Either the transaction is *committed*, meaning that all changes to the database are permanently made, or the transaction is *rolled back*, meaning any changes are undone. The individuals involved in the definitions of the rules or methods involving the use of the SCREEN function must be aware of transactions and their proper use. The most basic rule with respect to transactions is to keep them as short as possible. This means that each transaction should perform a single user defined activity. A single transaction should not be used to perform all the work that a single user should perform. The SCREEN function provides the flexibility in use of transactions.

Figure 8–5: SCREEN Function Options



The START-NEW-TRANSACTION option causes the screen to start a new transaction for the screen's database activity to be executed in. The previous transaction, if any, is first committed. If this option is not specified, then the first SCREEN function call will create the first transaction.

The FINISH-TRANSACTION option causes the current transaction to be committed once the screen's database activity has completed. This option used with the START-NEW-TRANSACTION option ensures that the screen's database activity is contained within a single transaction.

The CONTINUE-TRANSACTION option causes the current transaction to remain after the screen's database activity has completed. This is useful for having a transaction span several screen functions.

The SCREEN function attempts to reduce the locking conflicts that occur in Rdb/VMS by using READ ONLY transactions where possible. If a screen is invoked that is a READ screen and specifies both START-NEW-TRANSACTION and FINISH-TRANSATION options, it uses a READ ONLY transaction. If these conditions are not met, then a default READ WRITE transaction is started since the system does not know beforehand what other screens are going to be used.

8.3.2 Preloading Fields

Often when a screen is displayed, some field or fields may already have data in them. The data in the field is not modifiable, meaning that the user can not change any of the preloaded fields.

8.3.3 Saving fields

A critical part of the rule processing, is that **assertions** can have values changed and/or be created. Once the user has exited from a screen, then data from that screen, or more correctly, data from the database can be automatically set as an assertion. The creation of this assertion may then cause additional rules to fire. All the fields on a screen can be saved

as assertions by specifying the SAVE-ALL-FIELDS option. Individual fields can be saved by specifying the SAVE-FIELDS ... option and listing the fields and relations. The assertion name that is created to contain the last displayed screen value is

```
relation-field    (ie   STUDENT_HISTORY-STUDENT_ID )
```

8.3.4 Invisible Screens

By default all screens are visible. This means that the screen can be seen by the user and data operations performed. Much of the activities that are involved within an application may be visible. There are however some activities that must be performed transparent to the user. These are activities that are integral to the function being performed. These activities can be performed by creating "invisible screens". The invisible screen performs the same processing as visible screens without any user input. They can be used to delete records, update records, add new records, or perform a lookup of some data. By saving fields this "invisible" activity can be used to trigger more rules.

8.4 Data required by the SCREEN function

In order to perform properly, a data structure must be defined that describes the database for the **screen** function. This data is contained in the LISP variable MASTER-RELATIONS-FIELDS-LIST. It identifies all the relations and corresponding fields in the database, as well as the linking between each relation. Its format is seen in figure 8–6.

Each relation in the database that the **screen** function is to be used with must be listed.

Figure 8–6: MASTER-RELATIONS-FIELDS-LIST data structure format

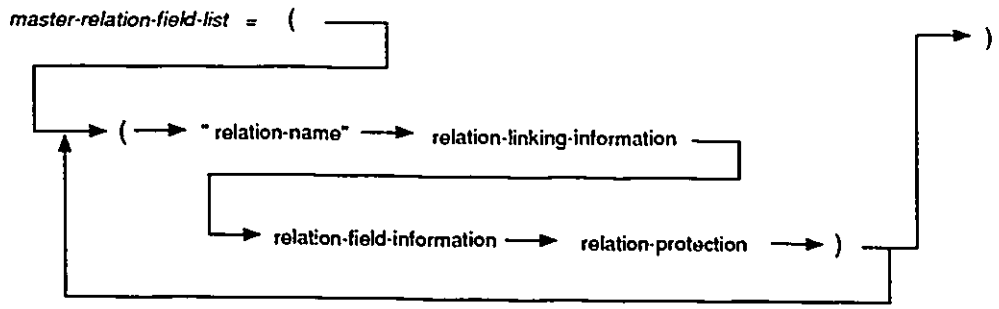
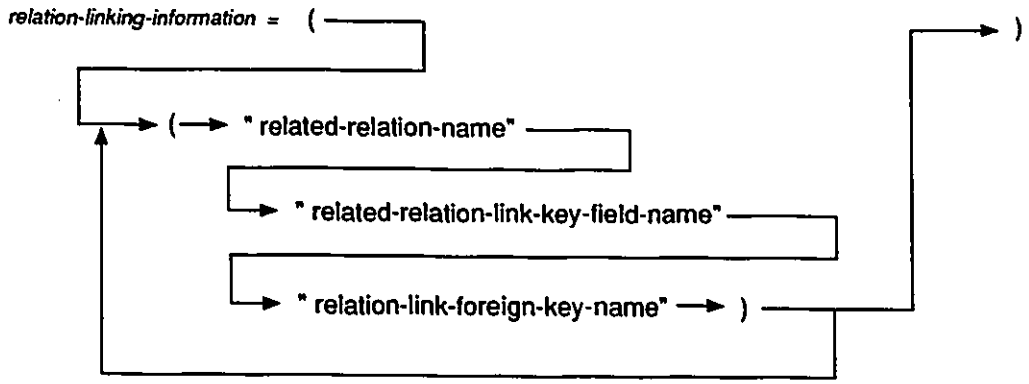


Figure 8–7: MASTER-RELATIONS-FIELDS-LIST relation-linking-information structure format

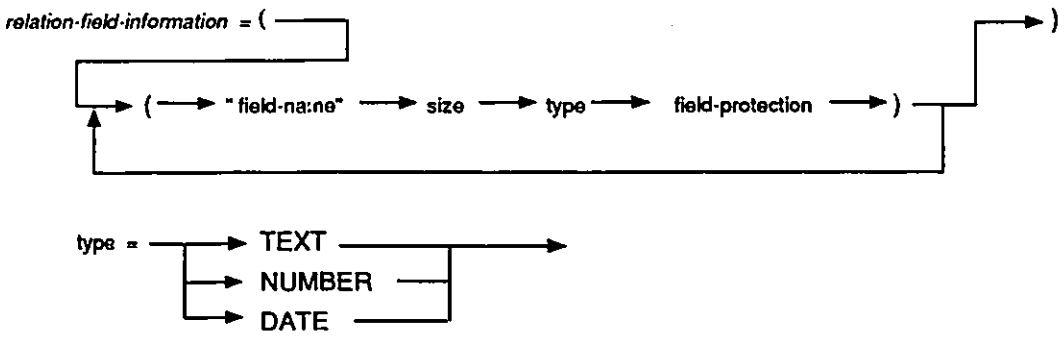


The *relation-linking-information* is a list of the various other relations that this one can be related to through a key/foreign key. Currently the key/foreign key pair can only be single fields from each relation.

Some control can be exercised over which relations are used in joining one relation to another. It is possible to omit a linking from this structure even though the linking exists. This would then result in another linking being used.

The *relation-field-information* is a list of the various fields that are in a relation is seen in figure 8–8. The field names must match the field names as defined in the Rdb/VMS database. The size is in bytes. The type is not currently used, but can be used for data interpretation.

Figure 8–8: MASTER-RELATIONS-FIELDS-LIST relation-field-information structure format



The *relation-protection* and *field-protection* are discussed in Chapter 9. For reference, the definitions for the sample system are in Appendix F.

8.5 Permanent Assertions

Table 8–1 contains a list of the special assertions and their function. These assertions are special in that they are permanent and do not disappear from the system. They are used by the SCREEN function in managing transactions.

Table 8–1: Special Assertions

Name	Function
TRANSACTION_ACTIVE	Used to indicate where a transaction is currentiy active. It is generally referenced by the SCREEN function, but could be used by any rule to typically perform some form of rollback activity. Valid values are Y or N.
OPERATION-ABORTED	Used to indicate that the current operation has been aborted. The value is set to N when the SCREEN function is called, and is set to Y if the user aborts the operation in that screen. This can be used to bypass any subsequent rules related to a specific function.

8.6 Special Function Keyboard Keys

Four keys perform special functions within a screen. These are the

- *Return key*
- *Enter key*
- *PF4 key*
- *F20 key*.

8.6.1 Read Screen

The *Return key* is used to complete input to the current field and move to the next field if it exists. If there are no more fields then the search is performed. The *Enter key* is used to complete input to the current field and immediately perform the database search. The *F20 key* is used to exit the screen or repeat the input sequence. If the current field is the first field of the screen, then the screen is exited without performing the desired function. If the current field is not the first field, then the user is returned to the first field. Any previous input remains on the screen, but is not used as input to the field. The *F20 key* will exit the screen and allow processing to continue. The *PF4 key* is used to abort the current screen and rollback the current transaction.

8.6.2 Update Screen

The first part of an update screen is a read screen to specify the records to update. As each record to update is displayed, the user is allowed to update the fields.

To skip a field in updating, use the *Return key* with no input for that field. To update a field, the user enters data into the field and presses the *Return key* to proceed to the next field. Once the updated fields have been completed, pressing the *Enter key* causes the update to complete, and the next record if any is displayed. The *F20 key* will exit the screen and allow processing to continue. The *PF4 key* is used to abort the current screen and rollback the current transaction.

8.6.3 Delete Function

The first part of an update screen is a read screen to specify the records to update. As each record to update is displayed, the user is allowed to update the fields.

To skip deleting the displayed record respond to the delete prompt with a N. The *F20 key* will exit the screen and allow processing to continue. The *PF4 key* is used to abort the current screen and rollback the current transaction.

CHAPTER 9
PROTECTION AND SECURITY

In any application system, and specifically in information systems, protection and security are important concerns. An information system potentially is a large corporate database(s) which all users access to perform their work tasks. The integration of the many different parts of an organisation is a great task, which is beyond the scope of this thesis. One major concern in any information system, is protection and security. It can be summed up simply in two statements:

1. Who can see information.
2. Who can update information.

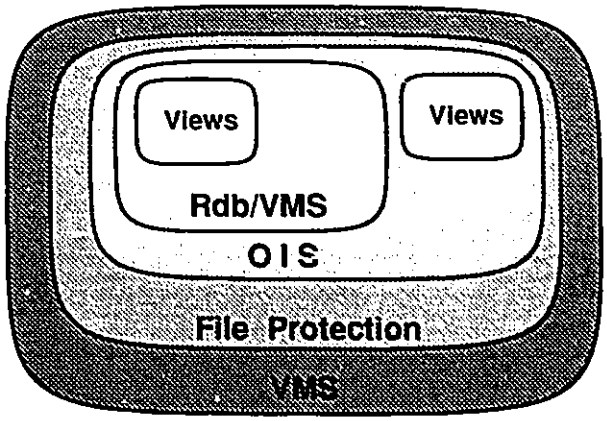
Which users access what information must be controlled, otherwise sensitive information could become known to the wrong person. Examples could include personal information, price information, processes or procedures and other proprietary information. Several articles in this area include [Dobkin 78], [Denning 78], [DeMillo et al. 78], [Chamberlin et al. 78], [Hsiao et al 79] and [Reiss 78].

Protection and security for an application generally consists of integrating the various protection and security features of the components that make up the system. This can include the hardware, the operating system, the database, the user account. Unfortunately the integration of many different and possibly diverse methods and procedures often causes problems and can leave openings that individuals can use to compromise protection and security.

9.1 System Protection and Security

The User-Definable Office Information System has several different components, each with its own form of protection and security. These are integrated to form a protection and security environment as seen in figure Figure 9–1. The following sections outline each of the components.

Figure 9–1: Protection/Security Environment



9.1.1 VMS Account

In order to gain access to the system, users must first log on a VAX/VMS computer system. In order to log on, the user requires a *VMS account* (commonly referred to a Username) and the corresponding *password*. With the correct Username and password, users can potentially run the system. Each user of the system could be assigned a different Username; thus each is independent from any other. Each could then have unique customisations that no other users have. Many users could also share the same Username, they could all be simultaneously

logged on, using the Office Information System. This situation would allow the definition of classes of users who perform exactly the same functions.

9.1.2 VMS File Protection

There are a number of files that are used by the system. These include among others, the files to define the menus and the rules (both system and user). Those files that are common to all users are placed in a shared directory that all users can read, but can not change. Other files specifically used for individual customisation would be placed in the user's own directory. This directory allows users to read and update their own files but not necessarily access any other user's directory and files.

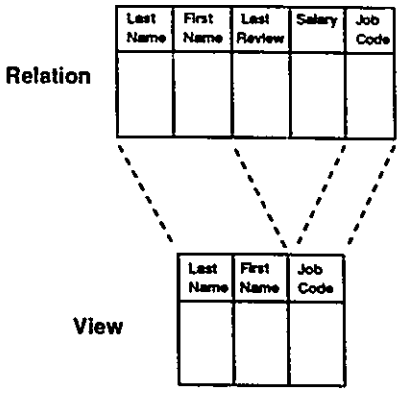
9.1.3 Database Protection

The Rdb/VMS database system as a protection scheme based on *Access Control Lists* (ACLs). These specify what users (in terms of VMS identifiers) are allowed to access relations in a database. The ACLs also determine what type of database operations the user is allowed to perform as well through *access rights*.

9.2 Views

A *view* can be defined within a database to create a subset of a relation's fields or to create superset of several relation's fields. The view, when a subset of a relation's fields, is accessed like a relation, but not all fields can be accessed. This method can be used for sensitive data, so that is inaccessible to some users who refer to the view, and yet accessible to other users who refer to the relation. A view is illustrated in figure 9–2.

Figure 9–2: A view



9.3 OIS Protection

The system has its own protection system. It consists of three items from [Hsiao et al 79]

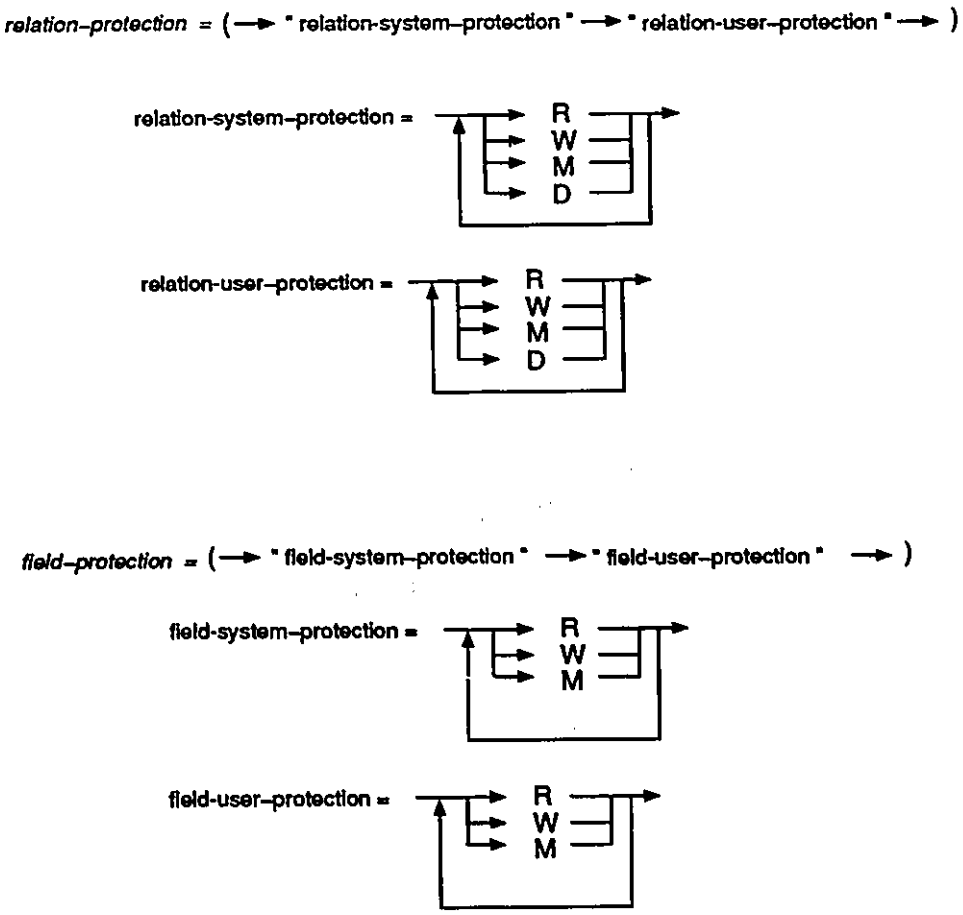
1. View Mechanisms
2. Query Modification
3. Context-Dependent Protection

9.3.1 View Mechanisms

A view of a relation can be defined, which is a subset of all the fields in the relation. This allows sensitive or restricted data to be seen only by required users. Each user has there own version of the MASTER-RELATIONS-FIELD-LIST data structure, which means that each user can have a different view of the entire database. The MASTER-RELATIONS-FIELDS-LIST data structure specifies the relations and fields that they have access to. By removing fields and/or relations for a particular user, a specific view can be created.

The operations that users can perform on the relations and fields also be specified. This represents the *relation-protection* and *field-protection*. This are illustrated in figure 9–3.

Figure 9–3: Relation and Field Protection



The *relation-protection* specifies the access to a relation in the database. The *field-protection* specifies the access to a field in the relation. A field can not be accessed if the relation is not accessible for the same type of access.

The values for relation and field protection are interpreted as in table 9–1.

Table 9–1: Relation and Field Protection Values

Value	Access	For Relation	For Field
R	Read	Read access to the Relation	Read access to the Field
W	Write	Ability to store new records in the relation	Ability to use this field for storing new data
M	Modify	Ability to read and then modify a relation	Ability to use this field to read and modify data
D	Delete	Ability to read and then delete records from this relation	Not applicable

For the access MODIFY and DELETE both implicitly allow read access to the relation and field. This is due to the nature of the SCREEN function that reads and then allows modification or deletion of the displayed data.

9.3.2 Query Modification

This protection method allows restrictions to be placed on the user's query tranparently to them. The restrictions are based on special assertions defined in the knowledge base. An example of a query modification might be that all secretaries can only see students in their own department. In this method, the user's query is modified and additional conditions are added to it.

In the system the query modification is implemented through the testing for the existence of specially named permanent assertions. These assertions can not be modified or overridden by users. The assertion name is of the following format QUERY-RESTRICTION-*relation-field*. The value of the assertion is then used as the restricting value when accessing the database. Currently only a single query restriction is supported for each field of each relation. If multiple restrictions or more complex restrictions are required, the author believes that

this is best accomplished through a *view*. A single assertion is used for both SYSTEM and USER rules. An example of the definition of an assertion for query modification based on the users department is seen in example 9–1.

Example 9–1: Query Modification Assertion

```
(set-assertion-permanent
      'QUERY-MODIFICATION-STUDENT_PROGRAM_HISTORY-DEPARTMENT
      (send me 'DEPARTMENT))
```

9.3.3 Context-Dependent Protection

Sensitive information can be classified in two categories: *context-dependent* and *context-independent*. In context-independent access control, no knowledge of previous accesses are used to determine if the current access is allowed. The current access is based on the conditions at the present time. Context-dependent access control uses knowledge of past accesses to determine if the current access is allowed. An example of context-dependent access is the following:

> *If item A and B are read, then item C can not be read. But if only item A or item B is read, then item C can be read.*

In the system, there is no specific support for context-dependent protection. Rather through the use of rules and assertions, an equivalent functionality can be achieved. As a user performs a specific access through the SCREEN function, an additional *action* can be performed which will set the value of an *assertion*. This same assertion can then be used as part of the conditions for another rule, that is specifically used to access another relation. The assertions must be *permanent assertions* so that their values are maintained across different menus. This is illustrated in example 9–2.

Example 9–2: Rules to Implement Context Sensitive Access

```
(user  ACCESS-A
       ((STRING= (GET-ASSERTION 'CONTROL1) "YES"))
       ((SCREEN '("A" "F1" "F2" "F3") 'READ
                "Reading A")
        (SET-ASSERTION 'CONTROL2 "YES")
        (SET-ASSERTION 'ACCESS-A "Y")) );
                        .
                        .
                        .
(user  ACCESS-B
       ((STRING= (GET-ASSERTION 'CONTROL4) "YES"))
       ((SCREEN '("B" "F1" "F2" "F3") 'READ
                "Reading B")
        (SET-ASSERTION 'CONTROL2 "YES")
        (SET-ASSERTION 'ACCESS-B "Y")) );
                        .
                        .
                        .
(user  ACCESS-C-1
       ((STRING= (GET-ASSERTION 'CONTROL5) "YES")
        (STRING= (GET-ASSERTION 'ACCESS-A) "Y")
        (NOT (STRING= (GET-ASSERTION 'ACCESS-B) "Y")))
       ((SCREEN '("C" "F1" "F2" "F3") 'READ
                "Reading C")
        (SET-ASSERTION 'CONTROL21 "YES"));
(user  ACCESS-C-2
       ((STRING= (GET-ASSERTION 'CONTROL5) "YES")
        (NOT (STRING= (GET-ASSERTION 'ACCESS-A) "Y"))
        (STRING= (GET-ASSERTION 'ACCESS-B) "Y"))
       ((SCREEN '("C" "F1" "F2" "F3") 'READ
                "Reading C")
        (SET-ASSERTION 'CONTROL21 "YES"));
```

9.4 System Failure

Should an unexpected error occur in the execution of the system, a global LISP error handler is invoked that will display the error message and terminate the system. Control is returned back to the VMS command language interface. The user can then make the necessary changes and reinvoke the system. Any active transactions on the database will have been rolled back.

CHAPTER 10
CONCLUSIONS AND RECOMMENDATIONS

10.1 Conclusions

The purpose of this thesis research was to develop a unique Office Information System that

- allowed users to define directly and provide direct input into the functionality of the system
- allowed user customisation
- integrated known software systems into a larger system
- introduced expert system like processing to the office domain
- provided multiple methods for task definition
- allowed changes in the system to be incorporated quickly
- avoided using traditional system developement techniques to eliminate frequent implementation problems
- provided a unique interface for accessing a database that uses database knowledge

The author was not attempting to formalise the office environment, or tasks, but rather create a system for use in the office environment, not using traditional system development techniques. The author believes that expert systems or expert-like systems are the systems of the future, including the area of Office Information Systems. The User-Definable Office Information is a realization of this. It does not solve all problems related to Information Systems, but it does provide an working system for examination that has new features and capabilities.

10.1.1 Customisation Capabilities

With this system, users have the ability to define the system on two different levels. First they are the subject matter experts, who provide all input to the knowledge engineer and database designer for task definition and database design. The developed system and tasks will function exactly as specified by the user. Second, the user has the ability to perform their own customisation. The areas of customisation are:

10.1.1.1 Menus

The menus that are displayed by the system can be created and maintained by users. They can create the menus through a simple text file. This can be created and updated using any of the various editors available on the system. The menus can be changed as users wish to change the selecting of an application in the system. Different users performing similar functions can have completely different menu structures. This will allow the system to adapt to users instead of users adapting to the system.

10.1.1.2 Rules

Users will work with the knowledge engineer to define the functionality of the various system applications through rules. Individual user customisation is capable through user rules and their execution at strategic times within an application. This feature is missing from many Office Information Systems. Users can now implement the various real world minor differences that exists in offices, that occur when different individuals perform the same task.

10.1.1.3 User Definitions

A flavour system, and its use in establishing the various user types of the system, will allow a close mapping between the structure of individuals in the office and the user types of the system. This could allow the appropriate functionality to be easily shared between different user types. Allowing system users to closely match office individuals is a feature which is missing from many Office Information Systems. It allows a better integration of the system into the real office environment.

10.1.2 Application Development

With this system, there exist three possible options for the development of specific tasks. This flexibility for tasks implementation is frequently missing in other systems. The three options for task implementation are:

- Database oriented
- Rule oriented
- Method oriented

10.1.2.1 Database Oriented

In this method, information related to a specific task would be included in the database as one or more relations. The functionality would then be implemented through the access of these relations. An example could be the checking of prerequisites for a student enrolling in a course. The data would be a relation containing the necessary prerequisites for a course. This relation would be checked to ensure compliance.

10.1.2.2 Rule Oriented

In this method, information related to a specific task would be included as rules in the Knowledge Base. Specific rules for function and possible user input data must be included. In the example of checking for prerequisites for a student enrolling in a course, specific rules for each course must be included in the rule base for all those courses that have prerequisites.

10.1.2.3 Method Oriented

In this method, functionality is embedded within the various methods associated with the user. Some rules will cause the method to be invoked for the current user. This method allows the rules to be common for all users. It also means that the functionality must be written in LISP using the various LISP function associated with this system.

10.1.3 The User-Database Interface

The User-Database interface provides an easy to use facility to access a relational database. Users do not need to learn any query language in order to access and manipulate data. The are presented with a full screen display of data which is an common occurrance in many applications. The interface dynamically builds and executes the users database queries allowing a virtual unlimited number of possible queries, as contrasted with other Office Information Systems which allow only limited number of queries to be performed. The interface also used database knowledge in joining together relations. This is a unique feature of the system.

10.1.4 Knowledge Based System

The system has rules, assertions and an inference engine, all the components of an expert system. Knowledge can be represented in the knowledge base and used then to form tasks, to

make decisions based on a limited amount of known information by inference. This capability is missing from many Office Information Systems. If the knowledge is embedded correctly, it can create a system which is capable of performing intelligent tasks for users in the office environment.

10.2 Recommendations

The author believes that improvements and changes could be made to the system. Additional related research is also possible related to this system. Under this section, improvements and changes to the current implementation of the system are discussed.

10.2.1 Database Interface

The current system is based on a VAX system running VMS using Rdb/VMS V3.0 as the database platform. Currently Rdb/VMS does not have an interpretive SQL interface, similar to the RDO interpretive interface. Should this SQL interface become available it would be the preferred interface. It would then allow the possible porting of the system to any computer that support Common LISP and a relational database with an interpretive SQL interface.

10.2.2 User-Database Interface

The User-Database Interface, through the SCREEN function could be extended to work on distributed database systems.

10.2.3 Rule Storage

Currently the rules are stored as a simple text file and read and loaded when the user invokes the application. Dynamic updating of the rules could be achieved on a user by user basis to allow them to test changes to applications. Allowing updates of rules to all active

users would be much more difficult, since it would require the rules to be stored in common memory and some form of synchronisation would be required to access and update them. It however is not an impossible task.

The rules could also be stored in the Rdb/VMS database instead of a text file. This would allow the database to truly contain all the data, rules and data records in relations.

10.2.4 Menu Definition Storage

Currently the menu definitions are stored as a simple text file, that is parsed and processed when the user invokes the application. These definitions could be stored in the Rdb/VMS database. This would then have one storage platform for all "data" required by the application.

10.2.5 Error handling

The system currently has minimal error handling. This could be improved to be more user friendly.

10.2.6 SCREEN System

The current SCREEN system performs minimal dynamic changes to the size of the screens created. This could be enhanced to have much more tailoring of the screens to match the data that is being displayed. It could even potentially change the size of the characters of the display for very large fields. This enhancements are possible using the existing VMS Screen Management Routines (SMG routines).

10.2.7 Rule Analysis

An additional feature could be added to perform some rule analysis and determine how complete the rules define the activities of the application.

10.2.8 Meta-Data Rules

A new type of rule could be added that could be used to control or determine how the system rules are processed. This could include the capability to have and specific different inference engines, or heuristics.

10.2.9 Rule, Screen, Function Formats

The current system was designed to test a theory, and as such lacks the polish and interfaces of a production system. The formats have generally followed LISP constructs which could be made more user friendly and pleasing.

CHAPTER 11
EXAMPLE SYSTEM

11.1 Sample Functionality

This chapter details some of the requirements in using the system to implemement an example tasks. Not all possible information is discussed, but rather the main areas. This information can be used as guidelines for the implememention of other tasks. The example system is for a university, in the area of student enrollment.

11.2 Rule Design

A critical part in the development of an expert system, is the mapping of real world or subject knowledge into the knowledge rules. This process in not simple, and is generally performed by a knowledge engineer. The definition of the rules, in the correct system format that map to the corresponding subject knowledge in not necessarily easily. It is not a direct parallel of traditional third generation language programming, as in COBOL or PASCAL. The rules form a series of tests on the system knowledge that can result in a series of actions occurring. Generally some trigger assertions are required to be initially set to some value(s) to establish some initial conditions for rules to be executed in. It is on this basis that the rules for the example system have been developed. Table 11–1 contains a list of the special assertions and their function. These assertions are special in that they are permanent and do not disappear from the system.

Table 11–1: Special Assertions

Name	Function
FUNCTION	Set by LISP functions prior to rule checking to establish a starting point. This assertion can be thought of as a switch for which rules are to fire. Its value may be changed by other rules to cause additional rules to fire.

Figure 11–1: Sample Rule

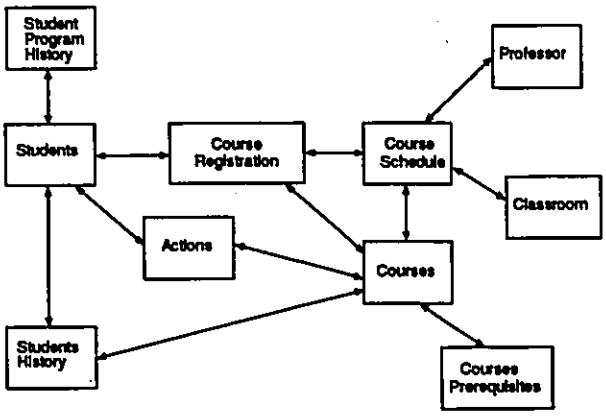
```
SECRETARY  DO-SOMETHING
      ((STRING= (GET-ASSERTION 'FUNCTION) "string"))
      (  ( action )
         ( action )
         ( action ) );
```

A portion of the rules for the example system are in Appendix D.

11.3 Example System Database Description

The logical database design for the example system is illustrated in figure 11–2.

Figure 11–2: Database Relations for System



Each of the relations is described in greater detail in the subsequent sections. The database was created as a single file database with a snapshot file for READ ONLY transaction. All the indexes that were defined were sorted.

11.3.1 Students

This relation contains information about each student. The fields are

Table 11–2: STUDENTS Relation Fields

Field Name	Datatype	Size	Description
STUDENT_ID	character	6	Student's Id number, unique to each student
LAST_NAME	character	15	Student's last name, family name
FIRST_NAME	character	15	Student's first name
MIDDLE_NAME	character	15	Student's middle name
STUDENT_PROGRAM	character	7	Indicates where an undergraduate, master's or Phd
PROGRAM_ENROLLMENT_DATE	date	6	The date the student has started the program. In the format YYMMDD.
ADDRESS	character	40	Street address of the student
CITY	character	20	City
PROVINCE	character	20	Province
POSTAL_CODE	character	7	Postal Code
DEPARTMENT_COMMENTS	character	50	Special comments by the Department

11.3.2 Student Program History

This relation contains information about each program that a student has been enrolled in. The fields are

Table 11–3: STUDENT_PROGRAM_HISTORY Relation Fields

Field Name	Datatype	Size	Description
STUDENT_ID	character	6	Student's Id number, unique to each student
STUDENT_PROGRAM	character	7	Indicates where an undergraduate, master's or Phd
FACULTY	character	20	Faculty of student's program
DEPARTMENT	character	20	Department of student's program
PROGRAM_ENROLLMENT_DATE	date	6	The date the student has started the program. In the format YYMMDD.
PROGRAM_COMPLETION_DATE	date	6	The date the student has completed the program. In the format YYMMDD.
PROGRAM_ACTIVE	character	1	Indicates whether the program is currently active or not
DEPARTMENT_COMMENTS	character	50	Special comments by the Department

11.3.3 Student History

This represents a list of all the courses the student has ever enrolled in. If the student completes the course, it contains their final mark.

Table 11–4: STUDENT_HISTORY Relation Fields

Field Name	Datatype	Size	Description
STUDENT_ID	character	6	Student's Id number, unique to each student
COURSE_ID	character	7	Course Id of a course the student has enrolled on.
DATE_ENROLLED	date	6	In the format YYMMDD.
DATE_DROPPED	date	6	In the format YYMMDD.
DATE_COMPLETED	date	6	In the format YYMMDD.
STATUS	text	1	Indicates the current status of this course for the student. Valid values are E (enrolled), P (pending enrolment), D (dropped)
MARK	number	2	Numeric mark
GRADE	character	2	In the form of A+, A, A-, ...
DEPARTMENT_COMMENTS	character	50	Special comments by the Department

11.3.4 Courses

This contains information about each different course. It contains the department and faculty that own the course.

Table 11–5: COURSES Relation Fields

Field Name	Datatype	Size	Description
COURSE_ID	character	7	Course Id for the course
COURSE_NAME	character	40	Name of the course
FACULTY	character	20	Faculty to which the course belongs
DEPARTMENT	character	20	Department to which the course belongs
CREDITS	number	1	The number of credits this course has associated with it.

11.3.5 Course Schedule

This contains information on each occurrence of a course. This includes the date and location of the course.

Table 11–6: COURSE_SCHEDULE Relation Fields

Field Name	Datatype	Size	Description
COURSE_ID	character	7	Course Id of the course
COURSE_DATE	date	6	In the format YYMMDD.
LOCATION_CODE	character	10	Primary location for the course
TIME	character	13	Time of the class in the format "DDD HHMM-HHMM".
LOCATION_CODE1	character	10	Additional location for the course
TIME1	character	13	Time of the class in the format "DDD HHMM-HHMM".
LOCATION_CODE2	character	10	Additional location for the course
TIME2	character	13	Time of the class in the format "DDD HHMM-HHMM".
LAB_LOCATION_CODE	character	10	Location for any laboratory work for the course
LAB_TIME	character	13	Time of the lab in the format "DDD HHMM-HHMM".
COURSE_ACTIVE	character	1	Indicates whether the course is current active for use, ie enrolling students. A course may be placed on a schedule well in advance, but no enrolments should take place. Valid values are A (active), I (inactive).
COURSE_STATUS	character	1	Indicates a status for the course. Valid values are F (full), S (space), C (cancelled).
PROFESSOR_ID	character		
MAXIMUM_STUDENTS	number	2	Maximum number of students in this class. The value is taken from the CLASSROOM relation.
CURRENT_ENROLLMENT	number	2	Current number of students in the class.

11.3.6 Course Registration

This contains the list of all the students who are currently enrolled on a course.

Table 11–7: COURSE_REGISTRATION Relation Fields

Field Name	Datatype	Size	Description
COURSE_ID	character	7	Course Id of the course the is enrolled in.
COURSE_DATE	date	6	In the format YYMMDD.
LOCATION_CODE	character	10	Building location code or name
STUDENT_ID	character	6	Student Id of the student enrolled
DATE_ENROLLED	date	6	In the format YYMMDD.
STUDENT_PROGRAM	character	7	Program to which the course is applied to.
REGISTRATION_STATUS	character	1	Indicates the current status of this course for the student. Valid values are E (enrolled), P (pending enrolment)

11.3.7 Professor

This contains information about each professor.

Table 11–8: PROFESSOR Relation Fields

Field Name	Datatype	Size	Description
PROFESSOR_ID	character	5	University id for the professor
PROFESSOR_FIRST_NAME	character	15	Professor's first name
PROFESSOR_LAST_NAME	character	20	Professor's last name
PHONE	character	12	Telephone number for the professor
OFFICE_LOCATION	character	10	Building code or name
FACULTY	character	20	Faculty to which the professor belongs
DEPARTMENT	character	20	Professor's department
DEGREES	character	50	Academic degrees held by the professor
TITLE	character	25	Professor's actual title

11.3.8 Classroom

This contains information about all the various locations for classes. It will indicate items like the class size and physical location.

Table 11–9: CLASSROOM Relation Fields

Field Name	Datatype	Size	Description
LOCATION_CODE	character	10	Building code or name of the room
FULL_NAME_DESCRIPTION	character	40	Full description of the location
MAXIMUM_STUDENTS	number	2	The maximum number of student that this location can hold.

11.3.9 Actions

This relation contains information on the various actions that different users must perform related to students and courses. Entries are generated in it based on other user activities.

Table 11–10: ACTIONS Relation Fields

Field Name	Datatype	Size	Description
FACULTY	character	20	Student's faculty
DEPARTMENT	character	20	Student's department
USER	character	13	User for action
ACTION_CODE	character	2	Code for action
PRIORITY	character	1	Priority for the action. Valid values are 1, 2 or 3.
STUDENT_ID	character	6	Student's ID number
COURSE_ID	character	7	Course id for action
COURSE_DATE	date	6	Date of course
LOCATION_CODE	character	10	Location of course
ACTION_TEXT	character	50	Comments about the action required

11.3.10 Course Prerequisites

This relation contains information about any prerequisite courses for a particular course. If a course has no prerequisites then there should be no data in this relation.

Table 11–11: COURSE_PREREQUISITES Relation Fields

Field Name	Datatype	Size	Description
COURSE_ID	character	7	Course Id
PREREQUISITE1	character	7	Course Id of first prerequisite course.
PREREQUISITE2	character	7	Course Id of second prerequisite course.
PREREQUISITE3	character	7	Course Id of third prerequisite course.
PREREQUISITE4	character	7	Course Id of fourth prerequisite course.
PREREQUISITE1_OR	character	7	Course Id of first prerequisite course where only one of several is required.
PREREQUISITE2_OR	character	7	Course Id of second prerequisite course where only one of several is required.
PREREQUISITE3_OR	character	7	Course Id of third prerequisite course where only one of several is required.
PREREQUISITE4_OR	character	7	Course Id of fourth prerequisite course where only one of several is required.
COREQUISITE1	character	7	Course Id of first corequisite course which is to be taken in conjunction with this course.
COREQUISITE2	character	7	Course Id of second corequisite course which is to be taken in conjunction with this course.

Appendix F contains the example MASTER-RELATIONS-FIELDS-LIST for the user ADMINISTRATOR.

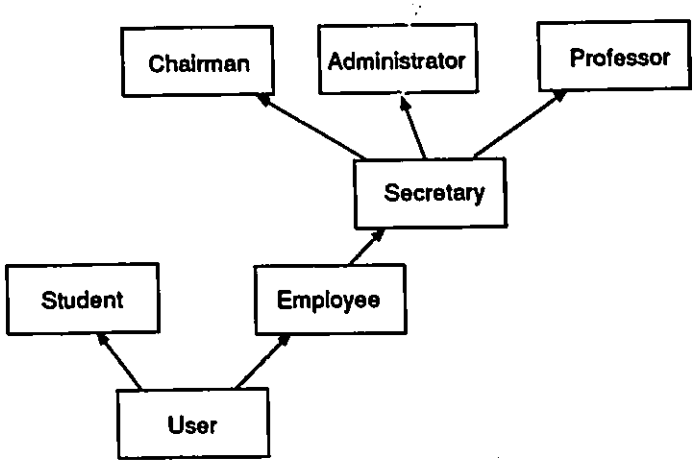
11.4 System User Definitions

Within this example system there are several different user types including the following

- Secretary
- Professor
- Administrator
- Chairman
- Student

Each user class has a function within the organisation with some specific and potentially common activities. The relationships between the different users is illustrated in figure 11–3

Figure 11–3: User relationships



11.4.1 User

Defines the lowest level entity with attributes that are common to all other users. Appendix B contains a listing of the flavour system definition for USER.

11.4.2 Employee

Defines common attributes for all employees of the university. Appendix B contains a partial listing of the flavour system definition of EMPLOYEE.

11.4.3 Student

This user has the ability to view the course schedule and classroom information. They are not employees of the university and as such there are certain attributes they do not have.

11.4.4 Secretary

This user has the ability to view student information as well as to perform some basic update of student information.

11.4.5 Professor

This user has the ability to view student information and course schedule information.

11.4.6 Chairman

This user has the ability to view all student information, professor information and course schedules. They may wish to get various reports on students and enrollments.

11.4.7 Administrator

This user has the ability to view and update student, course schedule and professor information. They generally perform the main update functions. They also enrol students on courses. Appendix C contains a partial lists of the definition of the Administrator. This includes the corresponding flavour and several method definitions.

11.5 Basic Relation Management

There are screens that map to each of the different relations in the sample database. These screens are used to perform the basic functions of *displaying, updating, deleting* and *adding* to the relations. The rules for these functions are common across all of the different user types. The action for each rule is to cause a specific method associated with each user to be executed.

Figure 11–4: Relation Maintenance Rules samples

```
SECRETARY  DISPLAY-STUDENT
     ((STRING= (GET-ASSERTION 'FUNCTION) "DISPLAY-STUDENT"))
     ((SEND ME 'DISPLAY-STUDENT));

SECRETARY  UPDATE-STUDENT
     ((STRING= (GET-ASSERTION 'FUNCTION) "UPDATE-STUDENT"))
     ((SEND ME 'UPDATE-STUDENT));
```

With this type of rule definition, the features of the flavour system are seen. It is very common that different users see subsets of all the fields of a relation. This is because some data is confidential or of no value or interest to that user. This subsetting is accomplished by the values used in the SCREEN function call in each of the methods for the different users. If the STUDENT user can not see certain information, then that field is simply omitted in the SCREEN function call within the method for that user.

11.6 Task Functionality

The functionality of a task is generated through a series of rules. The following sections give some examples of the example systems business rules and how they have been mapped into rules. The actual rules are listed in Appendix D, that implement the following items.

11.6.1 Enrolling a Student on a Course

This is a major activity of any school or learning institution. But there generally are a number of conditions that must be met for the student to successfully be registered in a course. For the sample system, the following business rules exist:

- A student can not enrol more that once in a course.
- A student can not automatically enrol in a course if they have previously dropped the course. Special permission is required.
- A student can not automatically enrol in a course if they have not taken all the necessary prerequisite courses.

11.6.2 Cancelling a Student from a Course

In the example system, the following business rule exists for cancelling a student from a course:

- A student can only be cancelled from a course they are enrolled in

11.7 Menu Definitions

A partial list of the menu definitions for the user ADMINISTRATOR is found in Appendix E.

APPENDIX A
REFERENCES AND RELATED PUBLICATIONS

[Alavi Jun84] Alavi, Maryan *An Assessment of the Prototyping Approach to Information Systems Development*, Communications of the ACM, June 1984, Volume 27, Number 6, pp 556-563.

[Appelbaum and Ruspini 85] Appelbaum, L., Ruspini, E.H., *ARIES: An approximate reasoning Inference Engine*, Approximate Reasoning in Expert Systems, M.M. Gupta, A. Kandel, W. Bandler, J.B. Kiszka (editors), North-Holland, 1985, pp 745-765.

[Aron 74] Aron, J.D., *The Program Development Process: Part 1, The Individual Programmer*, Addison-Wesley Publishing Company, 1974.

[Barker 87] Barker, Edmund, *4GLs - The Claims and Reality and their impact on Commercial Programming*, Software World, Vol 18, No. 3, 1987, pp 12-15.

[Boehm Dec76] Boehm, B., *"Software Engineering", IEEE Transactions on Computers*, Vol. C-25, No. 12 (December 1976), pp 1226-1241.

[Bracchi and Pernici Apr84] Bracchi, Giampio and Pernici, Barbara, *The Design Requirements of Office Systems*, ACM Transactions on Office Information Systems, Vol. 2, No. 2, April 1984, pp 151-170.

[Cardenas 79] Cardenas, Alfonso F., *Data Base Management Systems*, Allyn and Bacon, 1979.

[Cargill 89] Cargill, Carl F., *Information Technology Standardization: Theory, Process, and Organizations*, Digital Press, 1989.

[Chamberlin et al. 78] Chamberlin, D., et al., *Data Base System Authorization*, Foundations of Secure Computing, edited by Richard A. DeMillo et al., Academic Press, 1978, pp 39-52.

[Cox Jan84] Cox, Brad J., *Message/Object Programminng: An Evolutionary Change in Programming Technology*, IEEE Software, January 1984, pp 50-61.

[D'Anglo et al. 85] D'Angelo, Antonio et al., *A Mechanism for representing and using Meta-Knowledge in Rule-Based Systems*, Approximate Reasoning in Expert Systems, M.M. Gupta, A Kandel, W. Bandler, J.B. Kiszka (editors), North-Holland, 1985, pp 781-798.

[Dachouffe and Lesuisse Aug88] Dachouffe, Nadine and Lesuisse, Roland, *O.I.D.A.: Office Information Systems Design Aids*, Proceedings of the IFIP 8.4 Working Conference Office Information System: The Design Process, Linz, Austria, August 15-17, 1988, pp 9-39.

[Dachouffe and Lesuisse Aug88a] Op. Cit. pp 10.

[Date 87] Date, C.J., *An Introduction to Database Systems*, Volume1, Fourth Edition, Addison Wesley, 1987.

[DeMillo et al. 78] DeMillo, Richard, et al., *Combinatorial Inference*, Foundations of Secure Computing, edited by Richard A. DeMillo et al., Academic Press, 1978, pp 27-35.

[Denning 78] Denning, Dorothy E., *A Review of Research on Statistical Data Base Security*, Foundations of Secure Computing, edited by Richard A. DeMillo et al., Academic Press, 1978, pp 15-25.

[Dobkin 78] Dobkin, David, *The Foundations of Secure Computation*, Foundations of Secure Computing, edited by Richard A. DeMillo et al., Academic Press, 1978, pp 1-12.

[Fahlman 79] Fahlman, Scott E., *NETL: A System for representing and using Real-World Knowledge*, The MIT Press, 1979.

[Fikes and Kehler Sep85] Fikes, Richard and Kehler, Tom, *The Role of Frame-Based Representation in Reasoning*, Communications of the ACM, September 1985, Volume 28, Number 9, pp 904-920.

[Good et al. Oct84] Good, Michael D., et al., *Building a User-Derived Interface*, Communications of the ACM, October 1984, Volume 27, Number 10, pp 1032-1043.

[Güting 89] Güting, Ralf Hartmut, et al., *An Algebra for Structured Office Documents*, ACM Transactions on Information Systems, Vol. 7, No. 2, April 1989, pp 123-157.

[Hägglund Aug88] Hägglund, Sture, *Iterative Design and Adaptive Maintenance of Knowledge-Based Office Systems*, Proceedings of the IFIP 8.4 Working Conference Office Information System: The Design Process, Linz, Austria, August 15-17, 1988, pp 1-8.

[Hägglund Aug88a] Op. cit. pp xx

[Harmon and King 85] Harmon, Paul and King, David *EXPERT SYSTEMS: Artificial Intelligence in Business*, John Wiley & Sons, Inc., 1985.

[Hayes-Roth Sep85] Hayes-Roth, Fredrick, *Rule-Based Systems*, Communications of the ACM, September 1985, Volume 28, Number 9, pp 921-932.

[Hsiao et al 79] Hsiao, D.K., and Kerr, D.S., and Madnick, S.E., *Computer Security*, Academic Press, 1975.

[Jackson 86] Jackson, Peter, *Introduction to Expert Systems*, Addison-Wesley Publishing Company, 1986.

[Jenny 85] Jenny, Christian J., *Requirements on Expert Systems as seen by an Insurance Company*, ARTIFICIAL INTELLIGENCE: Towards Practical Application, T. Bernold and G.Albers (editors), North-Holland, 1985, pp 87-96.

[Kaplinsky 84] Kaplinsky, Raphael, *Automation: the technology and society*, Longman Group Limited, 1984.

[Keller 87] Keller, Robert, *Expert System Technology: Development and Application*, Yourdon Press, Prentice-Hall, Inc., 1987.

[Leavitt 65] Leavitt, H.J., *Applying organizational change in industry: Structural Technological and Humanistic Approaches, Handbook of Organizations*, Rand-MacNally, 1965.

[Martin 85] Martin, James, *Fourth Generation Languages, Volumne 1, Principles*, Prentice-Hall, 1985.

[McGowan and Kelly 75] McGowan, Clement L., and Kelly, John R., *Top-Down Structured Programming Techniques*, Mason/Charter Publishers, Inc., 1975.

[Misra and Jalics Jul88] Misra, Santosh K., and Jalics, Paul J., *Third-Generation versus Fourth-Generation Software Development*, IEEE Software, July 1988, pp 8-14.

[Motro Apr88] Motro, Amihai, *VAGUE: A User Interface to Relational Databases that Permits Vague Queries*, ACM Transactions on Office Information Systems, Vol. 6, No. 2, April 1988, pp 173-183.

[Myers 78] Myers, Glenford J., *Composite/Structured Design*, Van Nostrand Reinhold Company, 1978.

[Page-Jones 80] Page-Jones, Meiler, *The Practical Guide to Structured Systems Design*, Yourdon Press, 1980.

[Pressman 82] Pressman, Roger S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill Inc., 1982.

[Raman and Kerre 85] Raman, B., and Kerre, E.E., *Application of Fuzzy Programming to Ship Steering*, Approximate Reasoning in Expert Systems, M.M. Gupta, A. Kandel, W. Bandler, J.B. Kiszka (editors), North-Holland, 1985, pp 719-743.

[Rees 85] Rees, Brian,*Artificial Intelligence in a Large-Scale Enterprise - The experience of Digital Equipment Corporation*, ARTIFICIAL INTELLIGENCE: Towards Practical Application, T. Bernold and G.Albers (editors), North-Holland, 1985, pp 67-76.

[Reiss 78] Reiss, Steven P., *Medians and Database Security*, Foundations of Secure Computing, edited by Richard A. DeMillo et al., Academic Press, 1978, pp 57-91.

[Rubinstien and Hersh 84] Rubinstien, Richard, and Hersh, Harry, *The Human Factor: Designing Computer Systems for People*, Digital Press, 1984.

[Sagalowicz 85] Sagalowicz, Daniel, *Expert Systems in Services Sectors: Use of Expert Systems in six Sample Cases*, ARTIFICIAL INTELLIGENCE: Towards
Practical Application, T. Bernold and G.Albers (editors),
North-Holland, 1985, pp 77-79.

[Scheifler et al. 88] Scheifler, Robert W., and Gettys, James, and Newman, Ron, *X WINDOW SYSTEM C Library and Protocol Reference*, Digital Press, 1988.

[Siegel 86] Siegel, Paul, *EXPERT SYSTEMS: A Non-programmer's Guide to Development and Applications*, TAB BOOKS Inc., 1986.

[Snaman and Thiel Sep87] Snaman Jr., William E., and Thiel, David W., *The VAX/VMS Distributed Lock Manager*, Digital Technical Journal, Number 5, September 1987, Digital Equipment Corporartion, pp 29-44.

[Steele 84] Steele Jr., Guy L., *Common LISP The Language*, Digital Press, 1984.

[Stevens et al. May74] Stevens, W., Myers, G., Constantine, L., *Structured Design*, IBM Systems Journal, Vol. 13, No. 2, May 1974, pp 115-139.

[Suchman Apr88] Suchman, Lucy, *Designing with the User*, ACM Transactions on Office Information Systems, Vol. 6, No. 2, April 1988, pp 173-183.

[Summers Oct90] Summers, Eric, *ES: A Public Domain Expert System*, Byte, October 1990, pp 289-292.

[Tanimoto 87] Tanimoto, Steven L., *THE ELEMENTS of Artificial Intelligence*, Computer Science Press, Inc., 1987.

[Thompson 85] Thompson, Henry, *Office Automation: A field for Applied Artifical Intelligence*, ARTIFICIAL INTELLIGENCE: Towards Practical Application, T. Bernold and G.Albers (editors), North-Holland, 1985, pp 81-85.

[Wierzchon 85] Wierzchon, S.T., *Mathematical Tools for Knowledge Representation*, Approximate Reasoning in Expert Systems, M.M. Gupta, A. Kandel, W. Bandler, J.B. Kiszka (editors), North-Holland, 1985, pp 61-69.

[Wiig 85] Wiig, Karl, *Market Trends in Artificial Intelligence in the United States and Japan*, ARTIFICIAL INTELLIGENCE: Towards Practical Application, T. Bernold and G.Albers (editors), North-Holland, 1985, pp 169-180.

[Winston and Horn 84] Winston, Patrick Henry, and Horn, Berthold Klaus Paul, *LISP*, Second Edition, Addison-Wesley Publishing Company, 1984.

[Witten and Bramwell 85] Witten, Ian H., and Bramwell, Bob, *A System for Interactive viewing of Structured Documents*, Communications of the ACM, Volume 28, Number 3, March 1985, pp 280-288.

[Yang 88] Yang, Jianhua, *A Proposal for incorporating rules in ODA-Documents*, ACM Transactions on Office Information Systems, Vol. 2, No. 2, April 1984, pp 131-147.

[Zadeh_85] Zadeh, L.A., *The Role of Fuzzy Logic in the management of Uncertainty in Expert Systems*, Approximate Reasoning in Expert Systems, M.M. Gupta, A. Kandel, W. Bandler, J.B. Kiszka (editors), North-Holland, 1985, pp 3-31.

[Zemankova-Leech 85] Zemankova-Leech, M., *Uncertainty Propagation to Expert Systems*, Approximate Reasoning in Expert Systems, M.M. Gupta, A. Kandel, W. Bandler, J.B. Kiszka (editors), North-Holland, 1985, pp 529-548.

[Intro 87] *Introduction to Artifical Intelligence*, January 1987, Center of the International Cooperation for Computerization, Japan.

[Rdb/VMS Design 88] *VAX Rdb/VMS Guide to Database Design and Definition*, VAX Rdb/VMS V3.0, AA-N034C-TE, July 1988, Digital Equipment Corporation, Maynard, Massachuetts.

[Rdb/VMS Maint 88] *VAX Rdb/VMS Guide to Database Maintenance and Performance*, VAX Rdb/VMS V3.0, AA-N035D-TE, July 1988, Digital Equipment Corporation, Maynard, Massachuetts.

[Rdb/VMS Programming 88] *VAX Rdb/VMS Guide to Programming*, VAX Rdb/VMS V3.0, AA-N038D-TE, July 1988, Digital Equipment Corporation, Maynard, Massachuetts.

[Rdb/VMS Reference 88] *VAX Rdb/VMS Reference Manual*, AA-N039E-TE, VAX Rdb/VMS V3.0, July 1988 Digital Equipment Corporation, Maynard, Massachuetts.

[VMS SMG 88] *VMS RTL Screen Management (SMG$) Manual*, AA-LA77A-TE, April 1988, Digital Equipment Corporation, Maynard, Massachuetts.

APPENDIX B
DEFINITIONS FOR USER AND EMPLOYEE

```
(DEFFLAVOR 'USER
           '(I-AM-A)
           NIL
           '(GETTABLE SETTABLE))

(DEFFLAVOR 'EMPLOYEE
           '(LAST_NAME FIRST_NAME EMPLOYEE_ID JOB FACULTY DEPARTMENT)
           'USER
           '(GETTABLE SETTABLE))


(DEFMETHOD 'EMPLOYEE
           'LIST-ALL-USER-RULES
           '(&AUX OUT-WINDOW-ID CONTEXT-LIST-COPY)
           '( (setq screen-length 19)
              (SETQ OUT-WINDOW-ID (CREATE-AND-MAP-WINDOW screen-length
                                                         65 3 3
                                                         " User RULES "))

              (do ((i 0 (+ i 1))
                   (c 0))
                  ((or (> i (length master-rule-list))
                       (null (nth i master-rule-list))))
                (cond
                  ((equal 'user (rule-type
                                  (eval (read-from-string
                                          (format nil "rule-~a"
                                             (nth i master-rule-list))))))
                   (WRITE-WINDOW  OUT-WINDOW-ID
                           (FORMAT NIL "~A" (nth i master-rule-list)))
                   (setq c (+ c 1))
                   (cond
                     ((>= c 17)
                      (set-cursor out-window-id 19 1)
                      (READ-FROM-WINDOW OUT-WINDOW-ID 2
                                        "Press RETURN to continue")
                      (setq c 0)
                      (clear-window out-window-id)
                      (set-cursor out-window-id 1 1))
                     )
                   )
                  )
                )
              (set-cursor out-window-id 19 1)
              (READ-FROM-WINDOW OUT-WINDOW-ID 2
                   "THE END ... Press RETURN to continue")
              (REMOVE-WINDOW  OUT-WINDOW-ID))
 )
```

```
(DEFMETHOD 'EMPLOYEE
           'DATABASE-OPEN
           '( database-name &AUX window-id FOUND-FLAG)
           '( (COND
                ((NULL DATABASE-NAME)
                 (SETQ WINDOW-ID (CREATE-AND-MAP-WINDOW 10 50 3 3
                                  " Databases Available " ))
                 (WRITE-WINDOW WINDOW-ID " ")
                 (DO*
                   ((I 0 (+ I 1)))
                   ((>= I (LENGTH AVAILABLE-DATABASES)))
                   (COND
                     ((STRING= (NTH I AVAILABLE-DATABASES)
                               CURRENT-DATABASE)
                      (WRITE-WINDOW WINDOW-ID
                               (FORMAT NIL " ~A) ~A  <- Currently open"
                                       (+ I 1) (NTH I AVAILABLE-DATABASES))))
                     (T
                      (WRITE-WINDOW WINDOW-ID
                               (FORMAT NIL " ~A) ~A"
                                       (+ I 1) (NTH I AVAILABLE-DATABASES))))
                     )
                  )
                 (SET-CURSOR WINDOW-ID 10 1)
                 (SETQ DATABASE-NAME (READ-STRING-FROM-WINDOW WINDOW-ID 30
                                      "Enter Database name: "))
                 (REMOVE-WINDOW WINDOW-ID))
               )
              (SETQ FOUND-FLAG NIL)
              (DO*
                ((I 0 (+ I 1))
                 (MAX-LENGTH (LENGTH AVAILABLE-DATABASES)))
                ((>= I MAX-LENGTH))
                (COND
                  ((STRING= (NTH I AVAILABLE-DATABASES) DATABASE-NAME)
                   (SETQ FOUND-FLAG T)
                   (COND
                     ((NULL (RDB-DATABASE-OPEN database-name))
                      (DISPLAY-MESSAGE "Error in opening the database"))
                     (t
                      (SETF CURRENT-DATABASE DATABASE-NAME))
                    )
                  )
               )
              (COND
                ((NULL FOUND-FLAG)
                 (DISPLAY-MESSAGE "No such database"))
               )
              ) ))

(DEFMETHOD 'EMPLOYEE
           'DATABASE-CLOSE
           '()
           '( (RDB-DATABASE-CLOSE)
              (SETF CURRENT-DATABASE NIL)
              (SET-ASSERTION 'TRANSACTION-ACTIVE "N") ))
```

```
(DEFMETHOD 'EMPLOYEE
           'DATABASE-LIST
           '( &AUX WINDOW-ID )
           '( (SETQ WINDOW-ID (CREATE-AND-MAP-WINDOW 10 50 3 3
                                  " Databases Available " ))
              (WRITE-WINDOW WINDOW-ID " ")
              (DO*
                ((I 0 (+ I 1)))
                ((>= I (LENGTH AVAILABLE-DATABASES)))
                (COND
                  ((STRING= (NTH I AVAILABLE-DATABASES)
                            CURRENT-DATABASE)
                   (WRITE-WINDOW WINDOW-ID
                            (FORMAT NIL " ~A) ~A  <- Currently open"
                                    (+ I 1) (NTH I AVAILABLE-DATABASES))))
                  (T
                   (WRITE-WINDOW WINDOW-ID
                            (FORMAT NIL " ~A) ~A"
                                    (+ I 1) (NTH I AVAILABLE-DATABASES))))
                 )
               )
              (SET-CURSOR WINDOW-ID 10 1)
              (READ-FROM-WINDOW WINDOW-ID 2 "Press Return to continue")
              (REMOVE-WINDOW WINDOW-ID))
)
```

APPENDIX C
DEFINITIONS FOR ADMINISTRATOR

```
(DEFFLAVOR 'ADMINISTRATOR
           NIL
           'SECRETARY
           NIL)

(DEFMETHOD 'ADMINISTRATOR
           'DISPLAY-SELF
           '(&AUX WINDOW-ID)
           '( (SETQ WINDOW-ID (CREATE-AND-MAP-WINDOW 12 50 4 4 NIL 'REVERSE))
              (WRITE-WINDOW WINDOW-ID (FORMAT NIL "You are a ~A"
                                              (SEND ME 'I-AM-A)))
              (WRITE-WINDOW WINDOW-ID " ")
              (WRITE-WINDOW WINDOW-ID (FORMAT NIL "Employee Id:  ~A"
                                              (SEND ME 'EMPLOYEE_ID)))
              (WRITE-WINDOW WINDOW-ID (FORMAT NIL "Last Name:    ~A"
                                              (SEND ME 'LAST_NAME)))
              (WRITE-WINDOW WINDOW-ID (FORMAT NIL "First Name:   ~A"
                                              (SEND ME 'FIRST_NAME)))
              (WRITE-WINDOW WINDOW-ID (FORMAT NIL "Faculty:      ~A"
                                              (SEND ME 'FACULTY)))
              (WRITE-WINDOW WINDOW-ID (FORMAT NIL "Department:   ~A"
                                              (SEND ME 'DEPARTMENT)))
              (WRITE-WINDOW WINDOW-ID " ")
              (READ-FROM-WINDOW WINDOW-ID 2 "Press Return")
              (REMOVE-WINDOW WINDOW-ID))
           )

(DEFMETHOD 'ADMINISTRATOR
           'DISPLAY-STUDENT
           '( )
           '( (SCREEN '(("STUDENTS" "STUDENT_ID" "LAST_NAME" "FIRST_NAME"
                         "MIDDLE_NAME" "ADDRESS" "CITY"
                         "PROVINCE" "POSTAL_CODE" "PHONE"
                         "DEPARTMENT_COMMENTS")) 'READ
                         "Students")
            )
)
```

```
(DEFMETHOD 'ADMINISTRATOR
           'UPDATE-STUDENT
           '( )
           '( (SCREEN '(("STUDENTS" "STUDENT_ID" "LAST_NAME" "FIRST_NAME"
                        "MIDDLE_NAME" "ADDRESS" "CITY"
                        "PROVINCE" "POSTAL_CODE" "PHONE"
                        "DEPARTMENT_COMMENTS")) 'UPDATE
                        "Students")
             )
)

(DEFMETHOD 'ADMINISTRATOR
           'DELETE-STUDENT
           '( )
           '( (SCREEN '(("STUDENTS" "STUDENT_ID" "LAST_NAME" "FIRST_NAME"
                        "MIDDLE_NAME" "ADDRESS" "CITY"
                        "PROVINCE" "POSTAL_CODE" "PHONE"
                        "DEPARTMENT_COMMENTS")) 'DELETE
                        "Students")
             )
)

(DEFMETHOD 'ADMINISTRATOR
           'ADD-STUDENT
           '( )
           '( (SCREEN '(("STUDENTS" "STUDENT_ID" "LAST_NAME" "FIRST_NAME"
                        "MIDDLE_NAME" "ADDRESS" "CITY"
                        "PROVINCE" "POSTAL_CODE" "PHONE"
                        "DEPARTMENT_COMMENTS")) 'ADD
                        "Students")
             )
)


(DEFMETHOD 'ADMINISTRATOR
           'DISPLAY-STUDENT_HISTORY
           '( )
           '( (SCREEN '(("STUDENT_HISTORY" "STUDENT_ID" "COURSE_ID"
                        "STUDENT_PROGRAM" "DATE_ENROLLED" "DATE_DROPPED"
                        "DATE_COMPLETED" "STATUS" "MARK" "GRADE"
                        "DEPARTMENT_COMMENTS"))
                    'READ "Student History")
             )
)

(DEFMETHOD 'ADMINISTRATOR
           'UPDATE-STUDENT_HISTORY
           '( )
           '( (SCREEN '(("STUDENT_HISTORY" "STUDENT_ID" "COURSE_ID"
                        "STUDENT_PROGRAM" "DATE_ENROLLED" "DATE_DROPPED"
                        "DATE_COMPLETED" "STATUS" "MARK" "GRADE"
                        "DEPARTMENT_COMMENTS"))
                    'UPDATE "Student History")
             )
)
```

```
(DEFMETHOD 'ADMINISTRATOR
           'ADD-STUDENT_HISTORY
           '( )
           '( (SCREEN '(("STUDENT_HISTORY" "STUDENT_ID" "COURSE_ID"
                         "STUDENT_PROGRAM" "DATE_ENROLLED" "DATE_DROPPED"
                         "DATE_COMPLETED" "STATUS" "MARK" "GRADE"
                         "DEPARTMENT_COMMENTS"))
                      'ADD "Student History")
            )
)

(DEFMETHOD 'ADMINISTRATOR
           'DELETE-STUDENT_HISTORY
           '( )
           '( (SCREEN '(("STUDENT_HISTORY" "STUDENT_ID" "COURSE_ID"
                         "STUDENT_PROGRAM" "DATE_ENROLLED" "DATE_DROPPED"
                         "DATE_COMPLETED" "STATUS" "MARK" "GRADE"
                         "DEPARTMENT_COMMENTS"))
                      'DELETE "Student History")
            )
)
```

APPENDIX D
RULES FOR ADMINISTRATOR

```
ADMINISTRATOR DATABASE-OPEN
     ((STRING= (GET-ASSERTION 'FUNCTION) "DATABASE-OPEN"))
     ((SEND ME 'DATABASE-OPEN NIL));

ADMINISTRATOR DATABASE-CLOSE
     ((STRING= (GET-ASSERTION 'FUNCTION) "DATABASE-CLOSE"))
     ((SEND ME 'DATABASE-CLOSE));

ADMINISTRATOR DATABASE-LIST
     ((STRING= (GET-ASSERTION 'FUNCTION) "DATABASE-LIST"))
     ((SEND ME 'DATABASE-LIST));

ADMINISTRATOR  DISPLAY-SELF
      ((STRING= (GET-ASSERTION 'FUNCTION) "DISPLAY-SELF"))
      ((SEND ME 'DISPLAY-SELF));

ADMINISTRATOR  DISPLAY-STUDENT
      ((STRING= (GET-ASSERTION 'FUNCTION) "DISPLAY-STUDENT"))
      ((SEND ME 'DISPLAY-STUDENT));

ADMINISTRATOR  UPDATE-STUDENT
      ((STRING= (GET-ASSERTION 'FUNCTION) "UPDATE-STUDENT"))
      ((SEND ME 'UPDATE-STUDENT));

ADMINISTRATOR  ADD-STUDENT
      ((STRING= (GET-ASSERTION 'FUNCTION) "ADD-STUDENT"))
      ((SEND ME 'ADD-STUDENT));

ADMINISTRATOR  DELETE-STUDENT
      ((STRING= (GET-ASSERTION 'FUNCTION) "DELETE-STUDENT"))
      ((SEND ME 'DELETE-STUDENT));

ADMINISTRATOR  DISPLAY-STUDENT_HISTORY
      ((STRING= (GET-ASSERTION 'FUNCTION) "DISPLAY-STUDENT_HISTORY"))
      ((SEND ME 'DISPLAY-STUDENT_HISTORY));

ADMINISTRATOR  UPDATE-STUDENT_HISTORY
      ((STRING= (GET-ASSERTION 'FUNCTION) "UPDATE-STUDENT_HISTORY"))
      ((SEND ME 'UPDATE-STUDENT_HISTORY));

ADMINISTRATOR  ADD-STUDENT_HISTORY
      ((STRING= (GET-ASSERTION 'FUNCTION) "ADD-STUDENT_HISTORY"))
      ((SEND ME 'ADD-STUDENT_HISTORY));
```

```
ADMINISTRATOR  DELETE-STUDENT_HISTORY
      ((STRING= (GET-ASSERTION 'FUNCTION) "DELETE-STUDENT_HISTORY"))
      ((SEND ME 'DELETE-STUDENT_HISTORY));

ADMINISTRATOR  DISPLAY-COURSE_REGISTRATION-AND-STUDENT
      ((STRING= (GET-ASSERTION 'FUNCTION)
                              "DISPLAY-COURSE_REGISTRATION-AND-STUDENT"))
      ((SCREEN '(("COURSES"              "COURSE_ID"
                                         "COURSE_NAME"
                                         "FACULTY"
                                         "DEPARTMENT")
                 ("COURSE_REGISTRATION" "COURSE_DATE"
                                         "LOCATION_CODE"
                                         "STUDENT_ID"
                                         "DATE_ENROLLED"
                                         "STUDENT_PROGRAM")
                 ("STUDENTS" "LAST_NAME" "FIRST_NAME"))
               'READ "Course Registration and Student Information" ));

ADMINISTRATOR  DISPLAY-STUDENT-PROGRAM-HISTORY
      ((STRING= (GET-ASSERTION 'FUNCTION) "DISPLAY-STUDENT-PROGRAM-HISTORY"))
      ((SCREEN '(("STUDENTS" "STUDENT_ID" "LAST_NAME" "FIRST_NAME")
                 ("STUDENT_PROGRAM_HISTORY" "STUDENT_PROGRAM"
                       "FACULTY" "DEPARTMENT"
                       "PROGRAM_ENROLLMENT_DATE" "PROGRAM_COMPLETION_DATE"
                       "PROGRAM_ACTIVE")) 'READ "Student Program History"));

ADMINISTRATOR ENROLL-STUDENT
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT"))
      ((SCREEN '(("STUDENTS"
                     "STUDENT_ID" "LAST_NAME" "FIRST_NAME"
                     "MIDDLE_NAME")
                 ("STUDENT_PROGRAM_HISTORY"
                     "STUDENT_PROGRAM" "FACULTY" "DEPARTMENT"
                     "PROGRAM_ACTIVE"))
               'READ "Enroll a Student 1"
               '(start-new-transaction)
               '(save-fields ("STUDENTS" "STUDENT_ID")
                             ("STUDENT_PROGRAM_HISTORY" "STUDENT_PROGRAM"
                                 "PROGRAM_ACTIVE" "FACULTY" "DEPARTMENT")) )
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT2") );

ADMINISTRATOR ENROLL-STUDENT2
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT2")
       (STRING= (GET-ASSERTION 'STUDENT_PROGRAM_HISTORY-PROGRAM_ACTIVE)
                "Y")
       (NOT (IF-NULL 'STUDENTS-STUDENT_ID)) )
      ((SCREEN '(("COURSE_SCHEDULE" "COURSE_ID" "COURSE_DATE" "LOCATION_CODE")
                 ("COURSES" "COURSE_NAME")) 'READ "Enroll a Student 2"
               '(save-fields ("COURSE_SCHEDULE" "COURSE_ID"
                                   "COURSE_DATE" "LOCATION_CODE")))
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT2A") );
```

```
ADMINISTRATOR ENROLL-STUDENT2A
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT2A"))
      ((SCREEN '(("STUDENT_HISTORY" "STUDENT_ID" "COURSE_ID"
                                    "STUDENT_PROGRAM" "DATE_ENROLLED"
                                    "DATE_DROPPED"))
               'READ "Check for Previous Enrollment"
               '(invisible)
               '(pre-load ("STUDENT_HISTORY"
                           ("STUDENT_ID"
                              (GET-ASSERTION 'STUDENTS-STUDENT_ID))
                           ("COURSE_ID"
                              (GET-ASSERTION 'COURSE_SCHEDULE-COURSE_ID))
                           ("STUDENT_PROGRAM"
                              (GET-ASSERTION
                                'STUDENT_PROGRAM_HISTORY-STUDENT_PROGRAM))))
               '(SAVE-fields ("STUDENT_HISTORY" "DATE_ENROLLED"
                                                "DATE_DROPPED"))
               )
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT3"));

ADMINISTRATOR ENROLL-STUDENT3-ERROR1
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT3")
       (NOT (IF-NULL 'STUDENT_HISTORY-DATE_ENROLLED))
       (IF-NULL 'STUDENT_HISTORY-DATE_DROPPED))
      ((MESSAGE-DISPLAY "Student already enrolled in this course"));

ADMINISTRATOR ENROLL-STUDENT3-ERROR2
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT3")
       (NOT (IF-NULL 'STUDENT_HISTORY-DATE_ENROLLED))
       (NOT (IF-NULL 'STUDENT_HISTORY-DATE_DROPPED))  )
      ((MESSAGE-DISPLAY
         "Student previous dropped this course, special OK required")
       (SCREEN '(("ACTIONS" "FACULTY" "DEPARTMENT"
                            "USER" "ACTION_CODE"
                            "PRIORITY" "STUDENT_ID"
                            "COURSE_ID" "COURSE_DATE"
                            "LOCATION_CODE" "ACTION_TEXT"))
               'ADD "Create Action"
               '(invisible)
               '(PRE-LOAD  ("ACTIONS"
                            ("FACULTY" (GET-ASSERTION
                                        'STUDENT_PROGRAM_HISTORY-FACULTY))
                            ("DEPARTMENT"
                                (GET-ASSERTION
                                  'STUDENT_PROGRAM_HISTORY-DEPARTMENT))
                            ("USER" "ADMINISTRATOR")
                            ("ACTION_CODE" "DA")
                            ("PRIORITY" "3")
                            ("STUDENT_ID" (GET-ASSERTION
                                           'STUDENTS-STUDENT_ID))
                            ("COURSE_ID" (GET-ASSERTION
                                          'COURSE_SCHEDULE-COURSE_ID))
                            ("COURSE_DATE"
                               (GET-ASSERTION 'COURSE_SCHEDULE-COURSE_DATE))
                            ("LOCATION_CODE" (GET-ASSERTION
                                         'COURSE_SCHEDULE-LOCATION_CODE))
                            ("ACTION_TEXT"
                                "Student dropped and re-enrolled")))
       )) ;
```

```
ADMINISTRATOR ENROLL-STUDENT3
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT3")
       (STRING/= (GET-ASSERTION 'OPERATION-ABORTED) "Y")
       (IF-NULL 'STUDENT_HISTORY-DATE_ENROLLED)
       (IF-NULL 'STUDENT_HISTORY-DATE_DROPPED)
       (NOT (IF-NULL 'COURSE_SCHEDULE-COURSE_ID))
       (NOT (IF-NULL 'COURSE_SCHEDULE-COURSE_DATE))
       (NOT (IF-NULL 'COURSE_SCHEDULE-LOCATION_CODE)) )
      ((SCREEN '(("COURSE_PREREQUISITES"
                  "COURSE_ID"
                  "PREREQUISITE1" "PREREQUISITE2" "PREREQUISITE3"
                  "PREREQUISITE4"
                  "PREREQUISITE1_OR" "PREREQUISITE2_OR"
                  "PREREQUISITE3_OR" "PREREQUISITE4_OR"
                  "COREQUISITE1" "COREQUISITE2"))
               'READ "Get Preqs"
               '(invisible)
               '(save-all-fields)
               '(pre-load ("COURSE_PREREQUISITES"
                           ("COURSE_ID"
                               (GET-ASSERTION 'COURSE_SCHEDULE-COURSE_ID))))
       )
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4"));

ADMINISTRATOR ENROLL-STUDENT4
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4")
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE1)
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE2)
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE3)
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE4)
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE1_OR)
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE2_OR)
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE3_OR)
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE4_OR)
       (IF-NULL 'COURSE_PREREQUISITES-COREQUISITE1)
       (IF-NULL 'COURSE_PREREQUISITES-COREQUISITE2))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT5"));

ADMINISTRATOR ENROLL-STUDENT4-CHECK-PREQ1
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4")
       (NOT (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE1)))
      ((SCREEN '(("STUDENT_HISTORY"  "STUDENT_ID" "COURSE_ID"))
               'READ "Check Preq 1"
               '(invisible)
               '(pre-load ("STUDENT_HISTORY"
                           ("STUDENT_ID"
                               (GET-ASSERTION 'STUDENTS-STUDENT_ID))
                           ("COURSE_ID"
                               (GET-ASSERTION
                                   'COURSE_PREREQUISITES-PREREQUISITE1))))
               '(SAVE-PREFIX "PREQ1")
               '(SAVE-FIELDS ("STUDENT_HISTORY" "COURSE_ID")))
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-PREQ1")
       );
```

```
ADMINISTRATOR ENROLL-STUDENT4-PREQ1-OK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ1")
       (STRING= (GET-ASSERTION 'PREQ1-COURSE_ID)
                (GET-ASSERTION 'course_prerequisites-prerequisite1)))
      ((message-display "Student has Preq1")
       (set-assertion 'function "ENROLL-STUDENT4-CHECK-PREQ2"));

ADMINISTRATOR ENROLL-STUDENT4-PREQ1-NOTOK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ1")
       (not (STRING= (GET-ASSERTION 'PREQ1-COURSE_ID)
                     (GET-ASSERTION 'course_prerequisites-prerequisite1))))
      ((message-display "Student does not have Preq1"));

ADMINISTRATOR ENROLL-STUDENT4-PREQ1-MISSING
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ1")
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE1))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-PREQ1_OR"));

ADMINISTRATOR ENROLL-STUDENT4-CHECK-PREQ2
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-PREQ2")
       (NOT (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE2)))
      ((SCREEN '(("STUDENT_HISTORY"  "STUDENT_ID" "COURSE_ID"))
               'READ "Check Preq 2"
               '(invisible)
               '(pre-load ("STUDENT_HISTORY"
                           ("STUDENT_ID"
                               (GET-ASSERTION 'STUDENTS-STUDENT_ID))
                           ("COURSE_ID"
                               (GET-ASSERTION
                                   'COURSE_PREREQUISITES-PREREQUISITE2))))
               '(SAVE-PREFIX "PREQ2")
               '(SAVE-FIELDS ("STUDENT_HISTORY" "COURSE_ID")))
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-PREQ2")
       );

ADMINISTRATOR ENROLL-STUDENT4-PREQ2-MISSING
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-PREQ2")
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE2))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-PREQ1_OR"));

ADMINISTRATOR ENROLL-STUDENT4-PREQ2-OK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ2")
       (STRING= (GET-ASSERTION 'PREQ2-COURSE_ID)
                (GET-ASSERTION 'course_prerequisites-prerequisite2)))
      ((message-display "Student has Preq2")
       (print (get-assertion 'course_prerequisites-prerequisite2))
       (set-assertion 'function "ENROLL-STUDENT4-CHECK-PREQ3"));

ADMINISTRATOR ENROLL-STUDENT4-PREQ2-NOTOK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ2")
       (not (STRING= (GET-ASSERTION 'PREQ2-COURSE_ID)
                     (GET-ASSERTION 'course_prerequisites-prerequisite2))))
      ((message-display "Student does not have Preq2"));
```

```
ADMINISTRATOR ENROLL-STUDENT4-CHECK-PREQ3
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-PREQ3")
       (NOT (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE3)))
      ((SCREEN '(("STUDENT_HISTORY"  "STUDENT_ID" "COURSE_ID"))
               'READ "Check Preq 3"
               '(invisible)
               '(pre-load ("STUDENT_HISTORY"
                           ("STUDENT_ID"
                               (GET-ASSERTION 'STUDENTS-STUDENT_ID))
                           ("COURSE_ID"
                               (GET-ASSERTION
                                   'COURSE_PREREQUISITES-PREREQUISITE3))))
               '(SAVE-PREFIX "PREQ3")
               '(SAVE-FIELDS ("STUDENT_HISTORY" "COURSE_ID")))
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-PREQ3")
       );

ADMINISTRATOR ENROLL-STUDENT4-PREQ3-MISSING
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-PREQ3")
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE3))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-PREQ1_OR"));

ADMINISTRATOR ENROLL-STUDENT4-PREQ3-OK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ3")
       (STRING= (GET-ASSERTION 'PREQ3-COURSE_ID)
                (GET-ASSERTION 'course_prerequisites-prerequisite3)))
      ((message-display "Student has Preq3")
       (set-assertion 'function "ENROLL-STUDENT4-CHECK-PREQ4"));

ADMINISTRATOR ENROLL-STUDENT4-PREQ3-NOTOK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ3")
       (not (STRING= (GET-ASSERTION 'PREQ3-COURSE_ID)
                     (GET-ASSERTION 'course_prerequisites-prerequisite3))))
      ((message-display "Student does not have Preq3"));

ADMINISTRATOR ENROLL-STUDENT4-CHECK-PREQ4
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-PREQ4")
       (NOT (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE4)))
      ((SCREEN '(("STUDENT_HISTORY"  "STUDENT_ID" "COURSE_ID"))
               'READ "Check Preq 4"
               '(invisible)
               '(pre-load ("STUDENT_HISTORY"
                           ("STUDENT_ID"
                               (GET-ASSERTION 'STUDENTS-STUDENT_ID))
                           ("COURSE_ID"
                               (GET-ASSERTION
                                   'COURSE_PREREQUISITES-PREREQUISITE4))))
               '(SAVE-PREFIX "PREQ4")
               '(SAVE-FIELDS ("STUDENT_HISTORY" "COURSE_ID")))
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-PREQ4")
       );

ADMINISTRATOR ENROLL-STUDENT4-PREQ4-MISSING
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-PREQ4")
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE4))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-PREQ1_OR"));
```

```
ADMINISTRATOR ENROLL-STUDENT4-PREQ4-OK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ4")
       (STRING= (GET-ASSERTION 'PREQ4-COURSE_ID)
                (GET-ASSERTION 'course_prerequisites-prerequisite4)))
      ((message-display "Student has Preq4")
       (set-assertion 'function "ENROLL-STUDENT4-CHECK-COPREQ1"));

ADMINISTRATOR ENROLL-STUDENT4-PREQ4-NOTOK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ4")
       (not (STRING= (GET-ASSERTION 'PREQ4-COURSE_ID)
                     (GET-ASSERTION 'course_prerequisites-prerequisite4))))
      ((message-display "Student does not have Preq4"));

ADMINISTRATOR ENROLL-STUDENT4-CHECK-PREQ1_OR
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-PREQ1_OR")
       (NOT (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE1_OR)))
      ((SCREEN '(("STUDENT_HISTORY"  "STUDENT_ID" "COURSE_ID"))
               'READ "Check Preq_OR 1"
               '(invisible)
               '(pre-load ("STUDENT_HISTORY"
                           ("STUDENT_ID"
                               (GET-ASSERTION 'STUDENTS-STUDENT_ID))
                           ("COURSE_ID"
                               (GET-ASSERTION
                                 'COURSE_PREREQUISITES-PREREQUISITE1_OR))))
               '(SAVE-PREFIX "PREQ1-OR")
               '(SAVE-FIELDS ("STUDENT_HISTORY" "COURSE_ID")))
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-PREQ1_OR")
       );

ADMINISTRATOR ENROLL-STUDENT4-PREQ1_OR-MISSING
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-PREQ1_OR")
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE1_OR))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-COREQ1"));

ADMINISTRATOR ENROLL-STUDENT4-PREQ1_OR-OK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ1_OR")
       (STRING= (GET-ASSERTION 'COURSE_PREREQUISITES-PREREQUISITE1_OR))
                (GET-ASSERTION 'PREQ1-OR-COURSE_ID)))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-COREQ1"));

ADMINISTRATOR ENROLL-STUDENT4-PREQ1_OR-NOTOK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ1_OR")
       (NOT (STRING= (GET-ASSERTION 'COURSE_PREREQUISITES-PREREQUISITE1_OR))
                     (GET-ASSERTION 'PREQ1-OR-COURSE_ID))))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-PREQ2_OR"));
```

```
ADMINISTRATOR ENROLL-STUDENT4-CHECK-PREQ2_OR
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-PREQ2_OR")
       (NOT (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE2_OR)))
      ((SCREEN '(("STUDENT_HISTORY"  "STUDENT_ID" "COURSE_ID"))
               'READ "Check Preq_OR 2"
               '(invisible)
               '(pre-load ("STUDENT_HISTORY"
                           ("STUDENT_ID"
                               (GET-ASSERTION 'STUDENTS-STUDENT_ID))
                           ("COURSE_ID"
                               (GET-ASSERTION
                                 'COURSE_PREREQUISITES-PREREQUISITE2_OR))))
               '(SAVE-PREFIX "PREQ2-OR")
               '(SAVE-FIELDS ("STUDENT_HISTORY" "COURSE_ID")))
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-PREQ2_OR")
       );

ADMINISTRATOR ENROLL-STUDENT4-PREQ2_OR-MISSING
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-PREQ2_OR")
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE2_OR))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-COREQ1"));

ADMINISTRATOR ENROLL-STUDENT4-PREQ2_OR-OK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ2_OR")
       (STRING= (GET-ASSERTION 'COURSE_PREREQUISITES-PREREQUISITE2_OR))
                (GET-ASSERTION 'PREQ2-OR-COURSE_ID)))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-COREQ1"));

ADMINISTRATOR ENROLL-STUDENT4-PREQ2_OR-NOTOK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ2_OR")
       (NOT (STRING= (GET-ASSERTION 'COURSE_PREREQUISITES-PREREQUISITE2_OR))
                     (GET-ASSERTION 'PREQ2-OR-COURSE_ID))))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-PREQ3_OR"));

ADMINISTRATOR ENROLL-STUDENT4-CHECK-PREQ3_OR
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-PREQ3_OR")
       (NOT (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE3_OR)))
      ((SCREEN '(("STUDENT_HISTORY"  "STUDENT_ID" "COURSE_ID"))
               'READ "Check Preq_OR 3"
               '(invisible)
               '(pre-load ("STUDENT_HISTORY"
                           ("STUDENT_ID"
                               (GET-ASSERTION 'STUDENTS-STUDENT_ID))
                           ("COURSE_ID"
                               (GET-ASSERTION
                                 'COURSE_PREREQUISITES-PREREQUISITE3_OR))))
               '(SAVE-PREFIX "PREQ3-OR")
               '(SAVE-FIELDS ("STUDENT_HISTORY" "COURSE_ID")))
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-PREQ3_OR")
       );

ADMINISTRATOR ENROLL-STUDENT4-PREQ3_OR-MISSING
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-PREQ3_OR")
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE3_OR))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-COREQ1"));
```

```
ADMINISTRATOR ENROLL-STUDENT4-PREQ3_OR-OK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ3_OR")
       (STRING= (GET-ASSERTION 'COURSE_PREREQUISITES-PREREQUISITE3_OR))
                (GET-ASSERTION 'PREQ3-OR-COURSE_ID)))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-COREQ1"));

ADMINISTRATOR ENROLL-STUDENT4-PREQ3_OR-NOTOK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ3_OR")
       (NOT (STRING= (GET-ASSERTION 'COURSE_PREREQUISITES-PREREQUISITE3_OR))
                     (GET-ASSERTION 'PREQ3-OR-COURSE_ID))))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-PREQ4_OR"));

ADMINISTRATOR ENROLL-STUDENT4-CHECK-PREQ4_OR
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-PREQ4_OR")
       (NOT (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE4_OR)))
      ((SCREEN '(("STUDENT_HISTORY"  "STUDENT_ID" "COURSE_ID"))
               'READ "Check Preq_OR 4"
               '(invisible)
               '(pre-load ("STUDENT_HISTORY"
                           ("STUDENT_ID"
                               (GET-ASSERTION 'STUDENTS-STUDENT_ID))
                           ("COURSE_ID"
                               (GET-ASSERTION
                                 'COURSE_PREREQUISITES-PREREQUISITE4_OR))))
               '(SAVE-PREFIX "PREQ4-OR")
               '(SAVE-FIELDS ("STUDENT_HISTORY" "COURSE_ID")))
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-PREQ4_OR")
       );

ADMINISTRATOR ENROLL-STUDENT4-PREQ4_OR-MISSING
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-PREQ4_OR")
       (IF-NULL 'COURSE_PREREQUISITES-PREREQUISITE4_OR))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-COREQ1"));

ADMINISTRATOR ENROLL-STUDENT4-PREQ4_OR-OK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ4_OR")
       (STRING= (GET-ASSERTION 'COURSE_PREREQUISITES-PREREQUISITE4_OR))
                (GET-ASSERTION 'PREQ4-OR-COURSE_ID)))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-COREQ1"));

ADMINISTRATOR ENROLL-STUDENT4-PREQ4_OR-NOTOK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-PREQ4_OR")
       (NOT (STRING= (GET-ASSERTION 'COURSE_PREREQUISITES-PREREQUISITE4_OR))
                     (GET-ASSERTION 'PREQ4-OR-COURSE_ID))))
      ((MESSAGE-DISPLAY "Student has not preqs_OR"));
```

```
ADMINISTRATOR ENROLL-STUDENT4-CHECK-COREQ1
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-COREQ1")
       (NOT (IF-NULL 'COURSE_PREREQUISITES-COREQUISITE1)))
      ((SCREEN '(("STUDENT_HISTORY"  "STUDENT_ID" "COURSE_ID"))
               'READ "Check Coreq 1"
               '(invisible)
               '(pre-load ("STUDENT_HISTORY"
                           ("STUDENT_ID"
                               (GET-ASSERTION 'STUDENTS-STUDENT_ID))
                           ("COURSE_ID"
                               (GET-ASSERTION
                                 'COURSE_PREREQUISITES-COREQUISITE1))))
               '(SAVE-PREFIX "COREQ1")
               '(SAVE-FIELDS ("STUDENT_HISTORY" "COURSE_ID")))
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-COREQ1")
       );

ADMINISTRATOR ENROLL-STUDENT4-COREQ1-MISSING
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-COREQ1")
       (IF-NULL 'COURSE_PREREQUISITES-COREQUISITE1))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT5"));

ADMINISTRATOR ENROLL-STUDENT4-COREQ1-OK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-COREQ1")
       (STRING= (GET-ASSERTION 'COURSE_PREREQUISITES-COREQUISITE1)
                (GET-ASSERTION 'COREQ1-COURSE_ID)))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-CHECK-COREQ2"));

ADMINISTRATOR ENROLL-STUDENT4-COREQ1-NOTOK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-COREQ1")
       (NOT (STRING= (GET-ASSERTION 'COURSE_PREREQUISITES-COREQUISITE1)
                     (GET-ASSERTION 'COREQ1-COURSE_ID))))
      ((MESSAGE-DISPLAY "Student has not the Coreq"));

ADMINISTRATOR ENROLL-STUDENT4-CHECK-COREQ2
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-COREQ2")
       (NOT (IF-NULL 'COURSE_PREREQUISITES-COREQUISITE2)))
      ((SCREEN '(("STUDENT_HISTORY"  "STUDENT_ID" "COURSE_ID"))
               'READ "Check Coreq 2"
               '(invisible)
               '(pre-load ("STUDENT_HISTORY"
                           ("STUDENT_ID"
                               (GET-ASSERTION 'STUDENTS-STUDENT_ID))
                           ("COURSE_ID"
                               (GET-ASSERTION
                                 'COURSE_PREREQUISITES-COREQUISITE2))))
               '(SAVE-PREFIX "COREQ2")
               '(SAVE-FIELDS ("STUDENT_HISTORY" "COURSE_ID")))
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT4-COREQ2")
       );

ADMINISTRATOR ENROLL-STUDENT4-COREQ2-MISSING
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-CHECK-COREQ2")
       (IF-NULL 'COURSE_PREREQUISITES-COREQUISITE2))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT5"));
```

```
ADMINISTRATOR ENROLL-STUDENT4-COREQ2-OK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-COREQ2")
       (STRING= (GET-ASSERTION 'COURSE_PREREQUISITES-COREQUISITE2)
                (GET-ASSERTION 'COREQ1-COURSE_ID)))
      ((SET-ASSERTION 'FUNCTION "ENROLL-STUDENT5"));

ADMINISTRATOR ENROLL-STUDENT4-COREQ2-NOTOK
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT4-COREQ2")
       (NOT (STRING= (GET-ASSERTION 'COURSE_PREREQUISITES-COREQUISITE2)
                     (GET-ASSERTION 'COREQ1-COURSE_ID))))
      ((MESSAGE-DISPLAY "Student has not the Coreq"));

ADMINISTRATOR ENROLL-STUDENT5
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT5"))
      ((ENABLE-USER-RULES)
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT6"));

ADMINISTRATOR ENROLL-STUDENT6
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT6")
       (IF-NULL 'STUDENT_HISTORY-DATE_ENROLLED)
       (IF-NULL 'STUDENT_HISTORY-DATE_DROPPED)
       (NOT (IF-NULL 'COURSE_SCHEDULE-COURSE_ID))
       (NOT (IF-NULL 'COURSE_SCHEDULE-COURSE_DATE))
       (NOT (IF-NULL 'COURSE_SCHEDULE-LOCATION_CODE)) )
      ((SCREEN '(("COURSE_REGISTRATION"
                      "STUDENT_ID" "STUDENT_PROGRAM" "COURSE_ID" "COURSE_DATE"
                      "LOCATION_CODE" "DATE_ENROLLED"
                      "REGISTRATION_STATUS"))
               'ADD "Enroll a Student 3"
               '(pre-load ("COURSE_REGISTRATION"
                           ("STUDENT_ID"
                             (GET-ASSERTION 'STUDENTS-STUDENT_ID))
                           ("STUDENT_PROGRAM"
                             (GET-ASSERTION
                               'STUDENT_PROGRAM_HISTORY-STUDENT_PROGRAM))
                           ("COURSE_ID"
                             (GET-ASSERTION 'COURSE_SCHEDULE-COURSE_ID))
                           ("COURSE_DATE"
                             (GET-ASSERTION 'COURSE_SCHEDULE-COURSE_DATE))
                           ("LOCATION_CODE"
                             (GET-ASSERTION 'COURSE_SCHEDULE-LOCATION_CODE))
                           ("REGISTRATION_STATUS" "E")
                          ))
               '(SAVE-FIELDS ("COURSE_REGISTRATION" "DATE_ENROLLED")) )
       (SET-ASSERTION 'FUNCTION "ENROLL-STUDENT7"));
```

```
ADMINISTRATOR ENROLL-STUDENT7
      ((STRING= (GET-ASSERTION 'FUNCTION) "ENROLL-STUDENT7")
       (NOT (IF-NULL 'COURSE_REGISTRATION-DATE_ENROLLED)))
      ((SCREEN '(("STUDENT_HISTORY" "STUDENT_ID"
                                    "COURSE_ID"
                                    "STUDENT_PROGRAM"
                                    "DATE_ENROLLED"
                                    "STATUS"))
                'ADD "Invisibe Screen"
                '(pre-load ("STUDENT_HISTORY"
                            ("STUDENT_ID"
                              (GET-ASSERTION 'STUDENTS-STUDENT_ID))
                            ("COURSE_ID"
                              (GET-ASSERTION 'COURSE_SCHEDULE-COURSE_ID))
                            ("STUDENT_PROGRAM"
                              (GET-ASSERTION
                                'STUDENT_PROGRAM_HISTORY-STUDENT_PROGRAM))
                            ("STATUS" "E")
                            ("DATE_ENROLLED"
                              (GET-ASSERTION
                                'COURSE_REGISTRATION-DATE_ENROLLED))))
                '(finish-transaction)
                '(invisible)) );

ADMINISTRATOR CANCEL-STUDENT
      ((STRING= (GET-ASSERTION 'FUNCTION) "CANCEL-STUDENT"))
      ((SCREEN '(("COURSE_REGISTRATION"
                  "STUDENT_ID" "STUDENT_PROGRAM" "COURSE_ID"
                  "COURSE_DATE" "LOCATION_CODE" "DATE_ENROLLED"
                  "REGISTRATION_STATUS")
                 ("STUDENTS" "LAST_NAME" "FIRST_NAME"))
                'READ "Cancel a Student 1"
                '(start-new-transaction)
                '(save-fields ("COURSE_REGISTRATION"
                               "STUDENT_ID" "COURSE_ID" "STUDENT_PROGRAM"
                               "COURSE_DATE" "LOCATION_CODE"
                               "DATE_ENROLLED" "REGISTRATION_STATUS")))
       (SET-ASSERTION 'FUNCTION "CANCEL-STUDENT2"));

ADMINISTRATOR CANCEL-STUDENT2
      ((STRING= (GET-ASSERTION 'FUNCTION) "CANCEL-STUDENT2")
       (NOT (IF-NULL 'COURSE_REGISTRATION-COURSE_ID))
       (NOT (IF-NULL 'COURSE_REGISTRATION-STUDENT_ID))
       (NOT (IF-NULL 'COURSE_REGISTRATION-STUDENT_PROGRAM))
       (NOT (IF-NULL 'COURSE_REGISTRATION-LOCATION_CODE))
       (NOT (IF-NULL 'COURSE_REGISTRATION-COURSE_DATE))
       (NOT (IF-NULL 'COURSE_REGISTRATION-REGISTRATION_STATUS)))
      ((SCREEN '(("STUDENT_HISTORY"
                  "STUDENT_ID"
                  "STUDENT_PROGRAM"
                  "COURSE_ID"
                  "DATE_ENROLLED"
                  "DATE_DROPPED"
                  "STATUS"))
                'UPDATE "Update Student History Information"
                '(save-fields ("STUDENT_HISTORY"
                               "DATE_DROPPED"))
                '(update-protected ("STUDENT_HISTORY" "STATUS"
                                                      "STUDENT_ID"
                                                      "STUDENT_PROGRAM"
```

```
                                                    "DATE_ENROLLED"
                                                    "COURSE_ID"))
                '(pre-load ("STUDENT_HISTORY"
                            ("STUDENT_ID"
                               (GET-ASSERTION
                                   'COURSE_REGISTRATION-STUDENT_ID))
                            ("STUDENT_PROGRAM"
                               (GET-ASSERTION
                                   'COURSE_REGISTRATION-STUDENT_PROGRAM))
                            ("COURSE_ID"
                               (GET-ASSERTION
                                   'COURSE_REGISTRATION-COURSE_ID))
                            ("DATE_ENROLLED"
                               (GET-ASSERTION
                                   'COURSE_REGISTRATION-DATE_ENROLLED)))))
        (SET-ASSERTION 'FUNCTION "CANCEL-STUDENT3")) ;

ADMINISTRATOR CANCEL-STUDENT3
      ((STRING=   (GET-ASSERTION 'FUNCTION) "CANCEL-STUDENT3")
       (STRING/= (GET-ASSERTION 'OPERATION-ABORTED) "Y"))
      ((SCREEN '(("STUDENT_HISTORY"
                  "STUDENT_ID"
                  "STUDENT_PROGRAM"
                  "COURSE_ID"
                  "DATE_ENROLLED"
                  "DATE_DROPPED"
                  "STATUS"))
               'UPDATE "Update Student History Information"
               '(INVISIBLE)
               '(UPDATE-FIELDS ("STUDENT_HISTORY"
                                ("STATUS" "D")))
               '(pre-load ("STUDENT_HISTORY"
                           ("STUDENT_ID"
                              (GET-ASSERTION
                                  'COURSE_REGISTRATION-STUDENT_ID))
                           ("STUDENT_PROGRAM"
                              (GET-ASSERTION
                                  'COURSE_REGISTRATION-STUDENT_PROGRAM))
                           ("COURSE_ID"
                              (GET-ASSERTION
                                  'COURSE_REGISTRATION-COURSE_ID))
                           ("DATE_ENROLLED"
                              (GET-ASSERTION
                                  'COURSE_REGISTRATION-DATE_ENROLLED)))))
        (SET-ASSERTION 'FUNCTION "CANCEL-STUDENT4")) ;
```

```
ADMINISTRATOR CANCEL-STUDENT4
      ((STRING= (GET-ASSERTION 'FUNCTION) "CANCEL-STUDENT4")
       (NOT (IF-NULL 'STUDENT_HISTORY-DATE_DROPPED)))
      ((SCREEN '(("COURSE_REGISTRATION"
                       "COURSE_ID" "COURSE_DATE" "LOCATION_CODE"
                       "STUDENT_ID"))
               'DELETE "Remove student from registration"
               '(invisible)
               '(finish-transaction)
               '(pre-load ("COURSE_REGISTRATION"
                           ("COURSE_ID"
                              (GET-ASSERTION 'COURSE_REGISTRATION-COURSE_ID))
                           ("COURSE_DATE"
                              (GET-ASSERTION
                                   'COURSE_REGISTRATION-COURSE_DATE))
                           ("LOCATION_CODE"
                              (GET-ASSERTION
                                   'COURSE_REGISTRATION-LOCATION_CODE))
                           ("STUDENT_ID"
                              (GET-ASSERTION
                                   'COURSE_REGISTRATION-STUDENT_ID)))))
       );
```

APPENDIX E
USER ADMINISTRATOR MENU DEFINITIONS

```
MENU
  NAME ADMINISTRATOR-MENU
  USER    (ADMINISTRATOR)
  TITLE "Administrator MENU";

OPTION
  OPTION S
  MENU ADMINISTRATOR-MENU
  TEXT "Student Sub-system"
  USER    (ADMINISTRATOR)
  NEXT-MENU ADMINISTRATOR-STUDENT-MENU;

OPTION
  OPTION C
  MENU ADMINISTRATOR-MENU
  TEXT "Course Sub-system"
  USER    (ADMINISTRATOR)
  NEXT-MENU ADMINISTRATOR-COURSE-MENU;

OPTION
  OPTION CL
  MENU ADMINISTRATOR-MENU
  TEXT "Classroom Sub-system"
  USER    (ADMINISTRATOR)
  NEXT-MENU CLASSROOM-MENU;

OPTION
  OPTION P
  MENU ADMINISTRATOR-MENU
  TEXT "Professor Sub-system"
  USER    (ADMINISTRATOR)
  NEXT-MENU PROFESSOR-MENU;

OPTION
  OPTION A
  MENU ADMINISTRATOR-MENU
  TEXT "Action Sub-system"
  USER    (ADMINISTRATOR)
  NEXT-MENU ACTIONS-MENU;
```

```
MENU
  NAME ADMINISTRATOR-STUDENT-MENU
  USER   (ALL)
  TITLE "Administrator Student System MENU";

OPTION
  OPTION S
  MENU ADMINISTRATOR-STUDENT-MENU
  TEXT "Student Sub-system"
  USER   (ADMINISTRATOR)
  NEXT-MENU STUDENT-MENU;

OPTION
  OPTION SH
  MENU ADMINISTRATOR-STUDENT-MENU
  TEXT "Student History Sub-system"
  USER   (ADMINISTRATOR)
  NEXT-MENU STUDENT_HISTORY-MENU;

OPTION
  OPTION SPH
  MENU ADMINISTRATOR-STUDENT-MENU
  TEXT "Student Program History Sub-system"
  USER   (ADMINISTRATOR)
  NEXT-MENU STUDENT_PROGRAM_HISTORY-MENU;


MENU
  NAME ADMINISTRATOR-COURSE-MENU
  USER   (ALL)
  TITLE "Administrator Course System MENU";

OPTION
  OPTION C
  MENU ADMINISTRATOR-COURSE-MENU
  TEXT "Course Sub-system"
  USER   (ADMINISTRATOR)
  NEXT-MENU COURSE-MENU;

OPTION
  OPTION CS
  MENU ADMINISTRATOR-COURSE-MENU
  TEXT "Course Schedule Sub-system"
  USER   (ADMINISTRATOR)
  NEXT-MENU COURSE_SCHEDULE-MENU;

OPTION
  OPTION CR
  MENU ADMINISTRATOR-COURSE-MENU
  TEXT "Course Registration Sub-system"
  USER   (ADMINISTRATOR)
  NEXT-MENU COURSE_REGISTRATION-MENU;
```

```
OPTION
  OPTION CP
  MENU ADMINISTRATOR-COURSE-MENU
  TEXT "Course Prerequisites Sub-system"
  USER    (ADMINISTRATOR)
  NEXT-MENU COURSE_PREREQUISITES-MENU;


MENU
  NAME STUDENT-MENU
  USER    (ALL)
  TITLE "Student MENU";

OPTION
  OPTION L
  MENU STUDENT-MENU
  TEXT "List Student Personal Information"
  USER    (ALL)
  NEXT-FUNCTION DISPLAY-STUDENT;

OPTION
  OPTION U
  MENU STUDENT-MENU
  TEXT "Update Student Personal Information"
  USER    (ALL)
  NEXT-FUNCTION UPDATE-STUDENT;

OPTION
  OPTION A
  MENU STUDENT-MENU
  TEXT "Add New Student"
  USER    (ADMINISTRATOR)
  NEXT-FUNCTION ADD-STUDENT;

OPTION
  OPTION D
  MENU STUDENT-MENU
  TEXT "Delete a Student"
  USER    (ADMINISTRATOR)
  NEXT-FUNCTION DELETE-STUDENT;

OPTION
  OPTION DPH
  MENU STUDENT-MENU
  TEXT "Display Student Program History"
  USER    (ALL)
  NEXT-FUNCTION DISPLAY-STUDENT-PROGRAM-HISTORY;

OPTION
  OPTION DCH
  MENU STUDENT-MENU
  TEXT "Display Student Course History"
  USER    (ALL)
  NEXT-FUNCTION DISPLAY-STUDENT-COURSE-HISTORY;
```

```
OPTION
  OPTION E
  MENU STUDENT-MENU
  TEXT "Enroll a Student on a Course"
  USER   (ADMINISTRATOR)
  NEXT-FUNCTION ENROLL-STUDENT;
```

APPENDIX F

EXAMPLE UNIVERSITY DATABASE RELATION/FIELDS DEFINITIONS

```
(SETQ MASTER-RELATION-field-information
   '(("STUDENTS"
                              ( ("STUDENT_HISTORY" "STUDENT_ID" "STUDENT_ID")
                                ("STUDENT_PROGRAM_HISTORY" "STUDENT_ID"
                                                           "STUDENT_ID")
                                ("COURSE_REGISTRATION" "STUDENT_ID"
                                                       "STUDENT_ID")
                                ("ACTIONS"   "STUDENT_ID"  "STUDENT_ID"))
                              ( ("STUDENT_ID"                 6 TEXT
                                   ("RMW" "R"))
                                ("LAST_NAME"                 15 TEXT
                                   ("RMW" "R"))
                                ("FIRST_NAME"                15 TEXT
                                   ("RMW" "R"))
                                ("MIDDLE_NAME"               15 TEXT
                                   ("RMW" "R"))
                                ("ADDRESS"                   40 TEXT
                                   ("RMW" "R"))
                                ("CITY"                      20 TEXT
                                   ("RMW" "R"))
                                ("PROVINCE"                  20 TEXT
                                   ("RMW" "R"))
                                ("POSTAL_CODE"                7 TEXT
                                   ("RMW" "R"))
                                ("PHONE"                     12 TEXT
                                   ("RMW" "R"))
                                ("DEPARTMENT_COMMENTS"       50 TEXT
                                   ("RMW" "R")))
                            ("RMDW" "R")
      )
```

```
("STUDENT_HISTORY"
                        (   ("STUDENTS" "STUDENT_ID" "STUDENT_ID")
                            ("STUDENT_PROGRAM_HISTORY" "STUDENT_ID"
                                                       "STUDENT_ID")
                            ("COURSE_REGISTRATION" "STUDENT_ID"
                                                   "STUDENT_ID")
                            ("ACTIONS"     "STUDENT_ID" "STUDENT_ID"))
                        (   ("STUDENT_ID"        6 TEXT
                                ("RMW" "R"))
                            ("COURSE_ID"         7 TEXT
                                ("RMW" "R"))
                            ("STUDENT_PROGRAM"   7 TEXT
                                ("RMW" "R"))
                            ("DATE_ENROLLED"     6 DATE
                                ("RMW" "R"))
                            ("DATE_DROPPED"      6 DATE
                                ("RMW" "R"))
                            ("DATE_COMPLETED"    6 DATE
                                ("RMW" "R"))
                            ("STATUS"            1 TEXT
                                ("RMW" "R"))
                            ("MARK"              2 NUMBER
                                ("RMW" "R"))
                            ("GRADE"             2 TEXT
                                ("RMW" "R"))
                            ("DEPARTMENT_COMMENTS" 50 TEXT
                                ("RMW" "R")))
                        ("RMWD" "R")
)
("STUDENT_PROGRAM_HISTORY"
                        (   ("STUDENTS" "STUDENT_ID" "STUDENT_ID")
                            ("STUDENT_HISTORY" "STUDENT_ID"
                                               "STUDENT_ID")
                            ("ACTIONS"  "STUDENT_ID" "STUDENT_ID")
                            ("COURSE_REGISTRATION" "STUDENT_ID"
                                                   "STUDENT_ID"))
                        (   ("STUDENT_ID"              7 TEXT
                                ("RMW" "R"))
                            ("STUDENT_PROGRAM"         7 TEXT
                                ("RMW" "R"))
                            ("FACULTY"                20 TEXT
                                ("RMW" "R"))
                            ("DEPARTMENT"             20 TEXT
                                ("RMW" "R"))
                            ("PROGRAM_ENROLLMENT_DATE" 6 DATE
                                ("RMW" "R"))
                            ("PROGRAM_COMPLETION_DATE" 6 DATE
                                ("RMW" "R"))
                            ("PROGRAM_ACTIVE"          1 TEXT
                                ("RMW" "R"))
                            ("DEPARTMENT_COMMENTS"    50 TEXT
                                ("RMW" "R")))
                        ("RDMW" "R")
)
```

```
("COURSES"
                        (   ("COURSE_SCHEDULE" "COURSE_ID" "COURSE_ID")
                            ("COURSE_PREREQUISITES" "COURSE_ID"
                                                    "COURSE_ID")
                            ("COURSE_REGISTRATION" "COURSE_ID"
                                                   "COURSE_ID")
                            ("ACTIONS"    "COURSE_ID" "COURSE_ID"))
                        (   ("COURSE_ID"    7 TEXT
                                ("RMW" "R"))
                            ("COURSE_NAME" 50 TEXT
                                ("RMW" "R"))
                            ("FACULTY"     20 TEXT
                                ("RMW" "R"))
                            ("DEPARTMENT"  20 TEXT
                                ("RMW" "R"))
                            ("CREDITS"      2 TEXT
                                ("RMW" "R")))
                        ("RMWD" "R")
)
("COURSE_REGISTRATION"
                        (   ("COURSES" "COURSE_ID" "COURSE_ID")
                            ("COURSE_SCHEDULE" "COURSE_ID" "COURSE_ID")
                            ("COURSE_PREREQUISITES" "COURSE_ID"
                                                    "COURSE_ID")
                            ("ACTIONS"   "COURSE_ID"  "COURSE_ID")
                            ("STUDENTS" "STUDENT_ID" "STUDENT_ID")
                            ("STUDENT_HISTORY" "STUDENT_ID" "STUDENT_ID")
                            ("STUDENT_PROGRAM_HISTORY" "STUDENT_ID"
                                                       "STUDENT_ID"))
                        (   ("COURSE_ID"       7 TEXT
                                ("RMW" "R"))
                            ("COURSE_DATE"     6 DATE
                                ("RMW" "R"))
                            ("LOCATION_CODE"  10 TEXT
                                ("RMW" "R"))
                            ("STUDENT_ID"      6 TEXT
                                ("RMW" "R"))
                            ("DATE_ENROLLED"   6 DATE
                                ("RMW" "R"))
                            ("STUDENT_PROGRAM" 7 TEXT
                                ("RMW" "R"))
                            ("REGISTRATION_STATUS" 1 TEXT
                                ("RMW" "R")))
                        ("RMWD" "R")
)
```

```
("COURSE_SCHEDULE"
                        (   ("COURSES" "COURSE_ID" "COURSE_ID")
                            ("COURSE_REGISTRATION" "COURSE_ID"
                                                   "COURSE_ID")
                            ("COURSE_PREREQUISITES" "COURSE_ID"
                                                    "COURSE_ID")
                            ("ACTIONS"    "COURSE_ID" "COURSE_ID")
                            ("PROFESSOR" "PROFESSOR_ID"
                                         "PROFESSOR_ID")
                            ("CLASSROOM" "LOCATION_CODE"
                                         "LOCATION_CODE"))
                        (   ("COURSE_ID"           7 TEXT
                                ("RMW" "R"))
                            ("COURSE_DATE"         6 DATE
                                ("RMW" "R"))
                            ("LOCATION_CODE"      10 TEXT
                                ("RMW" "R"))
                            ("TIME"               13 TEXT
                                ("RMW" "R"))
                            ("LOCATION_CODE1"     10 TEXT
                                ("RMW" "R"))
                            ("TIME1"              13 TEXT
                                ("RMW" "R"))
                            ("LOCATION_CODE2"     10 TEXT
                                ("RMW" "R"))
                            ("TIME2"              13 TEXT
                                ("RMW" "R"))
                            ("LAB_LOCATION_CODE" 10 TEXT
                                ("RMW" "R"))
                            ("LAB_TIME"           13 TEXT
                                ("RMW" "R"))
                            ("COURSE_ACTIVE"       1 TEXT
                                ("RMW" "R"))
                            ("COURSE_STATUS"       1 TEXT
                                ("RMW" "R"))
                            ("PROFESSOR_ID"        5 TEXT
                                ("RMW" "R"))
                            ("MAXIMUM_STUDENTS"    2 NUMBER
                                ("RMW" "R"))
                            ("CURRENT_ENROLLMENT" 2 NUMBER
                                ("RMW" "R")))
                        ("RMWD" "R")
)

("COURSE_PREREQUISITES"
                        (   ("COURSES" "COURSE_ID" "COURSE_ID")
                            ("COURSE_SCHEDULE" "COURSE_ID" "COURSE_ID")
                            ("COURSE_REGISTRATION" "COURSE_ID"
                                                   "COURSE_ID")
                            ("ACTIONS"    "COURSE_ID"  "COURSE_ID"))
                        (   ("COURSE_ID"           7 TEXT
                                ("RMW" "R"))
                            ("PREREQUISITE1"       7 TEXT
                                ("RMW" "R"))
                            ("PREREQUISITE2"       7 TEXT
                                ("RMW" "R"))
                            ("PREREQUISITE3"       7 TEXT
                                ("RMW" "R"))
                            ("PREREQUISITE4"       7 TEXT
                                ("RMW" "R"))
```

```
                    ("PREREQUISITE1_OR"    7 TEXT
                        ("RMW" "R"))
                    ("PREREQUISITE2_OR"    7 TEXT
                        ("RMW" "R"))
                    ("PREREQUISITE3_OR"    7 TEXT
                        ("RMW" "R"))
                    ("PREREQUISITE4_OR"    7 TEXT
                        ("RMW" "R"))
                    ("COREQUISITE1"        7 TEXT
                        ("RMW" "R"))
                    ("COREQUISITE2"        7 TEXT
                        ("RMW" "R")))
                ("RMWD" "R")
)
("PROFESSOR"
                (   ("COURSE_SCHEDULE" "PROFESSOR_ID"
                                       "PROFESSOR_ID"))
                (   ("PROFESSOR_ID"           5 TEXT
                        ("RMW" "R"))
                    ("PROFESSOR_FIRST_NAME" 15 TEXT
                        ("RMW" "R"))
                    ("PROFESSOR_LAST_NAME"  20 TEXT
                        ("RMW" "R"))
                    ("PHONE"                12 TEXT
                        ("RMW" "R"))
                    ("OFFICE_LOCATION"      10 TEXT
                        ("RMW" "R"))
                    ("FACULTY"              20 TEXT
                        ("RMW" "R"))
                    ("DEPARTMENT"           20 TEXT
                        ("RMW" "R"))
                    ("DEGREES"              50 TEXT
                        ("RMW" "R"))
                    ("TITLE"                25 TEXT
                        ("RMW" "R")))
                ("RMWD" "R")
)

("CLASSROOM"
                (   ("COURSE_SCHEDULE" "LOCATION_CODE"
                                       "LOCATION_CODE"))
                (   ("LOCATION_CODE"         10 TEXT
                        ("RMW" "R"))
                    ("FULL_NAME_DESCRIPTION" 40 TEXT
                        ("RMW" "R"))
                    ("MAXIMUM_STUDENTS"       2 NUMBER
                        ("RMW" "R")))
                ("RMWD" "R")
)
```

```
("ACTIONS"
                        (   ("STUDENTS" "STUDENT_ID" "STUDENT_ID")
                            ("STUDENT_HISTORY" "STUDENT_ID" "STUDENT_ID")
                            ("STUDENT_PROGRAM_HISTORY" "STUDENT_ID"
                                                       "STUDENT_ID")
                            ("COURSES"  "COURSE_ID"  "COURSE_ID")
                            ("COURSE_SCHEDULE"  "COURSE_ID"
                                                "COURSE_ID")
                            ("COURSE_REGISTRATION" "COURSE_ID"
                                                   "COURSE_ID"))
                        (   ("FACULTY"          20 TEXT
                                ("RMW" "R"))
                            ("DEPARTMENT"       20 TEXT
                                ("RMW" "R"))
                            ("USER"             13 TEXT
                                ("RMW" "R"))
                            ("ACTION_CODE"       2 TEXT
                                ("RMW" "R"))
                            ("PRIORITY"          1 TEXT
                                ("RMW" "R"))
                            ("STUDENT_ID"        6 TEXT
                                ("RMW" "R"))
                            ("COURSE_ID"         7 TEXT
                                ("RMW" "R"))
                            ("COURSE_DATE"       6 DATE
                                ("RMW" "R"))
                            ("LOCATION_CODE"    10 TEXT
                                ("RMW" "R"))
                            ("ACTION_TEXT"      50 TEXT
                                ("RMW" "R")))
                        ("RMWD" "R")
    )
   )
)
```