ABSTRACT

Porting COSMOS Expert System
Form UNIX To DOS

by
Ching-Jenq Chiu

COSMOS is an object-oriented Knowledge Based System building Tools (KBSTs) to solve problem in engineering industry. COSMOS stands for C++ Object-oriented System Made for expert System development.

In order to provide more people those who don't have a Sun workstation to use this expert system, our task is porting COSMOS form UNIX to DOS.

Because the differents of workstation environment, the user interface and structure of original COSMOS no longer can be used, therefore we made some necessary change before we proting it to IBM Personal Computer.

In stead of X Window system®, we implemented ObjectWindows® runs on Microsoft Windows™. substitute AT&T C++ with Borland® C++, and because YACC is not a standard feature of DOS we consider the Window of Inference Engine Monitor as an independent object, create it by either *system* call or *makefile* at run time.

PORTING COSMOS EXPERT SYSTEM
FROM UNIX TO DOS

by
Ching-Jenq Chiu

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science
Department of Computer and Information Science
October, 1992

APPROVAL PAGE

Porting COSMOS Expert System
From UNIX to DOS

by
Ching-Jenq Chiu

Dr. David T. Wang, Thesis Adviser
Assistant Professor of Computer and Information Science, NJIT

Dr. Daniel Yuh Chao, Theais Co-adviser
Assistant Professor of Computer and Information Science, NJIT

Dr. Dao-Chuan Hung, Committee Member
Assistant Professor of Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author: Ching-Jenq Chiu

Degree: Master of Science in Computer and Information Science

Date: October, 1992

Date of Birth:

Place of Birth:

Undergraduate and Graduate Education:

- Master of Science in Computer and Information Science, New Jersey Institute of Technology, Newark, NJ, 1992

- Master of Science in Chemical Engineering, New Jersey Institute of Technology, Newark, NJ, 1990

- Bachelor of Science in Chemical Engineering, Tamkang University, Taipei, Taiwan, 1986

Major: Computer and Information Science

This thesis is dedicated to my parents,

Ching-Sung Chiu and Lung-Mey Chang

ACKNOWLEDGMENT

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 What Is KBESs

Knowledge-base is a collection of general facts, rules of the problem. The main purpose of KBES is to solve problem in engineering industry. The solution comes from skillful manipulation of large quantities of knowledge. It starts out with certain assumption and hypothesis and revises these assumptions and hypotheses until the solution is achieved.

KBES typically consist of the following three components (Figure 1.1)

1. **Knowledge-base** is a collection of general facts, rules of thumb and causal models of the behavior of the problem domain. A number of formalisms have been used to represent knowledge and the most widely used one is the *production system* model. In this formalism, the knowledge is encoded in the form of antecedent-consequent pairs or IF-THEN rules and uncertainty in the knowledge is represented by means of confidence factors. Other forms of representations commonly used are logic and frame-based schemes.

2. **Context** is a workspace for the solution constructed by the inference mechanism from the information provided by the user and the knowledge-base.

3. **Inference mechanism** is used to monitor the execution of the program by using the knowledge-base to modify the context. A number of problem solving strategies (control strategies) exist in current KBES.

1
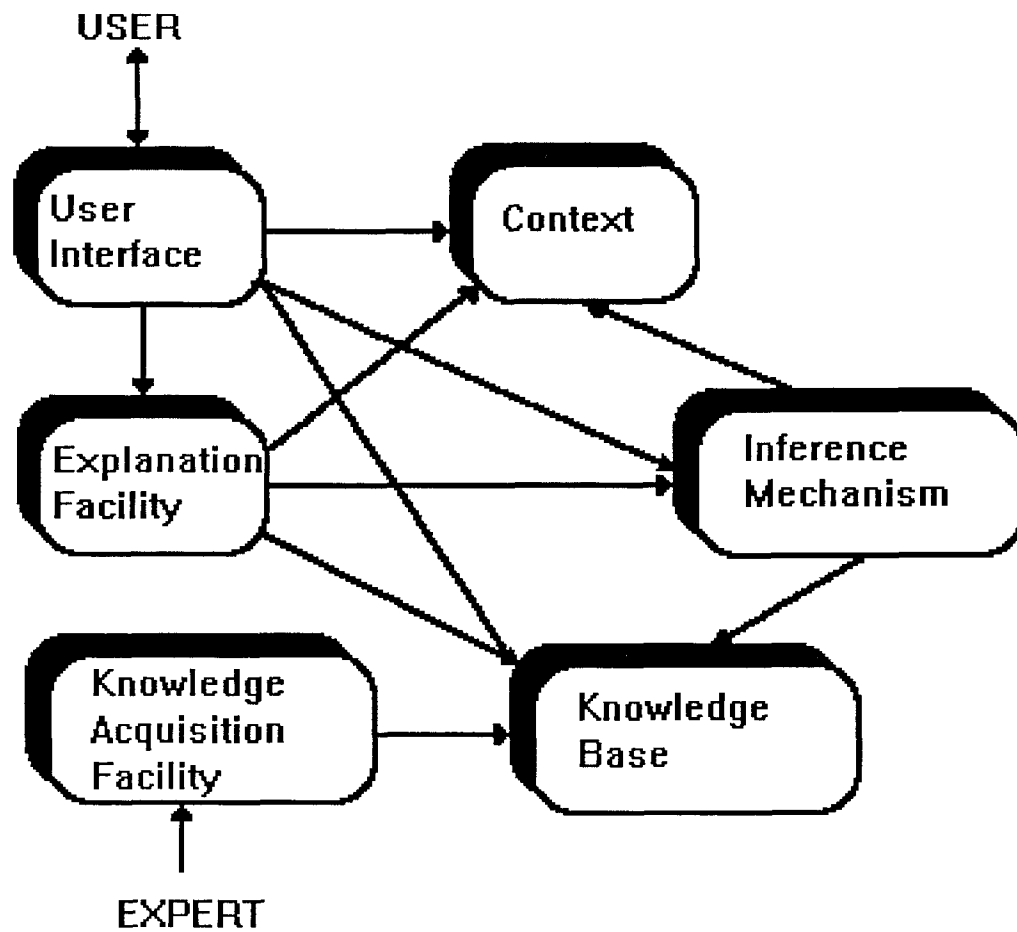
Figure 1.1 Schematic View of a Complete Knowledge-Based Expert

Several domain independent KBES shells have been developed for solving certain classes of problems. Programming using these shells involves encoding the knowledge of a particular domain, while the problem solving strategies are provided by the inference mechanism.

Apart from the components described above, there are three more modules which are desired in any expert system: a graceful *user interface*; an *explanation facility*; and a knowledge *acquisition facility*.

1.2 What Kind of Engineering Problems Can be Solved on This System?

Most of the engineering problems full on derivation-formation spectrum. In derivation type of problems, the solution comes from the identification of the solution path. And problem conditions are part of solution description. In derivation problems, we can follow the following process to solve the problem.

1. **Diagnosis**: The task is to involve reasons based on incomplete and inexact data or wrong sensors of the system.

2. **Interpretation**: The analysis is done only on complete and reliable data.

3. **Monitoring**: Signals are interpreted.

4. **Control**: Based on signal interpretation, the system is regulated.

5. **Education**: The users must have some knowledge to identify the problem and must respond to the various problems.

6. **Simulation**: When a problem is solved and we are satisfied with the results.

On the other hand, in the formation problems, the solution must satisfy as a whole. But, in KBES, we do not get an exact solution. By using inference mechanism, we can get the solution which provides the knowledge in the knowledge-base.

1. **Planning**: Some actions should be taken in order to use the resources and achieve some goals.

2. **Design**: Large problems can be broken down into some small pieces(problems). For example, top down design. And, small problems must interact properly.

1.3 Structure of COSMOS

COSMOS consists of the following modules: 1) User Interface; 2) Object Manager; 3) Rule-base/Parser; and 4) Inference Mechanism. These modules are briefly described below. Detailed descriptions of some of these modules are provided in the following sections.

User Interface. The User Interface module consists of the Expert System Development Tool (ESDT) and the Expert System End User Tool (ESEUT). ESDT is used by a knowledge engineer to input objects and rules. ESEUT is used by an end-user to run the knowledge-base expert system.

Object Manager. The Object Manager module is responsible for the maintenance of all classes and objects created at runtime, record keeping on the extension (all the instances) and intention (contents) of classes, access, retrieval and interaction functions at runtime on request from the user-interface and the inference engine, and persistence management of data and inference states across sessions.

Rule-base/Parser. The input to the Parser is the code generated (knowledge base) by the knowledge editor of ESDT. As its output, the Parser generates two data structures used by the Inference Mechanism. The first data structure is an inference network that is used by the backward chaining (BC) mechanism. The second data structure is an intermediate data structure, used by the RETE network building algorithm of the forward chaining (FC) mechanism of the inference engine of COSMOS to generate the RETE network.
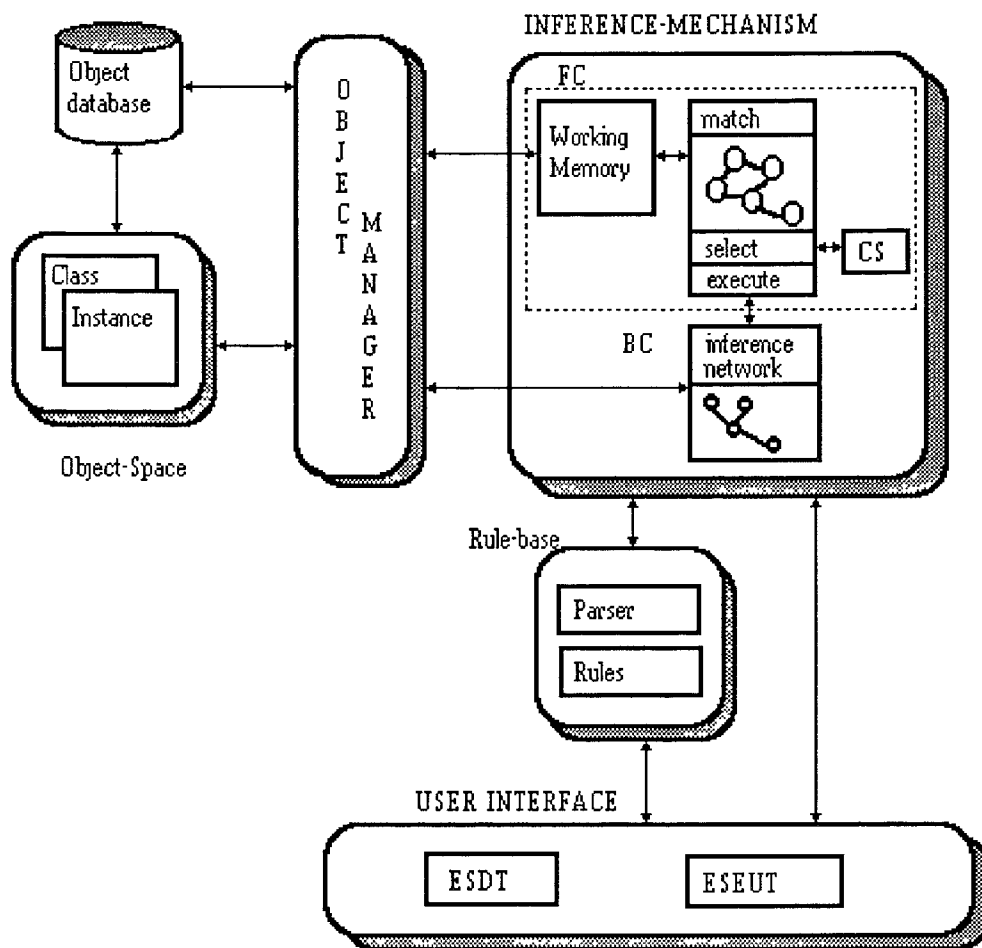
Figure 1.2 Structure of COSMOS

Inference Mechanism. The Inference Mechanism consists of two problem solving strategies: forward chain and backward chaining. The forward chaining strategy consists of a modified object-oriented RETE network. The backward chaining strategy utilizes an object-oriented inference network, and is based on the techniques used in the KAS system(1).

1.4 Our Goal

Our goal is porting COSMOS to IBM PC, extend C++ to support object evaluations, develop friendly user interfaces for C++ objects and rules, Provide problem solving support, make source code available so that parts of COSMOS can be integrated into engineering software, support links to external programs and run COSMOS on any PC supporting MS Windows 3.0.

CHAPTER 2

WHY WINDOWS 3.0?

With the advent of Microsoft Windows 3.0 (Figure 2.1). everything has changed. For the first time, users of DOS computers have a graphical user interface flexible and powerful enough to support applications of every kind and of every level of complexity and ambition. The release of Microsoft Windows 3.0 represents the best implementation of a graphical environment for PC users available anywhere. The first two version of Windows were suitable mostly for graphics applications and for rare programs like Microsoft Excel, which seemed more significant for the glimpse they gave of Windows' potential futures than they were for themselves. The third version of Windows is suitable for almost anything.

Over a million users rushed to take advantage of Windows 3.0 in the months after its release, and every major software vendor that doesn't already have a Windows-based product on the market is hurrying to fill the gap. Microsoft's public-relations blitz for Windows is only partly responsible for this stunning effect on the PC market. The real reason for Windows' meteoric success is the confluence of three major factors.

First, an enormous range of Windows applications is available now . Windows users don't have to wait years for the kinds of applications promised, but not yet delivered, for OS/2. With this major new release of the Windows Graphics User Interface (GUI), there is now precious little real difference between the software styles of the leading computers, and there fore, precious little reason to pay extra for it. This environment is bound to delight all sorts

of Windows users, form neophytes to power users, to programmers. Windows 3.0 features the same screen design used in OS/2 1.2. But since Windows 3.0 offers compatibility with current Windows programs and a 386 mode that runs multiple DOS sessions, it is an irresistible alternative for power-hungry users aching to break out of DOS's 640K limit.
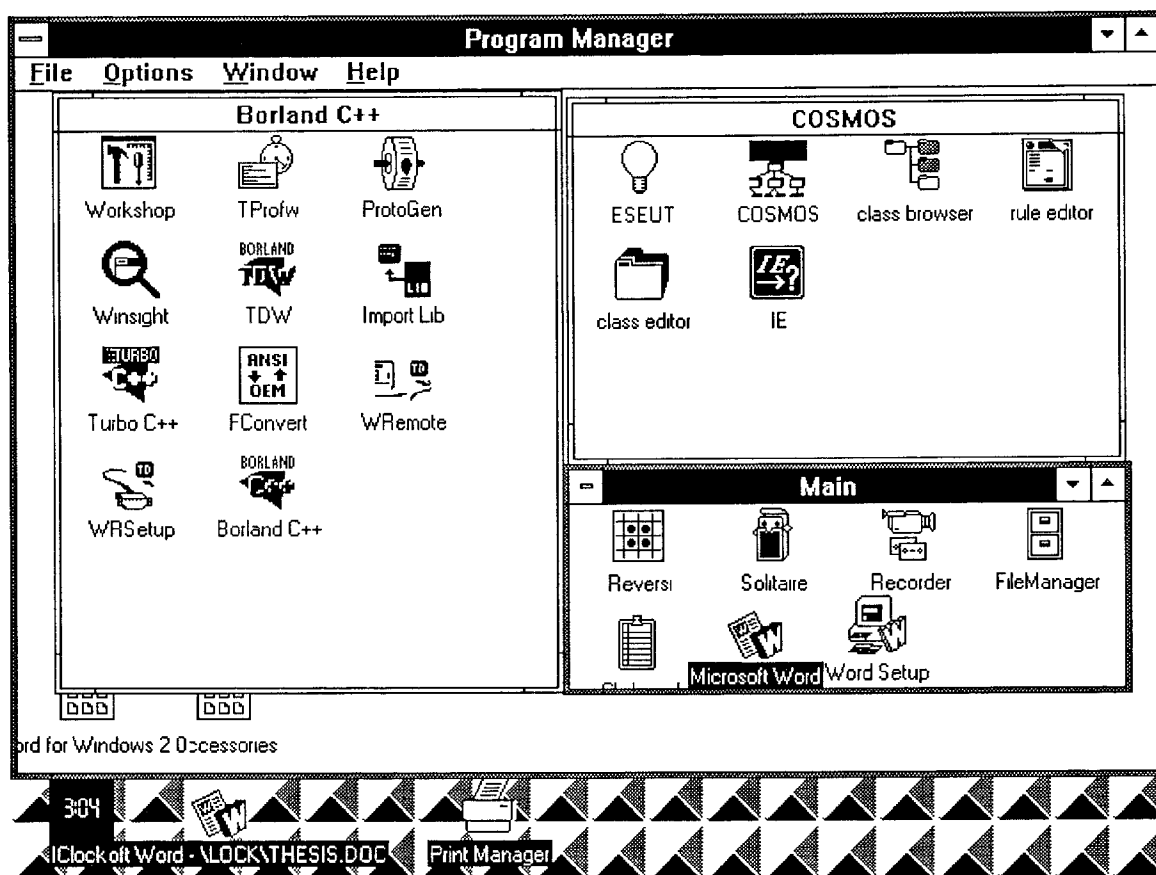
Figure 2.1 Graphic User Interface of Microsoft Window 3.0

Second, unlike the Macintosh, with its relatively small installed base and notoriously expensive hard ware, Windows runs on tens of millions of DOS-based machines. In fact, Windows can run on virtually any DOS-based machine

with a suitable graphics adapter and monitor. Adequate performance calls for added memory and a speedy processor, but Windows 3.0 made its appearance at an ideal historical moment when RAM is plentiful and cheap, and fast 386SX and 386-based machines are cheaper than ever.

Third, Window 3.0 is able to push any PC to its limits in ways that few other programs can. It exploits the powers of a 386 or 486 machine(with at least 2MB of RAM) by letting you multitask DOS sessions while you also run multiple Windows applications. Windows 3.0 not only lets Windows applications use all the memory in the system, but it multitasks so smoothly on a 386 or 486 that you can easily use it as a front end for all of your DOS applications. Even on a 286 machine Windows lets you multitask Windows applications and access up to 16MB of RAM. Although Windows' speed and abilities are constrained by an 8088-class computer, even those machines let Windows 3.0 use expanded memory and multitask all but a few especially demanding applications.

This upgrade has raised talk of trouble for Apple and NeXT, and it's not hard to see why. The clean design of the icon-based screen, the fast response time, and the ease with which most anyone can get up and running under Windows 3.0. puts everything in an entirely different perspective.

Any DOS application can become an icon in a Windows folder. Just double-click and run, quit an return to Windows. Forget about batch files and paths; you tell Windows about the path once, and be done with it.

With a DOS application running in Windows 386 mode, a quick Ctrl-Esc key sequence will switch you back to your Windows Program Manager and off to yet another DOS application. The same kind of program switching is possible on 286 and 8088 PCs, but at a speed that most people won't find tolerable.

There's a lot here for everyone, but it can be totally confusing without a basic understanding of the three distinct operating modes.

2.1 Three Windows Operating Modes

Real mode is the ground floor. It's really here for compatibility; it's the only mode that 8086s and 8080s can use, and it's the only mode that will run the Windows applications that are currently shipping. It looks better than Windows 2.01, since it's icon based and uses shading to accent screen objects. It falls short if you want to go beyond DOS's 640K barrier, though.

Standard mode is for 286s and 386s. new versions of Windows applications will let you use all of the available system memory, and that's going to provide a major performance boost to memory-constrained programs like PageMaker and Microsoft Excel. The Windows programs you now own won't run in this mode (unless you recently bought Microsoft's newest version of Excel or word for Windows), but software publishers are ready with upgrades for just about every application.

386 Enhanced mode pulls out all the stops. Going even beyond OS/2, it lets you multitask DOS applications. What can you do in this mode? Start a coffee-break-long-number-crunching session, and then switch to another DOS application. Now switch to Windows. Start as many separate sessions as you like--The numbers will keep crunching at a respectable rate.

Not even RAM is a limit, since Enhanced mode will page data to disk when it runs out of memory. You won't want to let it run out of RAM, though; you system will snap, crackle, and pop its way through commands so quickly that you'll be heading out to the store for extra RAM chips if it ever starts paging to disk.

This is the level of performance that has gotten developer like Lotus and Software Publishing to change their OS/2-based strategy to include Windows.

2.2 Object-Oriented Windows Design

This chapter concerns several different issues in object-oriented design for the Windows environment. There are tow basically different viewpoints represented here: design with respect to the user, and design with respect to the resources of the computer. As a reflection of this, the chapter is divided into roughly two different parts. The first part deals with object-oriented design, both generally and in the Windows environment. It does this from the point of view of the application, without reference to the issues of memory management on the computer. The second part takes up the latter issue, and, in particular, covers the topics of: 1) memory management in Windows 3.0, and 2) object-oriented memory management.

2.2.1 Designing for Users

The first principle of software design is that software must be designed to be used by someone, and therefore, its design must be conducted from the point of view of the people who will be the intended users. Although everyone will agree with this principle, it continues to be violated far more frequently than it is followed. It cannot be true that the principle is so seldom adhered to because of the sheer incompetence of software designers. The fact of the matter is that there is still very much to learn, and a surprising amount to take into consideration when designing for users. As all programmers know, a computer must be first loaded with data before its instructions are executed. A human being can begin to engage in a activity and then go out in search of the data that will be needed to carry out or complete the activity. Obviously, it is sometimes very important to a particular type of human work that it be able to

proceed in precisely this way. Software applications have to be designed with this sort of thing in view. Although these are very sweeping generalities, I have in mind some very specific examples.

Most readers will be familiar with popular paint programs, like the paintbrush application that comes as an accessory with Windows 3.0 an later. In these programs there will typically be a toolbox composed of icons that stand for different kinds of operation that a user might want to perform. The aspiring electronic artist feels at home with this to a certain degree. Just as a traditional painter often uses a paintbox with different colors that can be dipped into with a brush, the electronic artist can use the mouse to dip into different drawing tools. The graphics tools are verbs and the graphics lines and shapes are the nouns of a sentence that this kind of program speaks. Sometimes you must first select the noun before selecting the verb, but with the paintbox style, the typical case is the reverse: you first choose the verb and then create or add to the noun. On the whole, this can work well for creating things like drawings, because there is no set place where you have to work next. You can stay with one tool for a while, going from one area of a drawing to another and using it where it seems needed. As long as the final result is what you want, the order in which you do things doesn't really matter.

It is interesting to compare this with applications that involve sequential rather than spatial compositions, such as music compositions, animations, and computer programs. Here you are by no means as free to browse about at will within the work that's being created. It has to unfold as a continuous sequence, so there usually is a next place where you have to work. This next frame, next measure, or next line is the natural, built-in focal point. It is the noun that is primary over any verb. What's important is how the next noun behaves, not what tool you happen to be using. As a matter of fact, if elaborate tools are

attempted in this type of application, they can end up getting in the way to the extent that the result is unusable for many people. The reason is that you end up imposing a style of working on the user that is rather foreign to the type of work being done, and many people will refuse to accept that.

2.2.2 What's the Computer Really Doing?

The second principle of software design is that programs must, to the greatest degree possible, reflect the primary function for which the hardware is being used. In designing any artifact that is to be used repeatedly--so in designing computer programs--you also need to ask what primary function the machine is really performing for the user. As obvious as this sounds, it is overlooked surprisingly often. What is the main function that personal computers generally perform today? I suggest that it is this:

One of the most important roles of desktop computers is that of allowing a very large amount of information to be made visible rapidly at one time, in a way that can be easily modified as desired.

A key word here is 'visible'. Computers excel at making graphic, textual, and numeric information visible. If this is true, then it follows that the best designed software applications are those that allow this function to be performed the best.

It's the users who ultimately decide, but it's the designer's job to anticipate them as much as possible. If you design an application so that you allow the users to see as much or as little information at one time on the screen as they wish, and provide a wide variety of options for how that information is displayed, it's hard to see how you will be going wrong. As basic a principle as this is, it is surprising how many commercially available programs violate it.

2.2.3 Partitioning Procedures and Protocols

The first step in designing an object-oriented system is making lists of the important data items and procedures that the application will require. This is probably not something that's going to remain fixed throughout the course of the program's life, but the important thing is to get off to good start by developing a reasonable partitioning of some of the essential things the program will be dealing with. If the essentials are clarified and understood, the details can be added to this. Then the initial skeleton will have muscle connected to it. Once you have an idea of what the main classes will be for your application, then you will need to design a message protocol for it. A message protocol includes a plan of which messages will be understood by which objects, and how These messages will circulate through the program, and how descendant classes will utilize them.

The use of formal classes has an important role in the design of object-oriented systems. One of their most important features is providing functionality that is of the greatest generality without specifics that are useless for many applications. When an ancestor class has features that are undesirable, they have to be overridden by descendant classes, which means redefining them. This means additional labor and potential problems Ideally, a set of formal classes should be designed so that nothing ever has to be overridden. Though it's impossible to completely achieve this, it should be one of the goals of object-oriented design.

In an object-oriented system the typical situation is that there is more than one possible design solution to creating a system of objects to fulfill a given program requirement. Quit often there are actually several solutions. Which one is selected depends on the main goals and criteria that have to be solved. There are consequently at least two levels to object-oriented design: the

strategic level and the tactical. At the tactical level, design does not occur apart form coding. It is primarily the result of trying out the coding of an initial design that results in redesign and recording, and ultimately proves which design option works best. At the strategic level, the view is toward the perfect generic design where no line of code is wasted or redundant. Wherever code can be reused it is inherited rather than rewritten, both within an application, and across many similar programs.

2.2.4 Designing Classes

There are really several different viewpoints from which considerations can be made for designing classes. From a strictly formal viewpoint, you can consider all the variables that are nearly always used together with the same procedures and package them together into a class. Or you can look at the full scope of not just a single application, but all similar ones, and design a class that will play a desired role within a functional hierarchy. This latter approach is particularly clear, for example, when designing a formal or abstract class. It's often advantageous to create a new specialized subclass object for a special job, so that because of he design of the new object, the coding becomes tremendously simplified.

For the advanced designer of object-oriented systems, it is not only the current application that is kept in view when a system of classes are being designed, but many other applications that can or may be written as well. The very nature of object-oriented systems is that a host of different application programs can be assembled from a growing set of autonomously functioning classes. To make good on this inherent quality of the technology the designer has to look to the desirable applications that can be built while designing the current application that will be built. Ideally, there will be a kernel of classes

from the application that will form the basis for many applications in the future as well as for the current one.

2.2.5 Multiple Application Instances

It is often desirable to design applications in such a way that there can be multiple instances present at the same time. It is when an application consists of a number of cooperating objects that send messages to one another that special design considerations are needed. The reason is that to send messages, each object has to know the name of the instance of the object in its particular copy of the application. For multiple instances of the whole multiobject application to work properly, the names of corresponding objects have to be different, and each object within an application must know the name of those objects in the same copy of the application as itself. From this, it's clear that the instance names cannot appear in the code directly. The solution is simple and elegant. First, some mechanism must be adopted for generating unique names for the instances. Second, each object has instance variables that hold the names of the objects to which it must send messages. Messages access these instance variables to determine the object to which they will be sent. Here we see yet another important technique in which instance variables play the key role.

2.2.6 Object-Oriented Design for GUIs

The following sections describe some very prevalent issues and problems all graphic user interface (GUI) developers must face and attempt to show how object-oriented techniques can be a major help in addressing them. An important question to ask before getting absorbed in various details is whether object-oriented design makes any difference to end user or whether it is a refinement that only affects developers. Initially, it may seem that the benefits

of object-oriented systems are only for the programmers who do the development work on applications. However, it is becoming clear that if the object-oriented methodology is carried to its consistent logical conclusion, then it is also quite relevant to what the end user will be seeing in the way of applications. In the future, there will be a different type of application program emerging from which users will also derive great benefit.

In commercial programming, another important issue to consider in designing programs and user interfaces is portability to different computer platforms. For example, if a programmer wants to port a program being written under Windows to the Macintosh or X Windows, then there are some general thing to consider to make this easier. For example, there is mouse control. The Macintosh has only a one button mouse. Therefore in developing a class for mouse control it is important to remember that it must be possible to produce a version of one's program that can work with only a single mouse button.

2.2.7 Open Activity Chains

One thing that will help in designing object-oriented software is to have something to fasten onto that can allow the designer to realistically anticipate what users will expect out of program. This is always going to be a very large challenge, but I would like to suggest some concepts that may often be of help. A useful concept is that of open activity chains.

2.2.8 Factoring Command Methods

As discussed in the previous chapter, one of the enemies of complex menu systems in GUIs is the proliferation of very long sequences of case statements that can be very difficult to debug. The strategy of object-oriented programming (OOP) is based on building functioning packages that will always work, even though any number of new ones are added to the system. This implies that

good object-oriented design requires avoiding a single large command method for processing all menus whenever possible. As was seen earlier, by using the technique of factoring the processing of windows command messages into a number of separate command methods each of which can work independently, and also in cooperation with other menus, programs compile faster, are simpler to debug, use up less memory resources, and are considerably easier to update.

2.3 Windows Memory Management Design

In a multitasking system like Windows, There are often several different applications that will be requesting memory during the same Windows session. Memory management facilities are available to make sure that all applications retain access to the memory needed in as efficient manner as possible. Developers of Windows applications must make use of these facilities to ensure that their application is using the least amount of memory necessary at any given time. Windows has a total of forty different memory management functions to allow the application developer to address both the issues and the memory. In the following paragraphs, I will present a concise overview of the way memory management is handled under MS-Windows.

In Windows, memory can be allocated in two ways: from the global heap and from the local heap. The global heap comprises memory that is available to all applications. A local heap provides memory for just a single application. In Windows, memory is allocated in blocks and is relocatable. It can be moved around and even discarded. Movable memory blocks do not have fixed addresses. At any time, Windows can move them to a different location. Movable memory blocks allow free memory to be consolidated into the largest possible blocks. If an allocated block of memory lies between two free areas, the allocated block can be moved so that the two free areas are combined into one

block of consecutive addresses. Discardable memory is memory that can be freed and reallocated. Naturally, this involves destroying any data that may have been contained in it. When a block of memory is allocated in Windows, a handle to it is returned to the application requesting it. This handle is not an address but rather a means of retrieving whatever the memory block's current address may be.

Accessing memory blocks involves locking the memory handle. While the handle is locked Windows cannot move or discard it. An address pointer for the beginning of the block is returned, and the application is given reliable access to the memory. Unlocking the memory handle is up to the application. This means that to provide for the most efficient memory management, developers should adopt the rule that when an application is finished using a block of memory, its handle should be unlocked as soon as possible.

Most Windows applications use mixed memory models. The recommended method is with small code segments of about 4K each so that Windows can easily move these segments about in memory. Applications can allocate memory from either the global heap or from local heaps. The main consideration as to which heap will be used is usually the amount of memory needed. Larger memory blocks are usually allocated from the global heap where it is possible to allocate single blocks larger than 64K. The main Windows function for managing the global heap are Global Alloc. GlobalLock, GlobalUnlock, GlobalCompact, and GlobalFree. An application's local heap is the free memory in its data segment that can be allocated for various purposes. The local heap is not automatically assigned by Windows but must be requested with the HEAPSIZE statement. Normally, the local heap cannot be larger than 64K, which is the size of an application's data segment.

2.3.1 Types of Data Storage

In Windows, seven different types of data storage may be used:(Table 2.1)

Windows applications that are conversant with advanced memory management issues have to be able to respond to the WM_COMPACTING message.

Table 2.1 Types of Data Storage in Windows

Static data	Used for static variables such as those defined by the static and extern keywords in C
Automatic data	Used for variables already on the stack when functions are called.
Local dynamic data	Any data in memory areas allocated using LocalAlloc.
Global dynamic data	Any data in memory areas allocated using GlobalAlloc.
Window extra bytes	Used for additional storage that may be requested for a window class.
Class extra bytes	Used for additional storage that may be allocated after the WNDCLASS structure.
Resources	Memory used for resources in an application's .EXE file that have been loaded into memory.

2.3.2 Discardable Memory

In Windows, creating applications with discardable memory must be done explicitly. To create a discardable memory block, both the GMEM_MOVEABLE and the GMEM_DISCARDABLE options to the Global Alloc function must be used. For example, in C the declaration would be:

hMem = GlobalAlloc(GMEM_MOVEABLE — GMEM_discardable, 4096L);

Windows will discard discardable memory when it gets allocation requests that need to be met. Windows determines which discardable blocks to actually discard based on a least recently used algorithm. Using the GlobalDiscard

Windows function discards the data stored in the block but retains its handle. The GlobalReAlloc function makes nondiscardable memory blocks discardable and vice versa.

2.4 Advance Memory Management

In the following two paragraphs two advanced Windows memory configurations will be described, the standard and 386 Enhanced Mode configurations.

2.4.1 Standard Mode

The standard mode Windows memory configuration is the default on 286 computers with at least one megabyte(1M) of memory and 386 with more than one but less than 2 megabytes (2M). Windows uses the protected mode of the 80286 and 80386 processors in the standard memory mode. When Windows runs in this mode, the global heap is usually made up of three distinct blocks of memory. The first block is usually the 640K DOS segment. The second block is in extended memory, which is allocated using the extended memory device driver, but then is accessed directly. Finally, the third block in standard mode is the high memory area (HMA), which is only available if no other software has been loaded into high memory before launching Windows. The Windows global heap is formed by linking these three blocks of memory together. Discardable memory segments are allocated from the top of the heap, fixed segments from the bottom, and movable code and data from just above the fixed segments.

2.4.2 386 Enhanced Mode

with 386 computers with 2M or more of extended memory, Windows can be run in the 386 Enhanced Mode. In this mode Windows provides a virtual memory scheme that utilizes both extended memory and hard disk space to allow

memory spaces as large as 64M. In the enhanced mode, the Windows global heap is composed of a large single virtual address space. The size of this space available. Because the structure of this virtual memory space is composed of a single large block, it resembles that of basic memory configuration, and its layout is actually strongly analogous to it, though of course it includes a far larger address space.

The release of Microsoft Windows 3.0 represents the best implementation of a graphical environment for PC users available anywhere.

Windows 3.0 not only lets Windows applications use all the memory in the system, but it multitasks so smoothly on a 386 or 486 that you can easily use it as a front end for all of your DOS applications.

Windows 3.0 features the same screen design used in OS/2 1.2. But since Windows 3.0 offers compatibility with current Windows programs and a 386 mode that runs multiple DOS sessions, it is an irresistible alternative for power-hungry users aching to break out of DOS's 640K limit.

This upgrade has raised talk of trouble for Apple and NeXT, and it's not hard to see why. The clean design of the icon-based screen, the fast response time, and the ease with which most anyone can get up and running under Windows 3.0. puts everything in an entirely different perspective.

Any DOS application can become an icon in a Windows folder. Just double-click and run, quit an return to Windows. Forget about batch files and paths; you tell Windows about the path once, and be done with it.

With a DOS application running in Windows 386 mode, a quick Ctrl-Esc key sequence will switch you back to your Windows Program Manager and off to yet another DOS application. The same kind of program switching is possible on 286 and 8088 PCs, but at a speed that most people won't find tolerable.

There's a lot here for everyone, but it can be totally confusing without a basic understanding of the three distinct operating modes.

With the advent of Microsoft Windows 3.0, everything has changed.

For the first time, users of DOS computers have a graphical user interface flexible and powerful enough to support applications of every kind and of every level of complexity and ambition. The first two version of Windows were suitable mostly for graphics applications and for rare programs like Microsoft Excel, which seemed more significant for the glimpse they gave of Windows' potential futures than they were for themselves. The third version of Windows is suitable for almost anything.

Over a million users rushed to take advantage of Windows 3.0 in the months after its release, and every major software vendor that doesn't already have a Windows-based product on the market is hurrying to fill the gap. Microsoft's public-relations blitz for Windows is only partly responsible for this stunning effect on the PC market. The real reason for Windows' meteoric success is the confluence of three major factors.

First, an enormous range of Windows applications is available now. Windows users don't have to wait years for the kinds of applications promised, but not yet delivered, for OS/2.

Second, unlike the Macintosh, with its relatively small installed base and notoriously expensive hard ware, Windows runs on tens of millions of DOS-based machines. In fact, Windows can run on virtually any DOS-based machine with a suitable graphics adapter and monitor. Adequate performance calls for added memory and a speedy processor, but Windows 3.0 made its appearance at an ideal historical moment when RAM is plentiful and cheap, and fast 386SX and 386-based machines are cheaper than ever.

Third, Window 3.0 is able to push any PC to its limits in ways that few other programs can. It exploits the powers of a 386 or 486 machine(with at least 2MB of RAM) by letting you multitask DOS sessions while you also run multiple Windows applications. Even on a 286 machine Windows lets you multitask Windows applications and access up to 16MB of RAM. Although Windows' speed and abilities are constrained by an 8088-class computer, even those machines let Windows 3.0 use expanded memory and multitask all but a few especially demanding applications.

With this major new release of the Windows Graphics User Interface (GUI), there is now precious little real difference between the software styles of the leading computers, and there fore, precious little reason to pay extra for it. This environment is bound to delight all sorts of Windows users, form neophytes to power users, to programmers.

CHAPTER 3

OBJECTWINDOWS®

ObjectWindows is an language-independent application framework that works with Actor, Turbo Pascal for Window, Turbo C++ and Borland C++. The beauty of using an application frame work that supports a variety of languages is that you can do development in the language of your choice and later translate your program to other languages as required. ObjectWindows programs written in Borland C++ and Turbo Pascal for Windows match one another almost exactly on a line-for-line basis. Now with Borland's integrated development environments operating directly under Microsoft Windows, Actor and Smalltalk no longer provide the only development platforms that eliminate switching back and forth between Windows and DOS when constructing and testing programs. In addition, ObjectWindows takes much of the grief out of developing Windows programs. With Object Windows, it's just as easy to write programs for Windows as for DOS.

3.1 The ObjectWindows Hierarchy

ObjectWindows can also be describe as a library of hierarchical classes that allows the programmer to take full advantage of the object-oriented feature of inheritance. The data member and member function that are in ObjectWindows class allows the programmer of the Windows applications program with ease.(Figure 3.1)
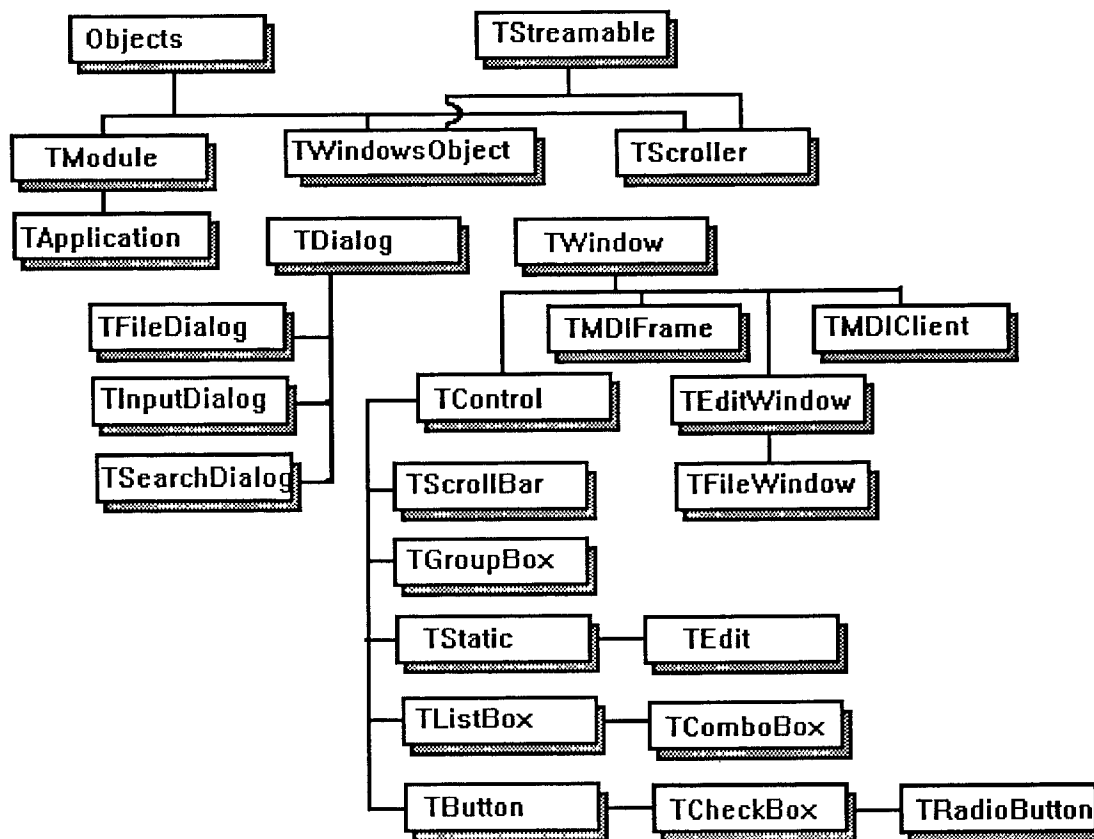
Figure 3.1 The ObjectWindows Hierarchy

3.1.1 Window Objects

Window objects are interface objects that are associated with window elements. A Window object is an interface object and has a corresponding interface element. In order to create a window, you must first define the object and then create the interface element.

3.1.2 Dialog Objects

Dialog objects are special purpose child windows that are used as user interfaces input-related tasks. Dialog Boxes are defined in resource files and are referced in an application by the dialog box ID specified in the resource file.

3.1.3 Control Objects

Control objects are special windows that serve as user interface elements. Control objects include buttons, scroll bars, list boxes, check boxes, group boxes, edit controls.

When object-oriented programming was introduced, it was not thought that it would become such a big thing. The way things are going today, it can be easily predicted that by the next decade, we would see nothing more but object-oriented programming. And it would be because of applications like **Object Windows** that every programmer would be able to write his own applications with great ease in the wonderful and mythical environment of Windows, where Multi-tasking and Multi-programming exists which have always been the dream of programmers.

3.2 Object-Oriented Design for MS-Windows

One of the original object-oriented design models is the Model-View-Controller architecture, or M-V-C for short. A key aspect of this architecture is the idea that modularity should prevail between the way information is organized and stored and the way it is accessed and displayed. In other words, the data for an application is stored in an appropriate set of objects, and this data model is independent of the of the way the information will be ultimately displayed to the user. A number of different objects can be designed for viewing the same data in alternate ways. A standard protocol is usually devised for showing a given set of data in these alternate viewing models. Although this idea may seem obvious or rudimentary, it is by no means the case that mainstream programming has operated in this way. In fact, the contrary has been the rule rather than the exception.

It will be useful to consider a concrete example. In the fields of document processing and desktop publishing, it has become popular to utilize special files known as style sheets. Although the applications that utilize them were not necessarily developed using object-oriented programming, there is a similarity between the use of style sheets and the M-V-C model we are discussing. The similarity lies in the idea of storing the text data in one file, and the style format by which it will be viewed and printed in another. In this way, alternate viewing styles can be quickly and easily assigned to the same text data. The advantages of doing things in this way have been clearly recognized in the industry, and there are many document processing applications that currently use this approach.

The modularity of the M-V-C architecture is, of course, far more fundamental and far-reaching than that of style sheets. The design embraces not just disk files, but application memory as well. And it invites levels of modularity that can have a very noticeable effect on how applications are used.

At this point, there have been three main object-oriented Windows programming systems: Actor, Views, and CommonView. There are striking differences in the approaches taken by all three as is clearly evident in their respective class hierarchies. The C++ Views approach is an explicit implementation of the M-V-C architecture. Views uses the Notifier class, a direct descendant of the root class Object, as the key to its control layer mechanism. In any given application there is always one and only one instance of the Notifier class, which is always globally assigned the name notifier. The main event loop of applications is incorporated in the notifier object. All applications also have a top view object that is an instance of a subclass of AppView. All messages between the Views object system and Windows are conducted through the notifier, which waits for the start message before it

begins processing events. The formal Window class provides methods to respond to messages of every event type.

The Views Window and View classes are similar to the Actor WindowsObject and Window classes, respectively. View has an instance variable called model that is used to store the ID of objects that provide the data model for and application. Views uses three classes to implement menu systems: Menu, PopupMenu, and MenuItem.

Views differs from Actor in forming separate classes for different types of button, such as PushButton, CheckBox, RadioButton, and TriState. Another major difference is that the TextEditor class is a descendant of Control rather than View, as it would be if a design like that used in Actor were followed. Another major design difference is that the Views Dialog class is a descendant of View. If a design like Actor were followed, Dialog would have instead been a direct descendant of Window and a peer of View, Menu bars in Views are created by making an instance of the Menu class and attaching instances of the PopupMenu class to it.

The difference in the design approaches used by Actor and Views can be quite readily seen in the way these two object-oriented Windows systems implement text editing Views provides a class called TextEditor, which is a descendant of Control and a direct and be aligned and justified. TextEditor objects are designed to work as part of a composite of subjects also from the String, Stream, and FileStream classes. This composite still has to be integrated into a larger composite that constitutes the main window, particularly if file operations are to be added. The ControlView class is designed to allow instances of the ControlWindow classes to be integrated as child views of a complete application. This handles list box controls, among other things.

3.3 Hierarchical Menus

Hierarchical or cascading menus offer a ready means of packing a lot of functionality into a single menu bar while at the same time avoiding a cluttered screen. Yet their use can certainly be overdone. Using a hierarchical menu resembles introducing the equivalent of an additional menu bar of options. Screen space is needed for those options that pop out to the side of the main menu rectangle. For my taste, two hierarchical menus is the most that should be attempted in a single window. Although hierarchical menus can be created either dynamically or statically, it turns out that the static resource script approach to creating them is so simple, that this very often is going to be the way to go right from the outset.

3.4 Handling Large Complex Menu Systems

There are important considerations with at least two different design levels for arriving at the optimal user interface design. First, there is the layout of the menu bar. As mentioned earlier, given the familiar layout, it is still quite possible to create user interfaces that can range from excellent to almost unusable. There is as yet not instant method for producing the optimal designs in user interfaces or any other area of OOP.

CHAPTER 4

ABOUT THE EXPERT SYSTEM

The structure of an expert system resembles a conventional software program, as shown in Table 4.1. The major components of an expert system are knowledge base, inference engine, user interface mechanism (including explanation facility), and data, but data is not critical; the major components of conventional programs are data (or data base), code, interpreter/compiler, and sparse user interface mechanism, but the interpreter/compiler is not obvious to the user. Expert systems are capable of symbol processing, inferencing, and explaining because of the inference engine and the knowledge base; conventional programs are generally strong in numerical processing and algorithms because of programs and extensive data. In general, the user interface mechanism of the expert system is more extensive than that of conventional programs; semi- or full-natural languages may be employed to enhance user-friendliness.

In terms of terminology used, expert systems can be considered as an advanced form of programming. The terminology of expert systems can be mapped up on a one-to-one basis to that of software programs as shown in Table 4.1. For example, a knowledge base of an expert system that contains rules (likely IF-THEN rules) and facts matches the program (code) of a software program. However, a knowledge base does not correspond to a data base. A knowledge base is executable, but a data base is not. A data base can only be queried and updated.

Table 4.1 One-to-One Correspondence in Expert System and Software Program Technologies

Expert Systems	Software Programs
Knowledge base	Program
Inference engine	Interpreter
Expert system tool/shell	Programming Language
Knowledge engineers	Software engineers/programmer analysts

Like an interpreter that evaluates a program in the source code and executes the statements, the inference engine takes the statements in a knowledge base and executes them because it contains search control and reasoning mechanisms. Expert system tools/shells that are used to build expert systems are very high-level programming languages with many unconventional conveniences such as explanation facility and trace. Even though they are different from conventional software programming languages (e.g., Fortran or C), they are programming languages in nature.

Often expert systems can be employed as intelligent front ends to facilitate the use of complex software packages because an expert system may contain the heuristic knowledge of its experienced user and contain user-friendly natural language query and explanation facilities.

4.1. The Ultimate Goal of Expert System Technology

The term "expert systems" suggests mimicry of human experts in a particular field. The ultimate goal of expert system technology is to equal the performance of human experts. In everyday life, we encounter many people whom we consider experts; they can be physicians, lawyers, engineering designers, or security brokers. They all share an important common characteristic: They must

make accurate decisions in environments that are fraught with uncertainty and risks, and they possess superior ability to do so as a result of training, experience, and professional practice. We generally visualize experts as having the following characteristics: quick, confident judgment made under pressure, a reassuring manner, and an ability to receive unusual or rare events.

The nature of the expertise we encounter in our day-to-day association with experts can be summarized as follows:

- Proficiency at arriving at quick, accurate solutions to problems
- Proficiency at explaining the results to the layperson
- Proficiency at learning from experience
- Proficient at restructuring knowledge to fit the environment
- Capability to make exceptions Awareness of their limitations

4.1.1 Solving the Problem

A solution to the problem is usually the most direct and important benefit we expect from experts who are capable of comprehending a particular type of problem that few other people would even attempt to solve. The focus here is the ability to solve a complex problem effectively and quickly-- the performance of an expert. This criterion not only helps us to narrow the selection of experts, but also leads us to take greater care in choosing the expert system task we intend to model. On the sole basis of performance, a general agreement as to who qualifies as an expert is easier to establish for some fields (e.g., medicine, geological exploration) than for others (e.g., stock trading, weather forecasting). The selection of experts and tasks is discussed later.

4.1.2 Explaining the Results

An expert can usually explain the results of his or her problem-solving to nonexperts in terms that they understand. An expert can respond intelligently

to inquiries concerning the reasons for the results, the logical processes derived, and the implications of the results. Normally, the inquirers can question the expert in lay terminology and receive modified outcome that is closer to the given condition perceived by them. The ability to explain the results thus enhances an expert's performance in solving the right problem.

4.1.3 Learning from Experience

Experts must learn from their own experience, as well as the experience of others; experts must further enhance this experience through various means (e.g., daily practice or trade journals) and must Update their knowledge bases and modify reasoning processes. Experts who do not keep up with their field quickly become obsolete, particularly in the present era of rapid technological development. This ability to maintain a high level of expertise also improves the expert's performance in problem-solving.

4.1.4 Restructuring Knowledge to Fit the Environment

Experts' effectiveness in problem-solving differs from that of a novice in that experts are proficient at restructuring and reorganizing knowledge to fit the environment:

- By subdividing their knowledge base and using the critical portion of knowledge so that the search time for the right answer can be reduced.

- By putting the problem in a different perspective using various portions of their knowledge

- By applying knowledge to the problem at different levels or angles

For example, When solving the problem of a malfunctioning computer system, the expert first uses a top-down rough organization chart of the system's main components to locate the most likely failed parts. The expert then applies the knowledge regarding the schematic of the part to identify the

problematic electronic device in the part. The expert's speed and effectiveness in performing these tasks determine the proficiency in solving the problem.

4.1.5 Making Exceptions

Great writers often deviate from grammatical rules. Experts can perceive rare and unusual events or occurrences and make exceptions from their usual modes of judgment. For example, an experienced stock analyst will discard a regular analytical pattern regarding a company's stock prices in response to news of the crash of the company jet in which its CEO was a passenger. An expert has this general ability to make exceptions to match previously unanticipated and unusual events.

4.1.6 Awareness of Limitations

Experts can assess the relevance of their expertise to a given problem and determine whether or not the problem is within their sphere of expertise. They also know when to refer inquiries to other experts. Experts became less proficient at solving problems when they reach the limits of their expertise. They can bow out gracefully with "qualified" answers; they know when to acknowledge their limits.

4.1.7 State-of-the-Art Technology Development

The "state-of-the-art" or "the land of accepted wisdom" expert system technology contains two aspects of experts' expertise: an ability to solve problems quickly and accurately, and an ability to explain the results in terms understandable to a nonexpert. The first generation of expert system technology focused on problem-solving performance. Such systems include DENDRAL [12] and MACSYMA [14]. The second generation systems began to explore

characteristics such as explanation (the Digitalis Advisor) and even a small portion of learning from experience, knowledge acquisition (TEIRESLAS [5]).

Some emerging expert systems may incorporate additional aspects of an expert's expertise such as learning. These emerging systems are largely confined to academic laboratories (e.g., MIT's computer diagnosis, Stanford's hardware diagnosis, UCLA's automated testing for electronic design). This new technology is based on representation and use of knowledge about structure, function, and design of physical systems. It extends the frontier of computer cognizance into reasoning beyond the rules in knowledge base. For more discussion of the development path of expert system technology, see Davis [5]

4.2 The Three Stages of Expert System Technology Growth

Human experts require time to mature from apprenticeship to mastership; expert systems can be improved through use to attain a certain degree of maturity. The three stages are

- Assistant: the expert system performs as an assistant to the user--an assistant system

- Colleague: the expert system performs as a colleague to the user--a colleague system

- Advisor: the expert system performs as an advisor toe the user--an advisor system.

The differences in these systems are based on the level of expertise: an assistant system performs at a level much inferior to that of a human expert; a colleague system performs at a level slightly inferior to that of a human expert; an advisor performs at a level equal to that of human experts.

The delineation of these stages is relatively subjective. In general, the large systems like XCON that employ more than 1,000 rules and sophisticated structure can perform at the level of a human specialist; they perform with high

confidence as an advisor to human specialists. On the other hand, very small systems like PUFF (which has 64 rules) may perform as assistants to human experts or specialists to undertake a repetitive and tedious decision making task under their constant supervision and evaluation. The medium-sized systems (between the large and the small) can be subjectively described as a colleague system. Most commercial expert systems fall within this category.

4.3 The Basic Structure of Expert Systems

The three basic elements of an expert system are a knowledge base, an inference engine, and a user or person-machine interface. The knowledge base contains facts and heuristics, the inference engine performs interpretation (reasoning) and control of search for solutions, and the user interface provides the user with semi- or full-natural language, or ultimately pictures and verbal responses. We will discuss (1) the knowledge Base, (2) the inference engine, (3) the person-machine interface, (4) uncertainty of knowledge, and (5) less frequently used features.

4.3 Knowledge Base

Let us assume that you, as a software engineer, are asked by the sales department of a Car & Automobile Repair (CAR) company to build a simple expert system, named COSMOS, to assist sales representatives in determining the operational and economic potential of implementing CARs onto various malfunctions that can happen to a car. The sample set of possible malfunctions is:

1. The battery is dead
2. The ignition system is bas
3. The fuel system is faulty
4. The car is out of gas

5. The starter is bad

6. The engine is flooded

The symptoms for the car malfunctions are:

1. The key is in the ignition and is turned on

2. The fuel gauge shows empty

3. The engine is turning over

4. The headlights are dim or dead

5. The carburetor smells like gasoline

6. There are sparks at teh spark plugs

The software engineer can use three types of knowledge to build expert systems: (1) rules of thumb, (2) facts and relations among components, and (3) assertions and questions (that sales representatives can direct to customers). To represent these types of knowledge in the knowledge base, methods are used:

- Rules to represent rules of thumb

- Frames to represent facts and relations

- Logic to represent assertions and questions

Rules are conditional sentences; they are expressed in the following form:

> IF (premise) FACT 1, FACT 2, ...
> THEN (conclusion) FACT 9, FACT 10, ...

These groups of rules in response to operational and economic potential of CARs in motors were extracted from various experienced sales engineers, and are written in an Object-Orient programming language, C++. for example, rule 1a of "dead battery" can be explained as follows:

> IF (Problem is in Starting System and Headlights Dim or Dead)
> THEN (Battery is Dead)

Other rules can be explained in the same manner.

Frames A frame contains the hierarchies of objects (components), the attributes of objects that can be assigned, inherited from other frame, or computed through procedures or other computer programs The attributes are filled in the "slots" of a frame.

Table 4.2 Frames of car

```
class car:
        public root
{
  public:
    char* engine_turning_over ;
    char* ignition_key ;
    char* starting_system ;
    char* headlights ;
    char*  init_problem ;
    char* spark_plug_spark ;
    char* Fuel_gauge_reading ;
    char* carburetor_gas ;
    char* problem ;
    char* car_make ;
    car(char* instance_name, char* c_name)
      {
        strcpy(name, instance_name);
        strcpy(classname, c_name);
        generate_askable_slot_list();

      engine_turning_over = NULL;
      ignition_key = NULL;
      starting_system = NULL;
      headlights = NULL;
      init_problem = NULL;
      spark_plug_spark = NULL;
      Fuel_gauge_reading = NULL;
      carburetor_gas = NULL;
      problem = NULL;
      car_make = NULL;
      }
    car(){ }
```

```
    void make_instance()
      {
        ptr = new car(name,classname);
      }
    ~car()
      {
        delete engine_turning_over;
        delete ignition_key;
        delete starting_system;
        delete headlights;
        delete init_problem;
        delete spark_plug_spark;
        delete Fuel_gauge_reading;
        delete carburetor_gas;
        delete problem;
        delete car_make;
      }
    void printout() ;
    int put_value(args* params, char* slot_value, char* index = 0);
    char* get_value(args* params, char* slot_type, char* index = 0);
    int set_low(args* params, char* lowvalue, char* index = 0);
    int set_high(args* params, char* highvalue, char* index = 0);
    int write_object(FILE* fout);
    root* read_object(FILE* fin);
    void invoke_method(char* method_name);
    char* get_low (args* params, char* slot_type, char* index = 0);
    char* get_high (args* params, char* slot_type, char* index = 0);
    char* ask_slot_string(char*, char* index = 0);
    void generate_askable_slot_list();
    slot_value_list* ask_slot_values(char*, char* index = 0);
};
```

Table 4.2 shows a sample frame, CAR, that contains the attributes of this special type of CAR. The frame shows that the CAR is part of root, many of its features are inherited from Frame root, it is installed to an induction motor, and it has two slots--car and mechanist. Each slot represents a special attribute

of the frame, its type, and value. The value can be computed by procedure or inherited from another frame (e.g., CAR). Rules can also be contained in the value of a slot as shown in the last slot of Table 4.2. Like an array, a frame or an attribute slot of the frame can be called upon by rules or logic expressions.

The main difference between frame and rule is that under a frame we can represent default values, references (pointers) to other frames, rules, procedures for which values can be specified, and terms and conditions of any action that needs to be taken. The ability to represent procedures and terms and conditions with values or actions that are required is useful in connecting the many components of information in that expert system. This feature enables us to represent a flow of instruction in the sequence of activities. This type of representation is sometimes called *procedural representation*, in contrast to *descriptive representation*, Which is simply an assertion of a fact or an event. In frames, we can integrate the two representations effectively.

Logic expressions consist of predicates and values to assess facts of the real world. A predicate is a statement concerning an object such as

kind-of (HONDA, CAR)

It may be interpreted as an HONDA, a kind of CAR. The object may be either a constant or a variable that may change over time. A predicate may have one or more arguments that are the objects it describes.

4.3.2 Inference Engine

Once the knowledge base is completed, it needs to be executed by a reasoning mechanism and search control to solve problems. The most common reasoning method in expert systems is the application of the following simple logic rule (also called *modus ponens*):

IF A is true, THEN B is true in a statement of "IF A, THEN B."

The implication of this simple rule is that

IF B is not true, THEN A is not true in the same statement.

Another implication of the simple logic rule is that

| Given: | IF A, THEN B and |
	IF B, THEN C
Conclusion:	IF A, THEN C.

In other words, IF A is true, THEN we can conclude C is also true.

These three simple reasoning principles are used to examine rules, facts, and relations in expert systems to solve problems. However, to minimize the reasoning time, search control methods are used to determine where to start the reasoning process and to choose which rule to examine next when several rules are conflicting at the same point. The two main methods of search are forward and backward chaining. The two methods of chaining may be combined in use in an expert system for maximum efficiency of search control.

4.3.3 Forward Chaining

When the rule interpreter is forward chaining, if premise clauses match the situation, then the conclusion clauses are asserted. For examples, in Rule dead battery of Append B, if the real situation matches the premise (that is, the problem is in starting system and headlights dim or dead), the problem of CARs will be battery is dead. Once the rule is used or "fired," it will not be used again in the same search; however, the fact concluded as the result of that rule's firing will be added to the knowledge base. This cycle of finding a

matched rule, firing it, and adding the conclusion to the knowledge base will be repeated until no more matched rules can be found.

In the case of COSMOS, the inference built into the C++ code is a simplified forward chaining that traverses the rules only once and in a predetermined order (i.e., if one rule is to affect another, it must appear before that rule). Part of the reason for this simplification is that a cumulative uncertainty probability function (to be discussed in certainty factor) is used in calculating the feasibility of an CAR's being installed to a given motor. Rules are used once so that their probability will be accounted for only once.

4.3.4 Backward Chaining

A backward chaining mechanism attempts to prove the hypothesis from facts. If the current goal is to determine the fact in the conclusion (hypothesis), then it is necessary to determine whether the premises match the situation. Since COSMOS does into use backward chaining, a different example is used to demonstrate.

Rule One:

IF you lose the key and the gas tank is empty

THEN the car is not running

Rule Two:

IF the car is not running and you have no cash

THEN you are going to be late

Fact One: you lost the key

Fact Two: the gas tank is empty

For instance, if we want to prove the hypothesis that "you are going to be late," given the facts and rules in the knowledge base (Facts 1 and 2, Rules 1 and 2), a backward chaining must be applied to determine whether the premises

match the situation. Rule 2, which contains the conclusion, would be fired first to determine whether the premises match the fact. Because the knowledge base does not contain the facts in the premises of Rule 2, "the car is not running" and "you have no cash,""the car is not running" becomes the first subgoal. Rule 1 will then be fired to assert whether the premises "you lost the key" and "the gas tank is empty" match the facts. Because the facts (Facts 1 and 2) in the knowledge base match the premise of Rule 1, the subhypothesis is proven. However, the system still has to prove that "you have no cash," which is not contained in the knowledge base and cannot be asserted through rules since no rule is related to it. The system will then ask the user, "IS IT TRUE THAT: you have no cash?" if the answer is "yes," then the second subgoal is also satisfied and the original hypothesis is there fore proven, concluding that "you are going to be late."

4.3.5 Human-Machine Interface

The Human-machine interface mechanism produces dialogue between the computer and the user. The current expert system may be equipped with "menus" or natural language to facilitate its use, and an explanation module to allow the user to challenge and examine the reasoning process underlying the systems answers.

Menu refer to groups of simplified instructional statements that appear on the computer screen and can be selected by the user by pushing designated buttons on a mouse or designated keys on the keyboard. The user does not have to type instructions. A semi or full-natural language interface is more sophisticated than a menu interface; it allows computer systems to accept inputs and produce outputs in a language closer to a conventional language such as English. Several expert systems incorporate primitive forms of natural language

in their user interface to facilitate knowledge base development. Explanation modules generate output statements of expert systems in language that can be understood by noncomputer professional users.

The three aspects of user interface mechanism that affect its efficiency are

- user modes: client, tutor, and pupil
- Interface purpose: testing, applications, and modifications
- User groups: domain expert, knowledge engineer, and general public

User mode is defined as how users are going to use the expert system when they interface with it. Three different user modes are associated with an expert system in comparison with the single mode of obtaining answers to problems characteristic of general computer application. Users can act as clients to obtain answers to problems from the expert system, as tutors to increase or improve the knowledge of the expert system, and as pupils to harvest the knowledge of the expert system for increasing their skill in a specified subject. Interface purpose is defined as the objective of the user's interacting with the expert system. Interface with the expert system can occur for testing the expert system before it is completely refined, for applying it to the real world situation for consultation on problems, and for modifying it when the experts find the answers are invalid or insufficient. Users are classified into three groups: domain experts, knowledge engineers, and general public. Domain experts are specialists in a given application field who assist in building the expert system initially, test it, and then apply it to solve problems when it is completely robust. Knowledge engineers are computer programmers who obtain knowledge from experts to develop expert systems and then refine the system; they frequently apply the system to real world problems. The general public does not develop or refine the system, but benefits using expert systems. The effectiveness of

human-machine interface for an expert system can be determined once the three aspects are identified.

User interface is the weakest but most critical element of expert systems because it determines how well the systems will be accepted by the end users. Major research efforts have been undertaken in natural language interface, voice recognition, and voice synthesis to make expert systems more user-friendly.

CHAPTER 5

FORWARD-CHAINING IN COSMOS

The forward chainer is based upon a variant of the RETE algorithm, originally introduced by Charles Forgy[8] .

Two features make this implementation more efficient than OPS5:

1. An object-oriented design, which makes it easy to modify and understand the RETE network. This design should be suitable for using on a parallel architecture.

2. A new conflict-set support, which allows very fast modifications of the RETE network.

5.1 The RETE algorithm

5.1.1 Production Systems and the RETE Philosophy

RETE algorithm has been designed to perform faster pattern-matching in large production systems. Usually, a production system is defined by a set of rules together with a collection of current assertions, or facts consisting the working-memory. The rules usually consist of a Left-Hand-Side (LHS) or condition part, and a Right-Hand-Side (RHS) or action part. Inferences are achieved by repeating match-select-execute cycles on the working-memory.

Basically, a cycle consists in:

1. **matching** data elements against LHS of rules. The set of rules which LHS has been matched form the *conflict-set*.

2. **selecting** one of the rules in the conflict-set.

3. **executing** the RHS of the rule.

Figure 5.1 illustrates this scheme.
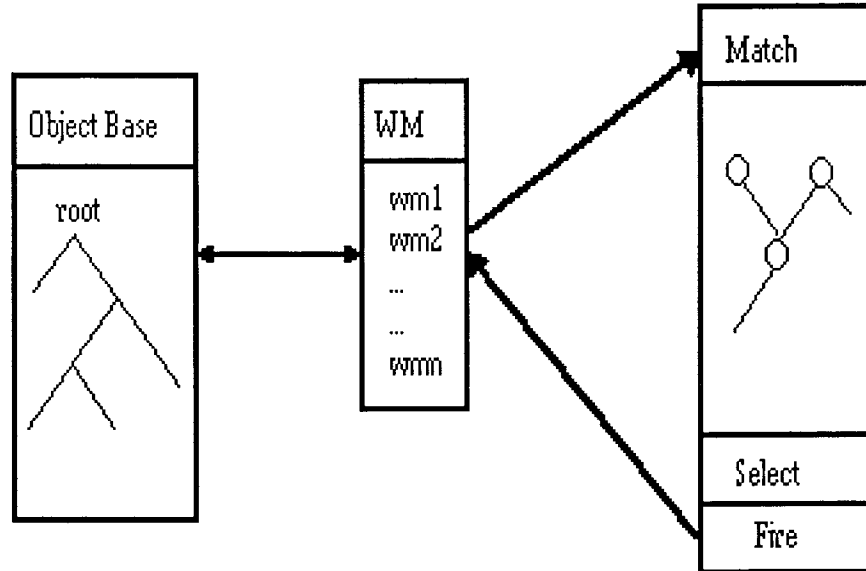
Figure 5.1 Objects Access in the Forward-chainer

RETE exploits two characteristics of such systems[2] .

1. While a data base may contain many facts, the percentage of data elements added or modified during a single cycle tends to be low.

2. Rules tend to share LHS conditions. Therefore, rules compiled into a network can share information.

From these two considerations follows the principle of the RETE algorithm. The idea is to manage a network - embedding rules and data elements - *incrementally* updated through cycles of inference. According to point 1, this updating should be fast. According to point 2, the network should be a good encoding of the rule-base, since redundancies are eliminated.

5.1.2 Representation of the RETE Network

To illustrate the RETE network, consider the following rule:

$$\textbf{if } (CE_1, ..., CE_n) \textbf{ then } A_1, ..., A_p$$

The LHS has the condition elements CE_i and the RHS has the actions A_i. The RETE network representing this rule is shown in Figure 5.2.
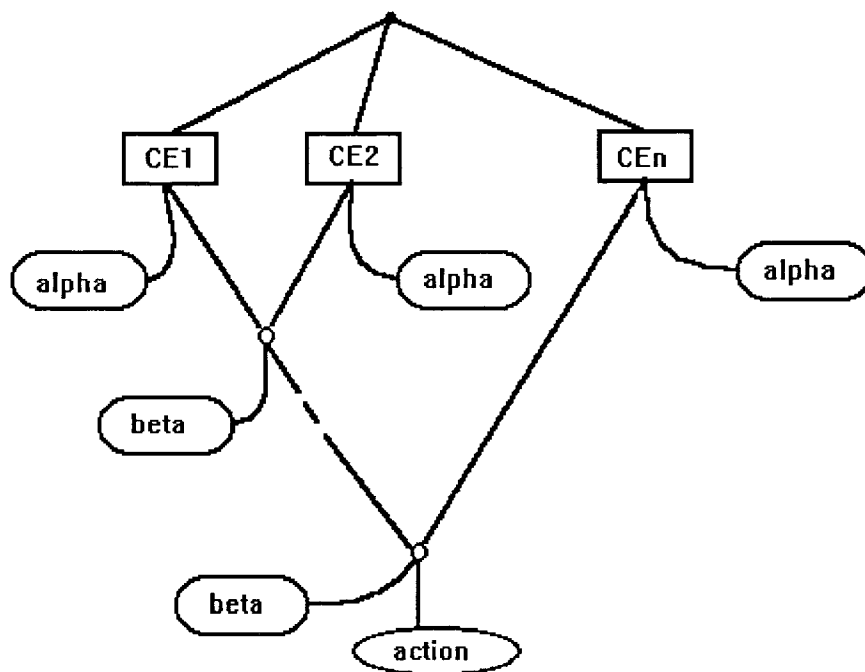
Figure 5.2 RETE Network for one Rule

When a new fact is added to the working-memory, a *token2* is created and sent to the network. When a token traverses the network, it is matched against CE_i. Tokens which satisfy condition-elements are stored in alpha-memories. Templates of alpha-memories are joined together by two-inputs nodes that test for consistent variables bindings. The joined tokens are stored in beta-memory and eventually reach the action-node. The last event results in a change in the conflict-set.

5.1.3 Improvements on RETE

Two directions have already been explored to improve the RETE algorithm:

- *Reduce the time and space complexity of the algorithm.* Miranker has introduced the TREAT algorithm which seems to be better than RETE in both time and space, according to the benchmarks reported in [14]. Briefly, TREAT does not use beta memories but rather sophisticated alpha-memories. This allows a reduction in space (size of the beta-memories) and in time, through a faster updating of the conflict-set.

- *Improve the generation of the network.* From a same rule-base, one can generate many RETE networks. These networks may not be equivalent during execution time. Bridgeland and Lafferty [2] have discussed a method for determining which of these networks is optimal. It is not clear, yet, if this search is worthwhile or not.

We introduce a third option which has not been explored yet: an object-oriented representation of the network. Miranker points out the difficulties encountered in porting RETE to a parallel environment. An object-oriented approach, we believe, should reduce these difficulties. Furthermore, we can work in a full object-oriented framework where the data-base is filled with classes and instances of classes, and the inference mechanism reduces to a set of nodes exchanging messages.

- As we shall see, it will also lead to a reduced cost for updating the conflict-set.

- In the following sections, we describe a representation of the RETE network in terms of classes and objects. When describing the different classes, we will skip some of their attributes, for the sake of clarity.

5.2 Basic Components

In this section we introduce the basic classes needed for inferencing, namely: the Inference engine (class **Ie**), a working-memory (class **WorkingMemory**), the RETE network (class **ReteNet**), and the conflict-set (class **ConflictSet**). Figure 5.3 illustrates the relation between these classes. Typically, there is one and only one instance of the above classes during run-time. But, if needed, several instantiations of these objects could be used simultaneously (each one carrying a specific knowledge, for example).
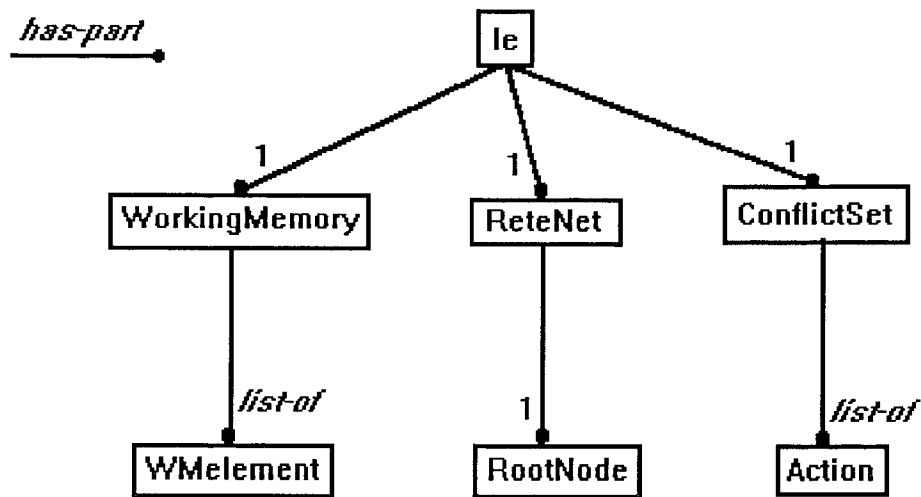
Figure 5.3 Has-part Relations Between Basic Components

5.2.1 Functional Description

(a)   The instance of **Ie**, named **theInferenceEngine**, monitors the whole inferencing mechanism, and is the interface between the forward-chainer and the rest of COSMOS. The only way one can access the forward-

chainer - to run it, load instances, browse the working-memory, and so on...- is to send a message to theInferenceEngine.

(b)   The instance of **WorkingMemory** consists basically of a list of **WMelements** (wme). It also has a counter which is a time reference. This counter is incremented every the **WorkingMemory** is accessed, and is used mainly by the conflict-set to determine the regency of an object. The **WorkingMemory** is needed for the following reasons:

- It ensures that objects and Inference engine are fully decoupled.

- We can use only the portion of the object base which is needed for inferencing. This is especially important for large data-bases.

- A working-memory element is more than a pointer to an object. Information regarding inferencing (last update, location(s) in the network, influence in the conflict-set) is stored in the wme. This information is needed only by the inference engine and is local to this process.

(c)   An instance of **ReteNet** is created every time the parser reads a new rule file. The whole RETE architecture is built, whose top-node is an instance of **RootNode**. This node is attached to the **ReteNet** which, in turn, is attached to **theInferenceEngine**. Further details are provided in Section 5.5.

(d)   The **ConflictSet** instance contains all the actions (RHS of rules plus bindings) which can be fired at a given cycle of inference. Several methods for conflict-resolution are discussed in Section 5.6.

5.2.2 Detailed Description

The declaration of **Ie** class is shown in Table 5.1. The methods are explained below:

- *reset()* sets the cycle_time to 0, and calls the reset() method of the **WorkingMemory**.

- *set_strategy()* allows the user to define the current strategy used by the **ConflictSet**.

- *load_wme()* loads a list of wmes into the **WorkingMemory**.

Table 5.1 Ie Class

class Ie

Private:
WorkingMemory *wmem; //pointer to the Working-Memory
ReteNet *rete; // point to the Rete Network
int cycle_time; // cycle counter
char *strategy; // name of the strategy
public:
ConflictSet *cs; //pointer to the Conflict-Set

Ie(); // Constructor
// method for end user:
void reset(); // init method
void set_strategy(char *); // changes strategy
void load_wme(root *); // loads a list of wme
void load1_wme(root *); // loads only one wme
void remove_wme(WMelement *); // remove a wme
void run(); // starts inferencing
// methods for bc
void record_change_object(root *);
void record_use_rule(char *);
// other methods
char *describe();
void attach_rete(ReteNet *r); // set the pointer to Rete

- *remove_wme()* removes a wme from the **WorkingMemory**. The **Rete** network is updated accordingly.

- *run()* begins inferencing. All the new wmes recently loaded in the **WorkingMemory** are propagated in the network. Rules present in the **ConflictSet** are fired, and a new cycle begins.

- *record_change_object()* is called by the Backward-chainer every time an object is modified. When backward-chaining ends, the forward-chainer is able to propagate these changes in the Rete Network to maintain the consistency.

- *record_use_rule()* is called by the Backward-chainer every time a rule is used. This rule will not be fired again when updating of the Rete network occurs.

The declaration of **WorkingMemory** class is shown in Table 5.2

Table 5.2 Working Memory Class

class WorkingMemory : public ObjectB

private:
Tlist(WMelement lwme); // list of wme
int access_time; // time reference

public:
void del_wme(WMelement *);
void add_wme(WMelement *);
char *describe();
void reset();
WorkingMemory(); //constructor

The methods are explained below:

- *del_wme()* deletes a wme from the Rete network. The algorithm used is explained in section 5.6.4. Then the wme is removed from `lwme`.

- *add_wme()* just pushes a wme on `lwme`. `access_time` is incremented.

- *reset()* sets the `access_time` to 0, then calls *del_wme* method for every wme in `lwme`. At the end of the procedure, the Rete network is empty.

The declaration of **WMelement** class is shown in Table 5.3

Table 5.3 WMelement Class

class WMelement : public ObjectB
friend WorkingMemory;
friend Ie;
friend ConflictSet;
private:
int cycle_time; // last cycle when the wme was modified
int load_time; // last load in the WorkingMemory
Tlist(Destroy_element) Idel; // map into the Rete network
public: root *proot; // pointer to the actual object
void keep_track(Token *, Tlist(Token));
void keep_track_cs(Token *, Action *);
char describe();
// Constructors WMelement (root *o);
WMelement (root *o, int t);

The methods are explained below:

- *keep_track()* allows a wme to record its location in the network at one stage of processing. Its successive locations are kept in the map `ldel`.

- *keep_track_cs()* same as above to record a location in the **ConflictSet**.

The declaration of **ReteNet** Class is shown in Table 5.4

Table 5.4 ReteNet Class

class ReteNet
private: RootNode *rootnode;

public:
ReteNet (RootNode *r); // Constructor
int process_token(Token *); // starts inferencing
char *describe();

One method is explained below:

process_token() takes a Token and sends it to the RootNode of the Rete

network.

The declaration of ConflictSet class is shown in Table 5.5

Table 5.5 ConflictSet Class

class ConflictSet friend ActionNode; friend DestroyInConflictSet;

private:
int strategy; // current strategy
Tlist(Action) llaction; // list of current instantiations of
rules

public:
Action *select();
int fire();
void set_strategy(int s);
ConflictSet();

The methods are explained below:

- *select()* uses the current strategy to choose among the rules kept in

  `llaction`.

- *fire()* first selects the action. Then takes the selected action and

  executes its RHS.

- *set_strategy()* allows a private access to the slot `strategy`.

5.3 Symbolic Information

5.3.1 Tokens

5.3.1.1 Functional Description

The **Token** class establishes the link between the working-memory and the network. Indeed, at a given cycle of inference, all current matchings, bindings are stored in the network. When a change occurs in the working-memory (a new fact is deduced by a rule, or added by the user), this information must be propagated in the network. This is the basic principle of incremental updating of RETE, outlined in Section 5.2. What is actually propagated in the network is an instance of **Token**.

Definition 1 A token is a triplet <*lw*, *lb*, *lc*>, where *lw* is a list of wme's, *lb* is a list of Bindings, and *lc* is a list of Checks.

Bindings are created after pattern-matching, Checks are special kinds of tests which could not be performed (for some variables were unknown). They are discussed in Section 5.5.2. Figure 5.4 illustrates the Token class.
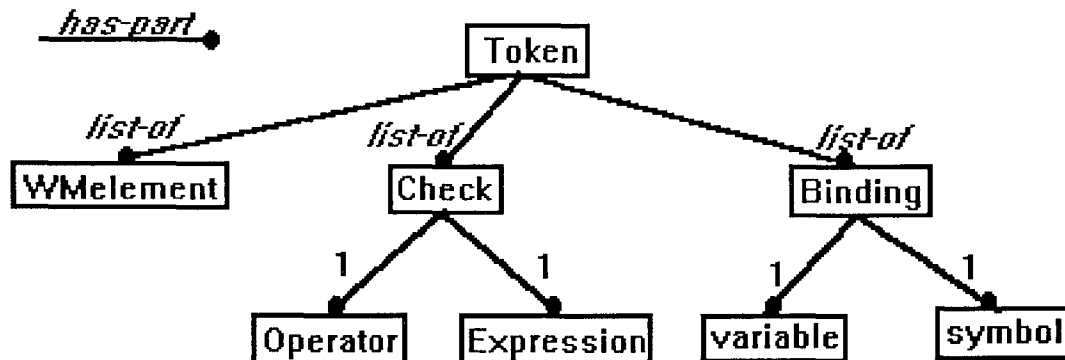
Figure 5.4 Token Class

5.3.1.2 Detailed Description

The declaration of **Token** class is shown in Table 5.6.

The methods are explained below:

- *duplicate()* generates a copy of the **Token** itself.

- *add_binding()* tries to add a **Binding** to the current binding-list lbind. If the **Binding** is not compatible with an existing one, returns **FAIL**.

- *print_bindings()* is used mainly for debugging purposes.

- *unify()* tries to merge two **Tokens** into a new one. The operation is successful if and only if all **Bindings** are compatible.

- *del_tok()* destructor for the **Token** class.

Table 5.6 Token Class

class Token : public ObjectB
public: Tlist(WMelement) lwme; Tlist(Binding) lbind; Tlist(Check) lcheck;

Token *duplicate();
int add_binding(Binding *);
char *describe();
void print_bindings();
Token *unify(Token *);
void del_tok();
// constructors
Token();
Token(WMelement *w);

5.3.2 Symbols and Expressions

5.3.2.1 Functional Description

What inferencing does is actually to deduce values of objects' slots, and update these objects. These values are either known constants, retrieved from other slots, or computed (basic arithmetic for example). During inferencing, every value handled by the forward-chainer is stored in a Symbol or an Expression.

- A **Symbol** instance contains information regarding the type of the symbol, and its external form (i.e. a printable name). In this sense, it is quite similar to a LISP symbol.

<div align="center">

Symbol types:

- string
- variable
- integer
- float
- root object
- expression
- WMelement

</div>
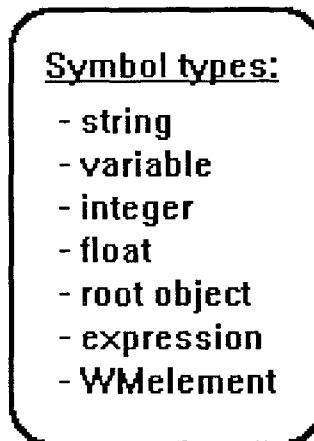
Figure 5.5 Symbol Types

During processing, symbols can be bound to variables which are special type of symbols. Note that they are not stored in variables, and thus a variable may be bound to several symbols. The combination of a variable name and a symbol is an instance of Binding.

- An **Expression** is a special class whose instances are able to evaluate themselves. Typically, an instance of expression represents a value

which needs some computation to be determined fully. Some example of extrusions are given below:

$$(3.14 * 6)^2$$

$$(\$pi * \$radius)^2$$

In the above examples, $pi represents a variable whose name is pi.

Attributes of an expression are sexpr: the string which represents the external form, var_list: a list of variables contained in the expression , and prog: a program which can be executed to compute the value (provided that all variables are bound). This framework is powerful for as soon as an instance of Expression is created, an instance of Prog is created after parsing. Then, one can evaluate the expression with different bindings without parsing it again.
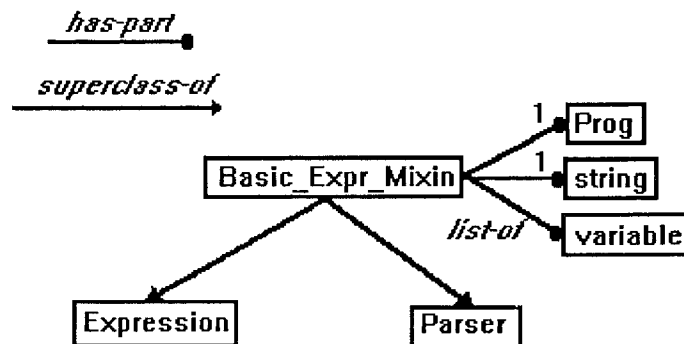
Figure 5.6 Expression Hierarchy

Figure 5.6 shows that both classes Expression and Parser inherit from the same attributes. A Parser instance is able to translate a string into a Prog, collecting the variables which appear in the string. All Expression instances use an instance of the same Parser to evaluate themselves.

5.3.2.2 Detailed Description

The declaration of **Symbol** class is shown in Table 5.7.

The declaration of **Binding** class is shown in Table 5.8

The methods are explained below:

- *compatible()* compares two Bindings. Returns SUCCESS if they are compatible, Fail otherwise.

- *equal()* returns SUCCESS if two Bindings are equal, FAIL otherwise.

Table 5.7 Symbol Class

class Symbol : public ObjectB

private:
char str[50]; // string description of the Symbol
public:
char type;
union {
int i;
float f;
char *s, *v;
Expression *e;
root *p;
WMelement *w;} value;

char *describe();
void fill(char , char *);
// constructors
Symbol();
Symbol(char , char *0);
Symbol(char , WMelement *);

Table 5.8 Binding Class

class Binding : public ObjectB
friend Expression;
friend Stack;
private:
char *variable;
Symbol *symbol;
public:
char *describe();
int compatible(Binding *);
int equal(Binding *);
binding(char *, Symbol *); // constructor

The declaration of **Basic_Expr_Mixin** class is shown in Table 5.9. This Mixin is designed for the only purpose of inheritance.

Table 5.9 Basic_Expr_Mixin Class

class Basic_Expr_Mixin
protected:
char *sexpr; // string representing the Expression
Prog *prog; // the program
Tlist(Symbol) var_list; // list of variables
Basic_Expr_Mixin(); // Constructor

The declaration of **Expression** class is shown in Table 5.10.

Table 5.10 Expression class

class Expression : private Basic_Expr_Mixin
public: void parse();
void append(Expression *);
void print();
Prog* set_prog();
Tlist(Symbol) get_var()
int is_evaluable(Tlist(Binding));
Symbol *eval(Tlist(Binding));
// constructors
Expression(char *);
Expression(char *, int);

The methods are explained below:

- *parse()* takes a string and parses it. The result is an executable program stored in **prog**. The parser recognizes arithmetic expressions and strings.

- *append()* takes two **Expressions** and merges them. The **prog** and **var_list** slots are concatenated.

- *print()* outputs a string representing an Expression.

- *get_prog()* and *get_var()* provide private access to prog and var_list.

- *is_evaluable()* checks if the program in prog is executable, given a list of Bindings.

- *eval()* executes prog.

5.4 Network Hierarchy

In this section, we deal with the design of the RETE network in terms of classes.

5.4.1 Functional Description
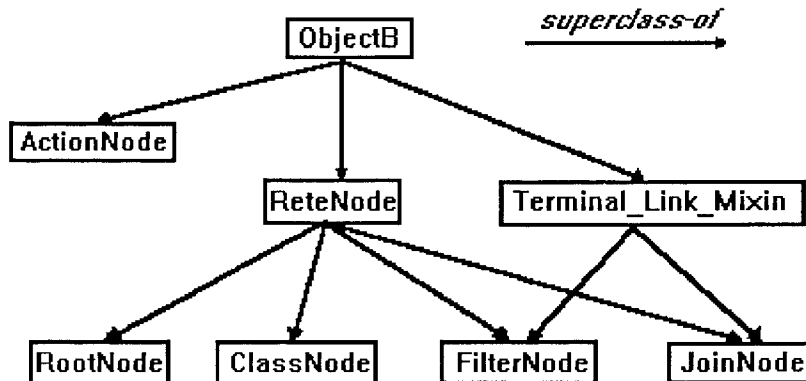
Figure 5.7 Hierarchy of the RETE Classes

Figure 5.7 shows the hierarchy. The top class is ObjectB, which contains methods for objects to describe themselves (a kind of vanilla flavor).

- **ReteNode** is the superclass for all the nodes in the RETE network (ActionNode is a special cases). It contains general attributes of nodes: a list of successors, a method to distribute forward a token, a method to process a token (each node redefines this virtual method according to its own function), and a method to link a node to another (used mainly while building the network).

- **Terminal_Link_Mixin** is a superclass of both FilterNode and JoinNode. It allows two instances of these classes to establish a link with an ActionNode (terminal node).

Before describing the different node classes, we first sketch how instances of nodes are organized inside an actual network. Figure 5.8 illustrates this architecture.
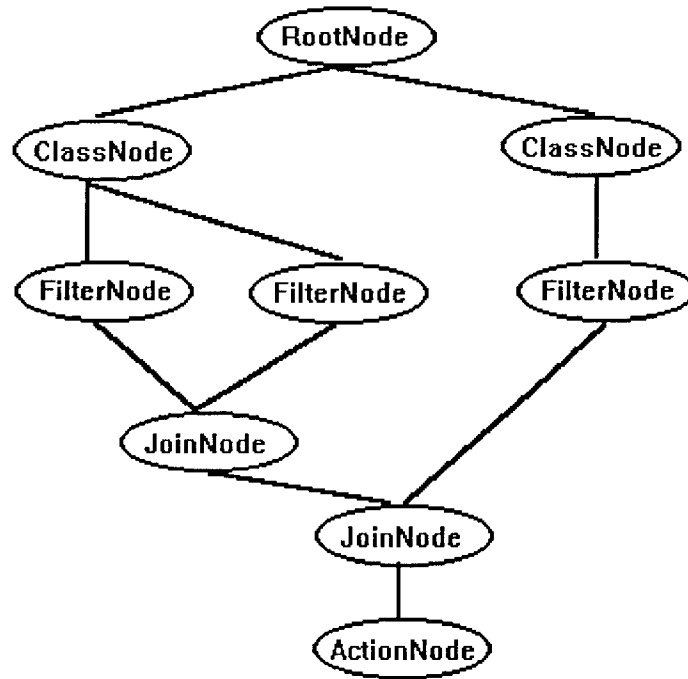
Figure 5.8 An Example of RETE

- The **RootNode** is at the top of the network. It links together the different Classnodes.

- **ClassNodes** and **FilterNodes** together check the status of a condition-element. The **Classnode** checks for the classname of an object and the **FilterNode** checks if the object's attributes match a given **Pattern**. Further details on pattern-matching are given in Section 5.5.2.

- **JoinNodes** link two **FilterNodes** together. They have a `right-memory` and a `left-memory` which play the role of a beta-memory. Another attribute is a the type of the node: this type may be AND or OR; in a rule, condition-elements are linked by logical connectors (AND or OR) and the **JoinNode** will have different behaviors depending on its type.

- **ActionNodes** are equivalent to RHS of rules. They have `attributes rulename` and `priority`, and a list of **Expressions** which will be evaluated if the rule is fired.

5.4.2 Detailed Description

The declaration of **ReteNode** class is shown in Table 5.11

Table 5.11 ReteNode Class

class ReteNode : public ObjectB
Protected: Tlist(ReteNode) lnext; // successors of the node

protected:
virtual int distribute_token(Token *);
public:
virtual int process_token(Token *, ReteNode *);
virtual int link(ReteNode *);
ReteNode(); // Constructor

the methods are explained below:

- *distribute_token()* sends a copy of the current processed **Token** to each of the successors of the node (members of lnext). This virtual method may be redefined by some of the nodes.

- *process_token()* is redefined by each specialization of **ReteNode**. This method contains the know-how of each node.

- *link()* allows a node to be linked with another. The other node is then pushed onto **lnext**. This method is used during the generation of the Rete network.

Table 5.12 Terminal_Link_Mixin Class

class Terminal_Link_Mixin
protected: ActionNode *action;
public: void link_action();

The declaration of **Terminal_Link_Mixin** class is shown in Table 5.12. This Mixin allows a node to be linked with an **ActionNode**. The method *link_action()* is used for that purpose during the Rete network generation.

The declaration of **ClassNode** class is shown in Table 5.13. One method is explained below:

- *process_token()* checks if the received **Token** belong to the class specified by **classname**. If not, the **Token** is disgarded. If context is non NULL, a **Binding** is created between the variable **context** and the object pointed by the slot proot of the **Token**. Then, *distribute_token()* is invoked.

The declaration of **FilterNode** class is shown in Table 5.14. The methods are explained below:

- *process_token()* carries out the pattern-matching between the processed Token and the reference **Pattern**, **refpattern**. The algorithm is explained in Section 5.5.2. Then, distribute_token() is invoked.

- *add_pattern()* pushes new **Patterns** onto `refpattern`. It is used during generation of the Rete network.

- *link()* pushes a **JoinNode** onto `lnext`. It also places the **FilterNode** on the appropriate channel of the **JoinNode** (one may be already used).

The declaration of JoinNode class is shown in Table 5.15. The methods are explained below:

- *install_channel()* is used to establish a link with a **FilterNode** or another **JoinNode** during the generation of the Rete network.

- *process_token()* tries to join a received **Token** with others from its `right_memory` or `left_memory`. If the type of the Node is OR, then the **Token** is immediately distributed. Otherwise, the type is AND, and the merge operation is performed as explained in Section 5.2.3.

- *link()* has the same functionality as the **FilterNode**'s one.

The declaration of **ActionNode** class is shown in Table 5.16. The methods are explained below:

- *add_to_conflict_set()* creates an **Action** out of the received **Token** and a pointer to the **ActionNode** itself. Then, this **Action** is sent to the **ConflictSet**.

- *add_action()* is used during generation of the Rete network to add a newly parsed expression to `laction`.

- *fire()* is used, once a rule has been selected, to execute its RHS. Every **Expression** in `laction` is evaluated given the list of **Bindings**.

- *set_rulename()* and set_priority() provide private access to `rulename` and `priority`.

The following section will describe the actual processing of a token by the network.

5.5 Network Interpretation

In this section, we describe how the inferencing is performed.

5.5.1 Token Flow

Figure 5.9 illustrates what happens when a change occurs in the working memory. A **Token** is created and sent to the **RootNode** of the network. Then, the token is sent from node to node and if all matchings succeed, a with associated bindings is sent into the **ConflictSet**.
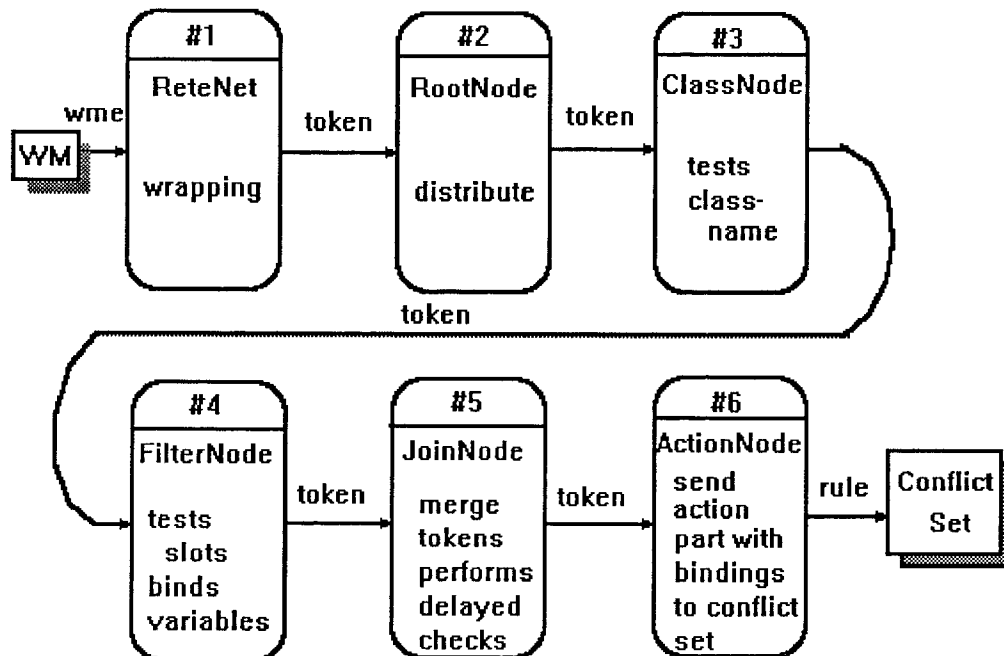
Figure 5.9 Flow in the Network

Actually, if the token match several patterns, this path may be followed by several copies of the token. It results in different bindings for each copy, and thus a same wme may trigger may rules.

5.5.2 Matching

The left-hand-side of a rule is a list of tests or *statements*. Each statement checks the state of a working-memory element. When all the tests succeed, then the rule matches and is a candidate for *firing*. There are two basic kinds of operations allowed on the LHS of a rule: *testing* and *binding*.

Depending on the form of the *statement*, different operations are performed. A *statement* has the general form: (<classname> [<context>][<slot ><lop><value>]).

Suppose that a token is processed in the network. Let us examine the various match operation between the token and a statement:

A first *test* occurs to check that the wme belongs to the specified class. This is done by the **ClassNode**. Then, the **FilterNode** checks each *<slot> <lop> <value>*. Each **FilterNode** has an attribute which is a list of **Pattern** instances. Each **Pattern** has the attributes: `slot`, `operator`, `value`. The pattern-matching between a **Token** and a **Pattern** is described below:

(a) **Simple test:** Where *<value>* is a constant, e.g.: a string, a float, or even a point. The operation performed is a `test`.

```
SimpleTest(Token<lw,lb,lc>):
Begin
   - letWME <- first(lw)
   - retrievevalue and type of the slot of WME
   - Depending on the operator, perform the boolean test
   between this retrieved value and the constand.
   - If the result is true
     then   accept the Token
        else   refuse it
End
```

<u>Example:</u> radius == 3.

Suppose that a wme of class "circle" is processed and its radius is 4. The above test will return a false value and the **Token** will be rejected.

(b) **Simple binding:**, where <*value*> is a variable, and <*lop*> is the operator equal:

```
SimpleBinding(Token<lw, lb, lc>):
Begin
   -let WME <- first(lw)
   - retrievevalue and type of the slot of WME
   - create symbol(value, type)
   - create Binding(variable, symbol)
   - If a binding for thesame variable already is in lw
        then If   same binding
                      then  accept the Token
                      else  refuse it
        else  - add binding to lw
               - accept the Token
   end
```

<u>Example:</u> radius == $X

$X will be bound to 4, and added to the token's bindings. If $X was previously bound to another value, the token would be rejected.

(c) **Complex Binding:**, for all the other cases. This case could also be called delayed check since no test can be performed at once. The rule statement

involves variables - not bound yet - in a complex expression such as: slot $== 2 * \$X + 1 - \$Y^2$ or slot $> \$X+1$ and so on. Note that this kind of matching is not available in any other existing tool. Usually, the kind of pattern allowed in a LHS is of type 1 and 2 only (see in [3]). When one wants to perform a complex test in OPS5, one must use a Lisp-dependent hack.

In COSMOS, we make the assumption that if a complex test is used, the variables appearing in this test must be bound somewhere else in the LHS. The check algorithm is the following:

The check occurs in the next stage of processing, and is carried out by a **JoinNode**.

```
ComplexBinding(Token<lw, lb, lc>):
Begin
  - let WME <- first(lw)
  - retrieve value and type of the slot of WME
  - construct a Check as following:
    - the boolean operator is the statement operator
    - the second argument is the complex expression
      with variables.
  - If the Check is evaluable. given lb.
      then  - evaluate the Check
            - apply SimpleTest(Token<lw, lb, lc>)
      else  - add the Check to lc
            - accept the Token
  end
```

Example: radius $> \$X$

The **Check** $(4 > \$X)$ is built, and will be evaluated by a **JoinNode** where $\$X$ will be bound to a specific value.

5.5.3 Merging

JoinNodes are two inputs nodes which receive tokens in their left or right-memory. Basically, a **JoinNode** performs a logical operation (AND or OR) on two statements, or condition-elements.

5.5.3.1 AND Case

This is the most common case, and the only one considered in Forgy's implementation. Suppose a Token enters the right or left channel of a JoinNode J. J will try to merge the received token with all the tokens stored in the opposite memory. If Checks are satisfied, and Bindings are compatible, then, tokens are merged.

Definition 2 Two bindings $<v_1, s_1>$, $<v_2, s_2>$ are compatible

$$iff \left(v_1 \neq v_2 \right) \vee \left(v_1 = v_2 \wedge s_1 = s_2 \right)$$

Definition 3 Let $t_1 = <lw_1, lb_1, lc_1>$ and $t_2 = <lw_2, lb_2, lc_2>$ be two tokens. The Merge operation is defined by:

$Merge(t_1, t_2) = <lw_1 \ U \ lw_2, \ lb_1 \ U \ lb_2, \ lc_1 \ U \ lc_2>$

Note that before the Merge operation, some Checks may have been deleted from lw if they were satisfied.

*If a Token enters the right-memory, the opposite-memory is the left-memory, and vice-versa.

5.5.3.2 OR Case

The algorithm is straightforward: as soon as a token enters the JoinNode, it exits it, i.e., no merging is necessary. The OR-test has been added for convenience but is very restrictive. Indeed, an OR between two condition-elements should be performed only if these elements are ground, i.e.: none of

the tests contains any variable (example: color = "blue" or price < 300). Otherwise, the consistence in the network could not be maintained anymore.

5.6 Conflict Resolution

5.6.1 Introduction

There are two aspects of management in the conflict-set:

(a) when there is more than one rule in the conflict-set, only one must be fired. A conflict resolution strategy must be chosen.

(b) If one adds a wme to the working-memory, a rule may be added to the conflict-set. But what happens when a wme is removed? Some rules in the conflict-set may become invalid and should thus be removed.

This section shows how these two issues are addressed in COSMOS. In particular, we show that issue 2 is solved more efficiently than Forgy's algorithm. We begin by introducing the conflict-Set structure.

5.6.2 Conflict Set Data Structure

The **ConflictSet** is a class whose instance is an essential part of **theInferenceEngine**. Figure 6.12 shows its structure.

It consists basically of a list of **Actions**, also called rules instantiations (slot llaction), and a **strategy**. The **strategy** indicates to the **ConflictSet** which conflict-resolution method to choose during the select part of the inferencing cycle (see next section). An **Action** (we will denote it as Action <lt, anode>) consists of a list of **Tokens** (along with Bindings) which have passed through the RETE network, and an instance of **ActionNode** which contains the RHS of the rule to be fired.
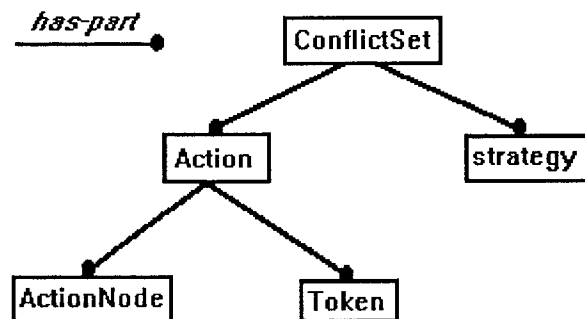
Figure 5.10 Conflict-Set Structure

5.6.3 Conflict Types

Currently, there are two strategies implemented in the forward-chainer: LEX and RPR. One can easily add a new strategy if needed.

5.6.3.1 LEX

We use the following sequence of steps, as reported in [3]. The LEX strategy proceeds in two steps:

(a) **refraction.** A rule which has been previously fired is disgarded from the conflict-set. Note, however, if one of the wme's of the instantiation has been modified, the rule won't be eliminated.

(b) **LEXical** ordering. We order the instantiations remaining in the conflict-set on the basis of the regency of the time tags of the wme's in these instantiations. All instantiations which have the largest time-tag are kept in the conflict-set, others are disgarded. Then the process is repeated for the second largest time-tag, and so on. If these attempts fail to determine a unique instantiation, an arbitrary one is chosen.

Note, that unlike in OPS5, the *specificity test* is not implemented.

5.6.3.2 RPR

RPR stands for rule priority. This strategy is convenient because it is simple and much faster than the previous one. In many cases, controlling inferencing through ordering of rules may be suited to the problem one has to solve.

The strategy is straight-forward. At a given cycle, sort the rules in the conflict-set according to their priority. If several rules have the same priority, then choose one randomly.

5.6.4 Consistency of the Network

When a wme is modified, many Tokens which depend on it must be modified accordingly.

Definition 4 A Token t<*lw,lb,lc*> depends on a wme w *iff*: $w \in lw$.

By transitivity, an Action in the **ConflictSet** may *depend* on a wme:

Definition 5 An Action A<lt,anode> depends on a wme w *iff*: $\exists t \in It$ such that: t depends on w.

When, a wme is removed from the memory, the following actions must be taken:

(a) Update the network to maintain consistency. It means that the wme should be removed from all its locations in the network.

(b) Update the conflict-set. Every instantiation of a rule which involves the given wme must removed from the conflict-set.

This is an important problem since it usually consumes a considerable amount of the inferencing time. Indeed, this update occurs at every call to a Remove operator or a Modify operator (a Modify operation can be viewed as a Remove followed by a Make).

This issue was addressed in [8] by searching the network to find the location at the given wme. For example one can send the token in the network

with an appropriate tag (a delete tag). The inferencing is carried out the same way as usual, except that the wme is deleted from the network instead of being added.

Our algorithm is based on the following reasoning. Why not use the time when the token is added to the network to record its locations in a data-structure called a map? When a deletion occurs, one just has to look at this map and remove the wme instantaneously. If we add to the map the locations where the wme appears in the conflict-set, we address issue 2, as well.

The gain in speed must involve an extra space occupancy. We can show that this is not a major problem, for the size of a map can be reasonable.

Figure 5.11 shows our implementation of this map: each wme has a slot ldel which is a list of DestroyElements. The first time a wme is created, a Token is sent to the network. Every beta-memory where this Token <lw,lb,lc> is stored, is recorded in the DestroyElements of all wmes in lw. This establishes the causal relation "depends-on" (as defined above) between all the wmes in lw and the Token. We do the same thing when a Token is sent into the ConflictSet as part of an Action.
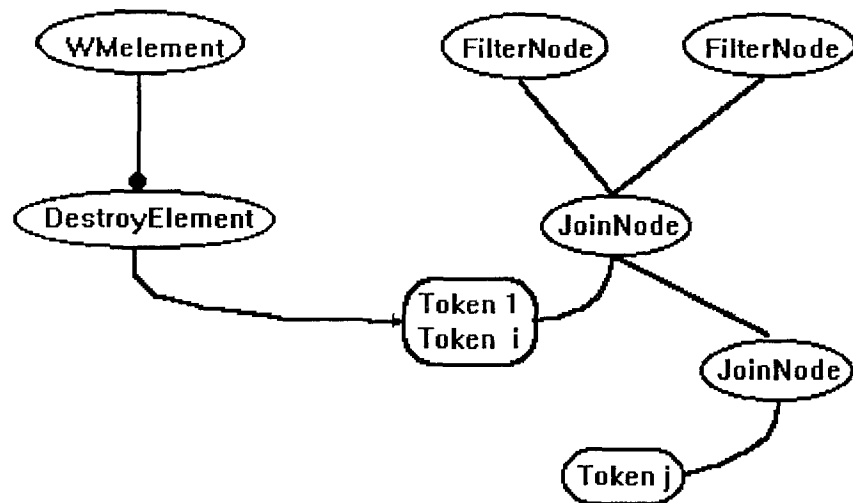
Figure 5.11 Dependence Between wmes and Tokens

Now, whenever a wme is removed from the WorkingMemory, we remove all Tokens and Actions which depend on it.

5.7 Detailed Example

In this section we will illustrate the generation of the RETE network for a particular case, along with the processing that occurs in the network.

Consider the following rule:

```
(RULE aRule
        IF (CLASS: Container
            (base == $BASE) AND
            (height == $HEIGHT) AND
            (name == $NAME1))
            AND
            (CLASS: Liquid
            (volume < $BASE*$HEIGHT) AND
            (name == $NAME2))
        THEN
            (PRINT $NAME2 " fits in " $NAME1))
```

We illustrate the following steps:

(a)   Parsing and the generation of the RETE network

(b)   Processing of some objects in the network

5.7.1 Generation of the Network

Building is carried out by performing the following subtasks:

(a)   Parse the rule and generate the intermediate structure.

(b)   Instantiate the objects representing the nodes of the RETE network.

The first step is carried out by the parser (see Parser chapter). Then instanciation of nodes proceeds as follow:

(a) Instantiate the top of the network: RootNode, ClassNodes and FilterNodes as given by the intermediate structure.

(b) Loop on every rule structure and instantiation of all JoinNodes between two FilterNodes. Redundancies are eliminated.

The network generated is given in Figure 5.12
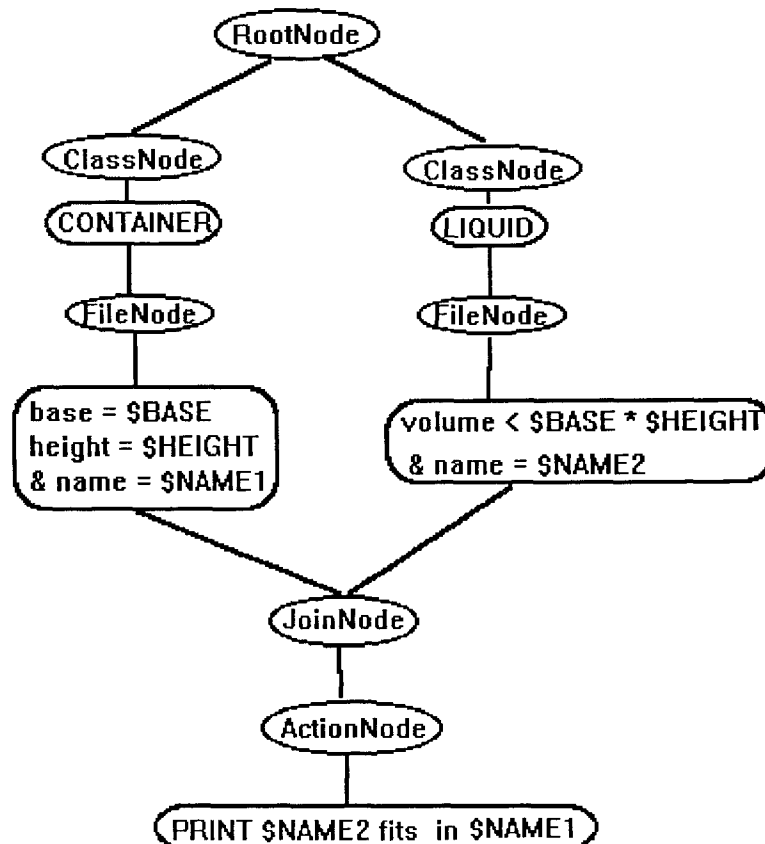
Figure 5.12 Instantiation of the Network After Parsing

5.7.2 Processing

Suppose the following instance is created in the working-memory (either by the user or by the firing of a rule):

```
instance:  my_tea
name:      "my tea"
volume:    30
```

Since this is a change in the working-memory, a **Token** is created and sent to the **RootNode**. The **Token** will match the **ClassNode** which test for the class *Liquid*, and will be sent to **FilterNodes** it is linked with. At this stage, one simple binding is performed between NAME2 and "my tea". Then, a **Check** is created since none of the variables HEIGHT and BASE are known at this stage. The value of volume is retrieved (which is 30) and the **Check** (30 < $HEIGHT*$BASE) is created. The **Token** is sent to the right channel of the **JoinNode** and it stays there. The **JoinNode** state is shown if Figure 6.15.
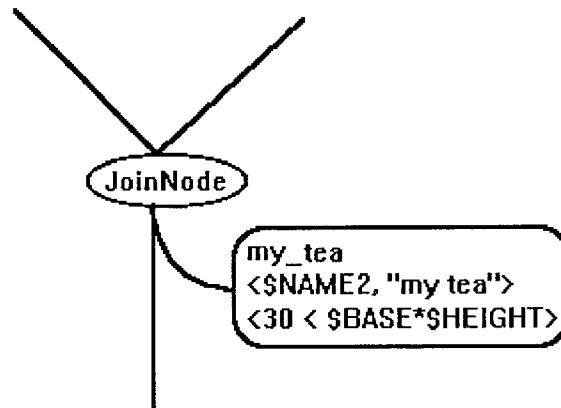
Figure 5.13 JoinNode State

Assume that a second instance is created:

```
instance:  my_mug
name:      "my mug"
base:      12
height:    4
```

A token is sent through the network and **ClassNode** which tests for the class Container is passed. Then, three simple bindings are created involving HEIGHT, BASE, and NAME1. The **Token** is sent to the left channel of the

JoinNode and the merging process begins. Since there aren't any inconsistent bindings between my_mug and my_tea, binding-lists are merged. Now, the **Check** can be evaluated since all variables are bound. The check is successful since 30 < 12*4, and the merged token is sent to the next node. The next node is actually an ActionNode and the following instanciation of the rule will sent to the **Conflict-Set**:

- The expression representing the executable action

- The Token, including its binding-list and wme-list

Figure 5.14 shows the state of the network at the end of processing.
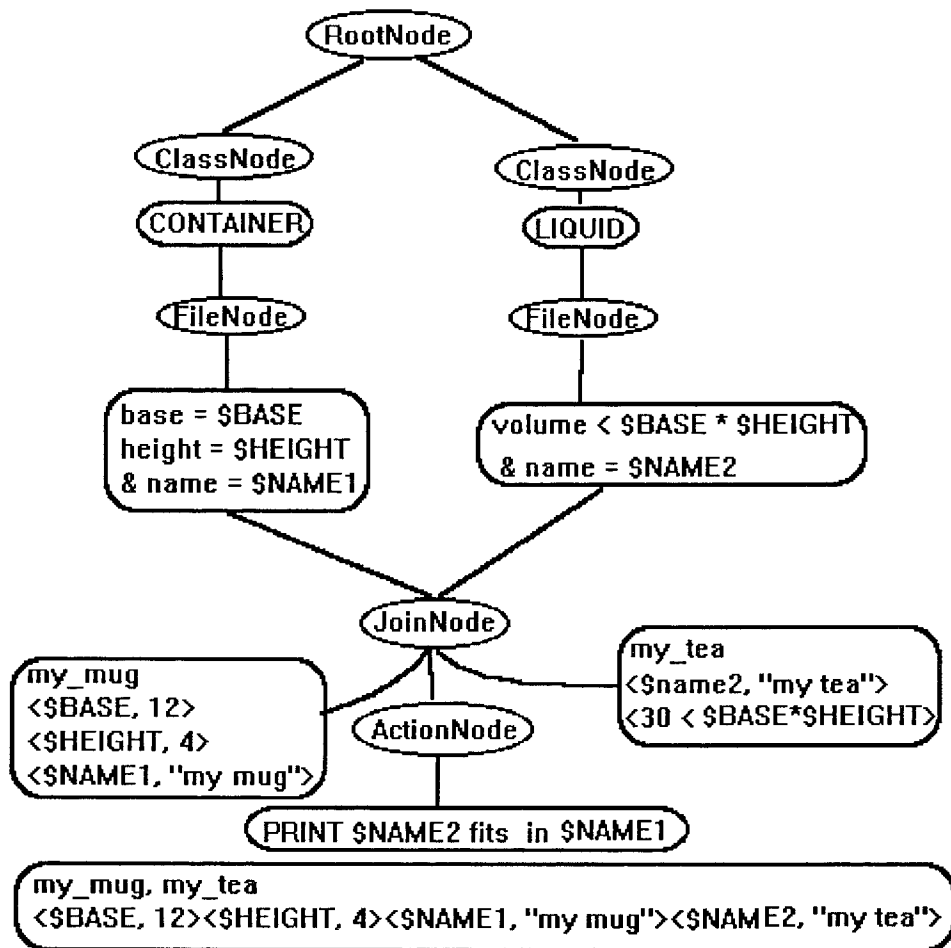
Figure 5.14 End of Processing

CHAPTER 6

DISCUSSION

The Inference Engine Monitor module provides the facilities for starting and monitoring the inferencing process, as well as checking the status of the working memory. The main interface function is the callback iemonitor() which initializes the widget tree and maps it onto the screen. The layout and widget hierarchy are shown in Figure 6.1 and Figure 6.2
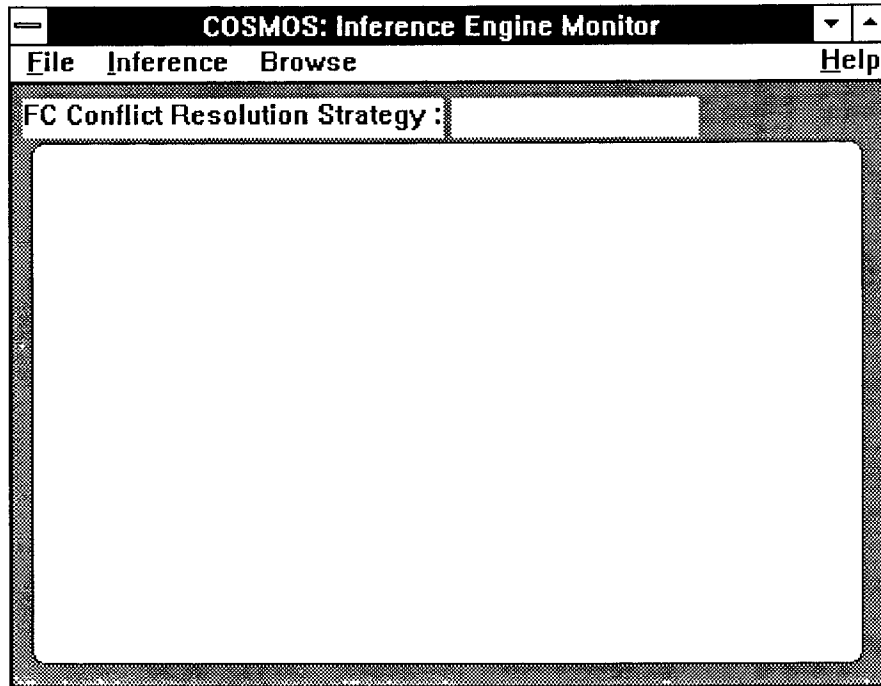
Figure 6.1 Layout of the Inference Engine Monitor Window

We described the forward-mechanism in COSMOS. It is based upon a variant of the RETE algorithm. However we have shown that our version is more powerful than OPS5.

The C++ implementation consists in definition of about 30 classes for the whole forward-chainer and their associated methods (about 130).

The size of the above implementation is about 5000 lines of C++ code. In appendices, we just list some of the programs.

```
┌─────────────────────────┐
│ Inference Engine Monitor│
└─────────────────────────┘
            │
    ┌───────────────┐
    │ TopLevelShell │
    └───────────────┘
            │
    ┌────────────────┐
    │ XmMainWindow   │
    └────────────────┘
         ╱        ╲
┌───────────┐    ┌─────────┐
│ XmMenuBar │    │ XmText  │
└───────────┘    └─────────┘
```

┌──────────────────┐ ┌──────────────────┐ ┌──────────────────┐ ┌──────────────────┐
│ XmCascadeButton │ │ XmCascadeButton │ │ XmCascadeButton │ │ XmCascadeButton │
│ (File)   │ │ (Inference)   │ │ (Browse)   │ │ (Help)   │
└──────────────────┘ └──────────────────┘ └──────────────────┘ └──────────────────┘
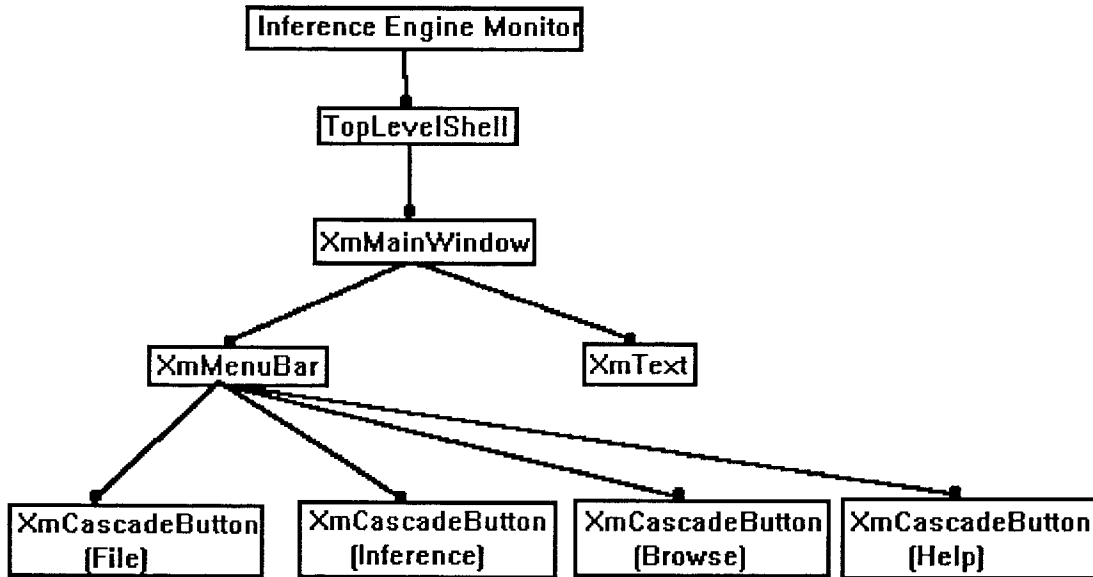
Figure 6.2 Widget Hierarchy of the Inference Engine Monitor Window

6.1 User-Interface Objects Between X Window and ObjectWindows

Windows has built-in support for a number of user-interface objects: windows, icons, menus, dialog boxes, etc. Built-in support means that the amount of effort required to create and maintain these objects is fairly minimal. In particular, if you were to write your own code to support these objects, it would require a vast amount of effort on your part. And the results would probably not be as flexible nor as robust as the user-interface objects that Windows provides.

Taking advantage of what these user-interface objects can provide requires you to understand how each is implemented. As we look at the different types of user-interface objects, we'll provide some insights into the design and

implementation of each. In many case, this will mean a discussion of the messages that are associated with a given user-interface object. In other case, this means delving into the various Windows library routines that control each type of object. For now, we're going to introduce you to the user-interface objects and describe the role of each in the user interface.

Among user-interface objects, the most important is the window. Any program that wishes to interact with the user must have a window, since a window receives mouse and keyboard input and displays a program's output. All other user-interface objects, like menus, scroll bars, and cursors, play supporting roles for the leading character: the window.

6.1.1 The Window

The window is the most important part of the user interface. Form the perspective of a user, a window provides a view of some data object inside the computer. But it is more than that, since to a user, a window is an application. When the user starts to run an application, a window is expected to appear. A user closes a window to shut down an application. To decide the specific application to be worked with, a user selects the application's window.

To programmers, a window represents several things. It serves to organize the other user-interface objects together and directs the flow of messages in the system. A window provides a display area that can be used to communicate with the user. Input is channeled to a window and thereby directed to the program. Applications also use windows to subdivide other windows. For example, dialog boxes are implemented as a collection of small windows inside a larger window.

Every window is created from a window class. A window class provides a template from which to create windows. Associated with every window class--

and therefore with every window--is a special type of subroutine called a window procedure. The job of a Windows, you can imagine that this is an important task. In fact, most of the work that you will do as a Window, which arrives in the form of messages. It receives notifications about other events of interest, such as changes in the size and location of a window. Here is an example in COSMOS

```
n = 0;
XtSetArg( wargs[n], XmNtitle, "COSMOS: Inference Engine Monitor" );
n++;
XtSetArg( wargs[n], XmNiconName, "IE Monitor" ); n++;
Widget toplevel = XtAppCreateShell( "ieMonitor", "COSMOS",
                                applicationShellWidgetClass,
                                display0, wargs, n );
Widget *topl = (Widget *)XtMalloc( sizeof(Widget) );
*topl = toplevel;
submenu_1[ quit_entry ].data = (caddr_t)topl;
```

In X Window, Rather than dealing directly with windows, applications built using the Xt Intrinsics use Widgets. In addition to the ID of the X window used by the widget, the widget structure contains additional data needed by these procedures.

The shell widget serves as a wrapper around its child, providing an interface between the child widget and the window manager. Applications that use multiple, independent windows must create an additional shell widget for each top-level window. The function

<div align="center">XtCreateWidget()</div>

provides the general mechanism for creating all widgets except shell widgets.

To create a second (or third, or whatever) top-level shell widget, using XtCreateApplicationShell() or XtAppCreateShell()-- The later for those who have Release 4 of the X Window System. You can use this second

top-level shell as a parent to create a child widget, and then create many more child widgets of that child, and so on, just like we've done in our application

In the other hand, life in a ObjectWindows program starts in the WinMain function. Most OWL programs will have a WinMain function that look like IE2's:

```
int PASCAL WinMain(HANDLE hInstance,
                   HANDLE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)
{
   Tie2 ie2 ("COSMOS: Inference Engine Monitor", hInstance,
       hPrevInstance, lpCmdLine, nCmdShow);
     MakeHelpPathName(szHelpFileName);
   ie2.Run();
   return ie2.Status;
}
```

An application object is created and run.

6.1.2 Icons

An icon is a symbol that serves as a reminder to the user. GUI systems are built on the principle that what is concrete and visible is more easily understood than what is abstract and invisible. Icons provide a concrete, visible symbol of a command, a program, or some data. By making such things visible, a Windows program makes them accessible. By making all of a user's choices visible, Windows programs lessen the user's dependence on memorized information.
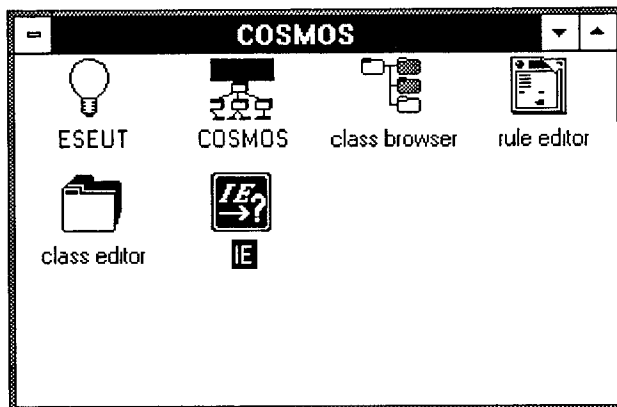
Figure 6.3 Icons in the COSMOS

Examples of icons include standard window ornaments: the system menu box, the minimize box, and the maximize box. As depicted in Figure 6.3, one of the most common uses of an icon is to represent a program. In the program Manager's window, an icon reminds the user of the programs that are available to be run. On the desktop, an icon serves to remind the user of the programs that are currently running, but whose windows have been closed. Icons can also be used to represent commands.

6.1.3 Menus

A menu is a list of commands and program options. Windows has five types of menus: system menus, menu-bar menus, pull-down menus, nested menus, and tear-off menus. The system menu, shown in Figure 6.4, provides a standard set of operations that can be performed on a window. These operations are referred to as "system commands." Users expect to find a system menu on the top-level window of every program they run. System commands require very little work on the part of a program, since Windows itself does everything to make system commands operational and uniform throughout the system.
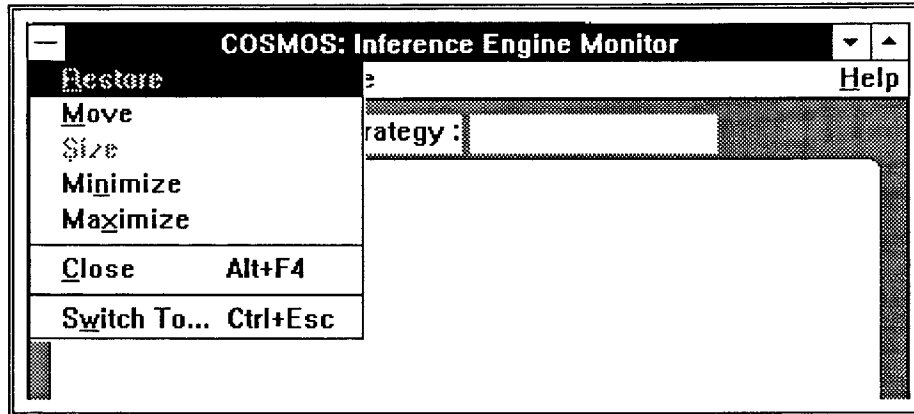
Figure 6.4 The System menu

Figure 6.5 shows the three types of menus that are connected together: The menu-bar menu connects to the top of a window, popup menus appear when a menu-bar item is selected, and nested menus are displayed when a popup menu item that has an arrow is selected. Applications can nest menus as far as they'd like, although in general programmers should avoid nesting too deeply, since this can disorient the user.
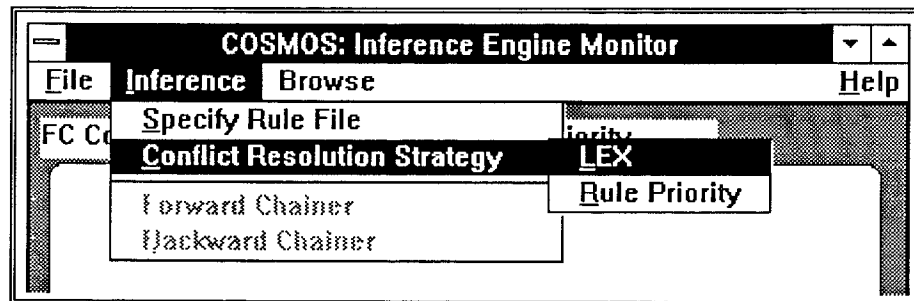
Figure 6.5 Three Types of Menus

In Motif, the menu structure is clearly defined by the **menubar arrays.** Adding a new entry requires only defining a callback function and adding an entry to the array that describes the menu.

The xc_create_menu_buttons() function loops through the array of menu entries creating an appropriate widget for each entry. Here is how Motif version COSMOS create menu:

```
static menu_struct submenu_22[] = {
  { "LEX", (XtCallbackProc)set_crs, NULL },
  { "Rule Priority", (XtCallbackProc)set_crs, NULL }
};
static menu_struct submenu_2[] = {
  { "Specify Rule File...", (XtCallbackProc)specify_rule_file, NULL },
  { NULL, NULL, NULL },
  { "Conflict Resolution Strategy", NULL, NULL,
      submenu_22, XtNumber( submenu_22 ), NULL },

static menu_struct MenuData[] = {
  {"File",    NULL, NULL, submenu_1, XtNumber( submenu_1 ), NULL },
  {"Inference", NULL,NULL,submenu_2,XtNumber( submenu_2 ), NULL },
  {"Browse",NULL, NULL, submenu_3, XtNumber( submenu_3 ), NULL },
  {"Help", (XtCallbackProc)HelpCB,NULL},
};

    Widget menubar = XmCreateMenuBar( main, "menubar", NULL, 0 );
    xc_create_menu_buttons( NULL, menubar, MenuData,
                          XtNumber( MenuData ) );
```

In ObjectWindows, an application accesses its attached resources by specifying the resource ID. This ID is an integer, such as 101, or a string identifier, such as "IDM_LEX". An application distinguishes one menu selection from another by the menu ID associated with each menu item. Here is the resource of menu and how the main program create the menu.

```
IE MENU
BEGIN
   ...
```

```
POPUP  "&Inference"
BEGIN
MENUITEM    "&Specify Rule File",IDM_SPECIFYRULEFILE
    POPUP      "&Conflict Resolution Strategy"
    BEGIN
        MENUITEM   "&LEX",       IDM_LEX
        MENUITEM   "&Rule Priority",IDM_RULEPRIORITY
    END
...
```

```
void TMainWindow::GetWindowClass(WNDCLASS _FAR & AWndClass)
{
   TDialog::GetWindowClass(AWndClass);
   AWndClass.hIcon = LoadIcon(AWndClass.hInstance, "IE");
   AWndClass.lpszMenuName = (LPSTR)"IE";
   AWndClass.hCursor = LoadCursor(AWndClass.hInstance, "IE");
}
```

In Motif version Cosmos, they use **rule_fname** as a flag to check the rule file specified or not, in program iemon.c:

```
if ( rule_fname == NULL ) {
   xc_warning( "Specify the rule file first!" );
   return;
```

The method we used is set the menu items `Forward Chainer` and `Backward chainer` to GRAY before rule file is specified (Figure 6.5). It disables the command and grays the displayed text. The shading lets the user know the command is not currently available.

6.1.4 Scroll Bars

When a scroll bar is shown in a window, the user knows that the data object is larger than the window, Scroll bars provide a menus by which the user can control the display of such objects and also see at a glance the relative location

of an object that is being viewed. Figure 6.6 shows one of the two types of scroll bars: vertical.
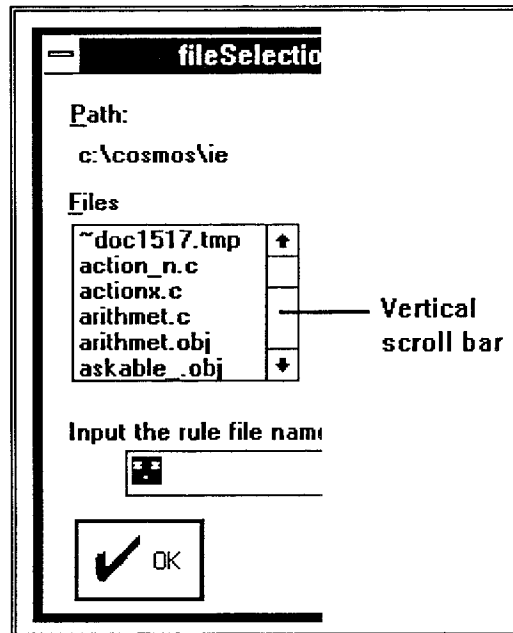
Figure 6.6 Vertical Scroll Bars

XmCreateScrolledList() creates a scrolled window widget and then a list widget as a child of the scrolled window widget in Motif. A scrolled window widget is first created and then a text widget is created as a child of the scrolled window. Here is an example in COSMOS.

```
Widget inst_list = XmCreateScrolledList( form,
                                        "instanceList", wargs, n );
```

A scroll bar is installed in a regular window that was created with either the WS_HSCROLL or WS_VSCROLL style bits. Also listboxes and edit controls can be created with a built-in scroll bar. Here is how we create a scroll bar in Figure 6.6.

```
CONTROL "", ID_DLIST, "LISTBOX", LBS_STANDARD |
    WS_CHILD | WS_VISIBLE | WS_HSCROLL, 91, 47, 64, 51
```

6.1.5 Dialog Boxes

Dialog boxes, also known as dialogs, provide a standard way to receive input from users. In particular, when a user has entered a command for which additional information is required, dialog boxes are the standard way to retrieve that input. While browsing though a windows program, you often see an ellipsis (...) as part of a menu name. This indicates that a dialog box will appear when the menu item is selected.

One dialog box that is quite common is displayed whenever the user asks for a file to be opened. It is the file-open dialog box, shown in Figure **. This dialog box provides the user with the opportunity of typing in a file name. It also shows two lists: one of file names and the other of directory names and disk drives. If the user is unable to remember a specific file name, he can browse the directories until he find the desired file.

Notice that this dialog box has two pushbuttons: one marked "Ok" and the other marked "Cancel." In general, pushbuttons in dialog boxes are used to request an action. For this dialog box, there are two possible actions. The Ok pushbutton tells the program to accept the values that the user has entered. The Cancel pushbutton tells the program to ignore the values that have been entered in the dialog box. In general, wherever possible, programs should allow a user to withdraw a request without incurring any damage to files or data.

```
Widget form = XmCreateFormDialog( parent, "instSelectionDialog",
                NULL, 0 );
```

To create a dialog box object in ObjectWindows, you will want to use the **new** keyword. This is the safest approach to take, since it works for both modal and modeless dialog boxes. Once a dialog box object has been created, you'll call one of two **TApplication** member functions to create the MS-Windows dialog box: **ExecDialog** (for a modal dialog box) or **MakeWindow** (for a modeless dialog box). Here is how we do it in Inference Engine Monitor Module.

```
GetApplication()->ExecDialog(
              new TDLG_STARTDlg(this, "DLG_START"))
```

Working Memory Select Dialog show in Figure A.3.

6.1.6 Dialog Box Controls

Most widgets allow the programmer to affect the way the widget appears or behaves in X Window by specifying values for resources used by the widget. Here, the term resource simply means any data used by the widget. The function XtCreateWidget() allows the programmer to pass an array specifying these resources. It is often more convenient for the programmer to use the macro XtSetArg(arg, name, value) to set a single value in a previously allocated argument list. For example:

```
XmString tcs1 = XmStringCreateLtoR( prompt.
          (XmStringCharSet)XmSTRING_DEFAULT_CHARSET );
XtSetArg( wargs[n], XmNselectionLabelString, tcs1 );  n++;
XmString tcs2 = XmStringCreateLtoR(tmp,
          (XmStringCharSet)XmSTRING_DEFAULT_CHARSET );
XtSetArg( wargs[n], XmNdirMask, tcs2 );  n++;
XtSetArg( wargs[n], XmNnoResize, False );  n++;
Widget w = XmCreateFileSelectionDialog( parent, "fileSelectionDialog",
          wargs, n );
```

Like all dialog boxes, the file-open dialog in ObjectWindows is a window that holds individual windows that either display information or accept input from the user. Each of these tiny windows is called a dialog box control. For example, the file-open dialog contains nine dialog box controls: two pushbuttons (Ok and Cancel), two listboxes, an edit control, and four static text controls, Windows has six predefined window classes from which dialog box controls are created: button, combobox, edit, listbox, scroll bar, and static. Here is resource code of file-open dialog. Here is how we create the file open dialog in resource file.

```
IE_FILEOPEN DIALOG DISCARDABLE LOADONCALL PURE
MOVEABLE      5, 17, 165, 166
CAPTION "fileSelectionDialog_popup"
FONT 8, "Helv"
CLASS "BorDlg"
STYLE WS_TILED | WS_CAPTION | WS_SYSMENU | DS_SETFONT
    | DS_MODALFRAME
BEGIN
    CONTROL "", -1, SHADE_CLASS, 1, 4, 2, 156, 30
    CONTROL "", ID_FNAME, "EDIT", WS_CHILD | WS_VISIBLE |
        WS_BORDER | WS_TABSTOP | ES_AUTOHSCROLL,
        24, 116, 128, 14
    CONTROL "", ID_FPATH, "STATIC", WS_CHILD | WS_VISIBLE
        |WS_GROUP,  10, 20, 146, 9
    CONTROL "", ID_FLIST, "LISTBOX", LBS_STANDARD |
        WS_CHILD | WS_VISIBLE | WS_HSCROLL, 9, 47, 64, 51
    CONTROL "", ID_DLIST, "LISTBOX",  LBS_STANDARD |
        WS_CHILD | WS_VISIBLE | WS_HSCROLL, 91, 47, 64, 51
    CONTROL "", IDOK, BUTTON_CLASS, BS_DEFPUSHBUTTON |
        WS_CHILD | WS_VISIBLE | WS_TABSTOP, 9, 137, 37, 24
    CONTROL "", IDCANCEL, BUTTON_CLASS, BS_PUSHBUTTON |
        WS_CHILD | WS_VISIBLE | WS_TABSTOP, 118, 138, 36, 24
    CONTROL "&Path:", -1, "STATIC", SS_LEFT, 8, 8, 38, 12
    CONTROL " &Files", -1, SHADE_CLASS, 1, 4, 34, 74, 67
    CONTROL " &Directories", -1, SHADE_CLASS, BSS_GROUP |
```

```
        WS_GROUP, 86, 35, 74, 67
    CONTROL " Input the rule file name:", -1, SHADE_CLASS,
        BSS_GROUP | WS_CHILD | WS_VISIBLE, 4, 105, 156, 28
    CONTROL "", -1, SHADE_CLASS, 2 | WS_GROUP, 0, 134, 165, 2
END
```

6.2 Recommendations

The recommendations are:

1. Provide interfacing with other programs such dBase IV, Lotus 1-2-3, Poet, and Symphony. An SQL interface for building client-server connectivity to other databases is a big help.

2. Creating interactive graphic objects, creating an application iconically under Windows is of great value. Color changes remind user of what he or she has chosen, speeding up the process considerably.

3. Improve in error-handling capabilities. Let user go back to previous points in development, and provide input-completion and consistency-checking facilities.

APPENDICES

Appendix A: Windows of Inference Engine Module
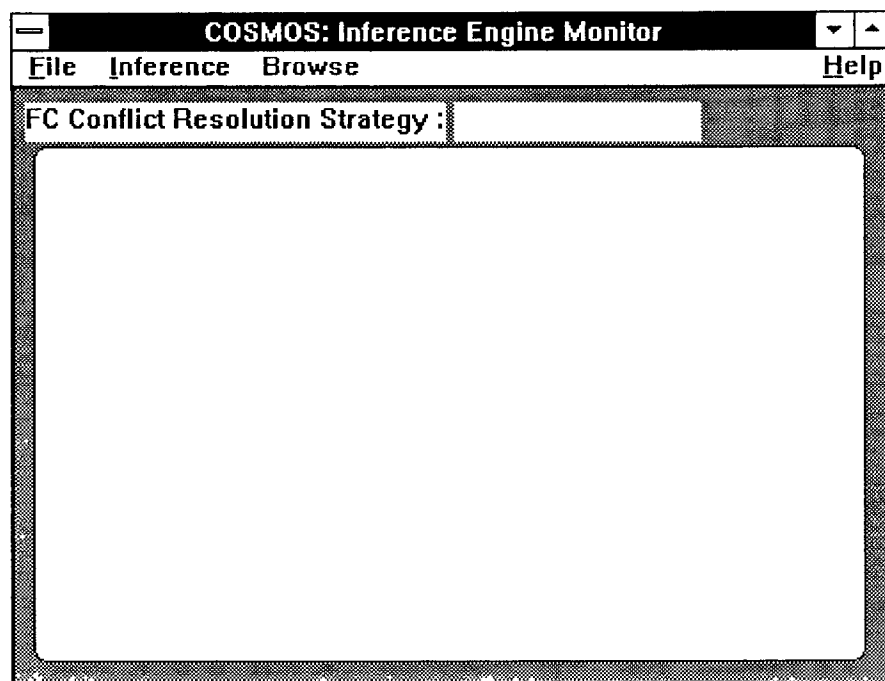
Figure A.1 Inference Engine Monitor Window
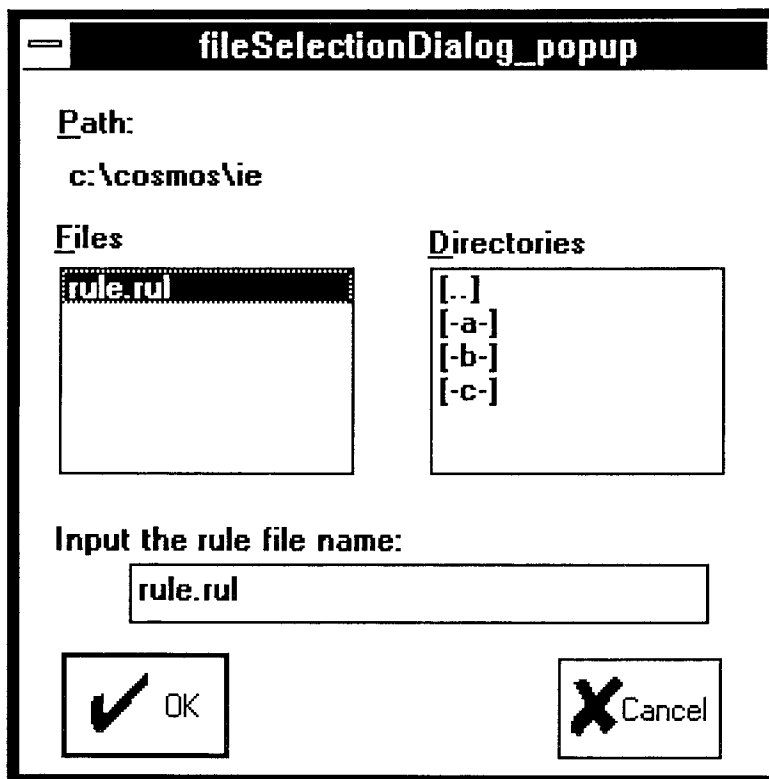
Figure A.2 Rule File Select Dialog

```
┌──────────────────────────────────────────────────────┐
│ ▬              startDialog_pupop                       │
├──────────────────────────────────────────────────────┤
│                                                        │
│  Start inferencing by:                                 │
│                                                        │
│    ✦  Initializing the working memory with All instances│
│                                                        │
│       Initializing the working memory with Selected instances│
│                                                        │
│       Adding Selected instances to the working memory  │
│                                                        │
│                                                        │
│        ┌─────────┐            ┌─────────┐              │
│        │ ✔  OK   │            │ ✗Cancel │              │
│        └─────────┘            └─────────┘              │
│                                                        │
└──────────────────────────────────────────────────────┘
```
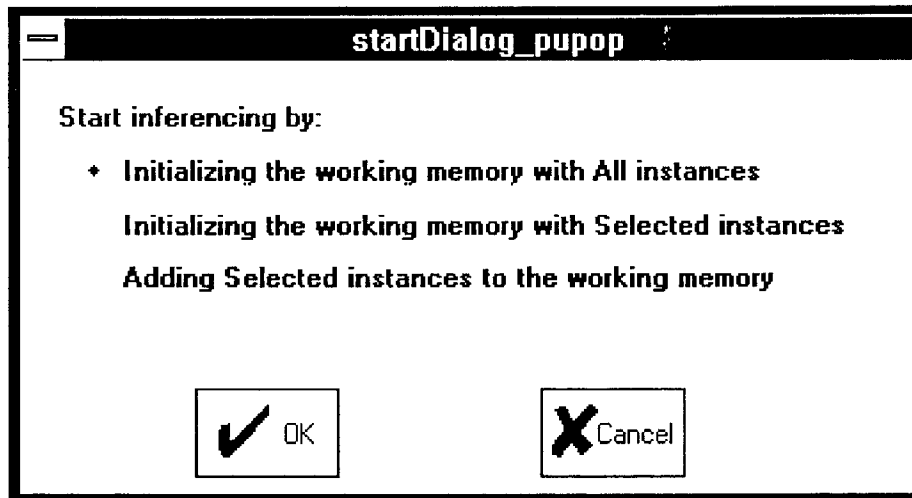
Figure A.3 Working Memory Select Dialog

```
┌──────────────────────────────────────────────────┬──┬──┐
│ ▬        COSMOS: Inference Engine Monitor         │ ▼│ ▲│
├──────────────────────────────────────────────────┴──┴──┤
│ File    Inference   Browse                        Help  │
├─────────────────────────────────────────────────────────┤
│ FC Conflict Resolution Strategy : LEX                   │
│  ┌────────────────────────────────────────────────────┐ │
│  │                                                    │ │
│  │  Hallow sir, my name is chiu                       │ │
│  │  Please turn on the key of your HONDA              │ │
│  │                                                    │ │
│  │                                                    │ │
│  │                                                    │ │
│  │                                                    │ │
│  │                                                    │ │
│  │                                                    │ │
│  │                                                    │ │
│  │                                                    │ │
│  │                                                    │ │
│  │                                                    │ │
│  │                                                    │ │
│  └────────────────────────────────────────────────────┘ │
└─────────────────────────────────────────────────────────┘
```
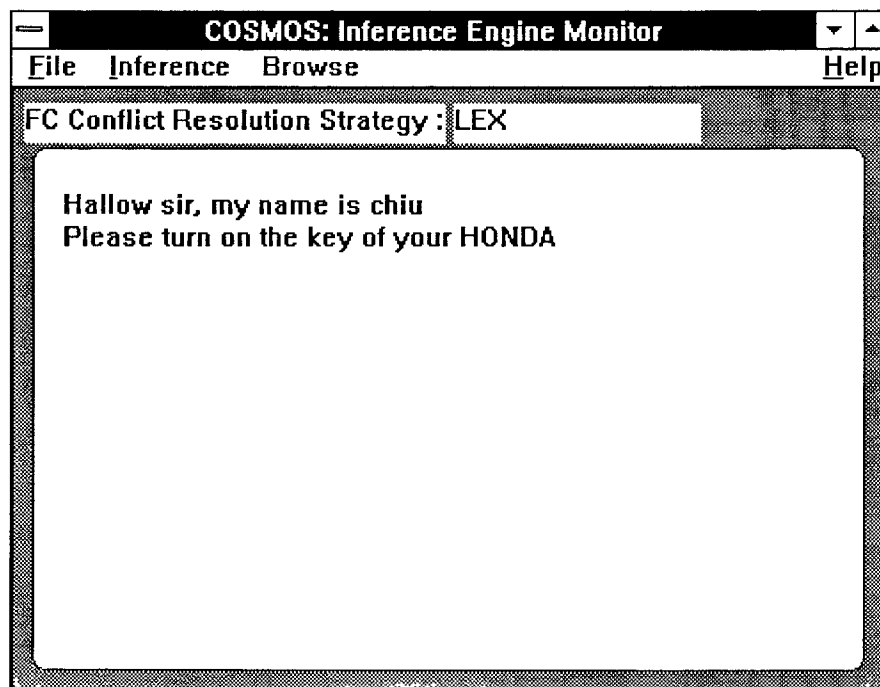
Figure A.4 Instruction Prompt
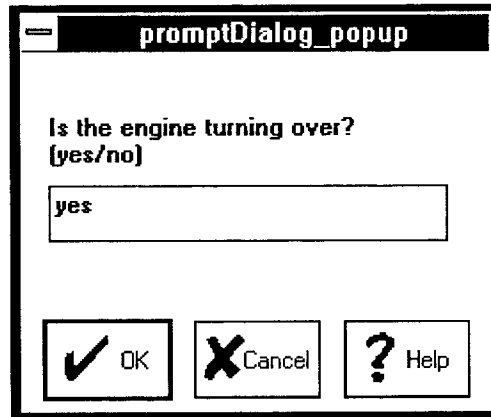
Figure A.5 Input Dialog
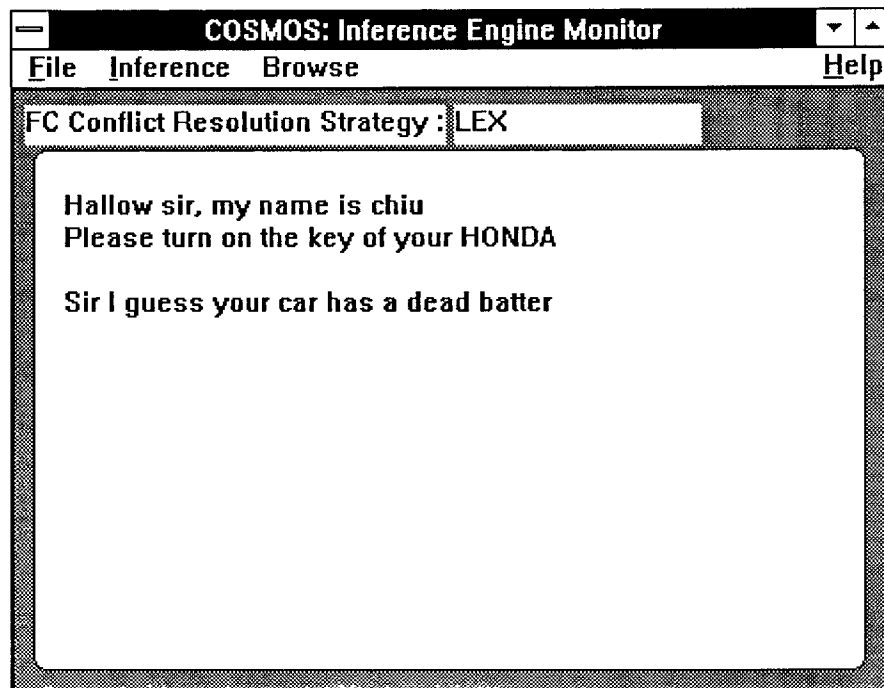
Figure A.6 Final Report Window

Appendix B: C++ Programs for Inference Engine

```
/*****************************************************************
 *
 *    Source File:    ie2.cpp
 *    Date:    Wed Jul 01 04:26:38 1992
 *    Author:    Ching-Jenq Chiu
 *    Supervisor: D. Wang
 *
 *****************************************************************/
#define __LIST_H    //disable "classlib"include"list.h
#include <owl.h>
#include <edit.h>
#include <stdio.h>
#include <listbox.h>
#include <dialog.h>
#include <bwcc.h>
#include <bstatic.h>
#include <filedial.h>
#include <inputdia.h>
#include "ie2.h"
#include "extern.h"
#include "ie.h"
#include "root.h"
#include "reteexte.h"


#define COMMANDMSG(arg) (arg.WParam)



char  szHelpFileName[MAXPATH];    /* Help file name*/
unsigned char apchIEout[500];
void MakeHelpPathName(char*);  /* Function deriving help file path */
int ie_monitor_flag = 0;
static char *rule_fname = NULL;
int rete_flag = 0;
Ie* theForwardChainer = new Ie();

// Define application class derived from  TApplication
class Tie2 : public TApplication
{
public:
  Tie2(LPSTR AName, HANDLE hInstance, HANDLE hPrevInstance,
```

```
          LPSTR lpCmdLine, int nCmdShow) : TApplication(AName,
                hInstance, hPrevInstance, lpCmdLine, nCmdShow) { };
            virtual void InitMainWindow();
};


// Declare TMainWindow, a TDialog descendant
class TMainWindow : public TDialog
{
public:
        TBStatic* ConflictResolution;
        TMainWindow(PTWindowsObject AParent, LPSTR ATitle);
        ~TMainWindow();
        virtual void LOADWMELEMENTS(RTMessage Msg) ;
        virtual void SAVEWMELEMENTS(RTMessage Msg) ;
        virtual void SPECIFYRULEFILE(RTMessage Msg);
        virtual void LEX(RTMessage )
                        { ConflictResolution->Clear();
                          ConflictResolution->SetText("LEX");}
        virtual void RULEPRIORITY(RTMessage )
                        { ConflictResolution->Clear();
                          ConflictResolution->SetText("Rule Priority");}
        virtual void FORWARDCHAINER(RTMessage Msg) ;
        virtual void BACKWARDCHAINER(RTMessage Msg) ;
        virtual void BROWSEWORKINGMEMORY(RTMessage Msg);
//      virtual void WARNING(RTMessage Msg, LPSTR str);
        virtual void PROMPT(RTMessage Msg, LPSTR str);
        virtual void WMPaint(RTMessage Msg)= [WM_PAINT];
        virtual void WMCommand(RTMessage Msg) = [WM_COMMAND];
    char path[50];
    unsigned int xposition;
    unsigned int yposition;
    int yinc;
    HDC hdc;

protected:
    virtual void GetWindowClass(WNDCLASS _FAR & AWndClass);
    virtual LPSTR GetClassName();
    virtual void SetupWindow();
}
/*************************************************
/
```

```
 * TMainWindow  implementations:
 ********************************************/

// Define TMainWindow, a TWindow constructor
TMainWindow::TMainWindow(PTWindowsObject AParent, LPSTR ATitle)
                                     : TDialog(AParent, ATitle)
{
   TEXTMETRIC tm;

   ConflictResolution = new TBStatic(this, ID_CONFLICTRESOLUTION,
                                          sizeof(path) );
   hdc = CreateDC("DISPLAY", 0, 0, 0);
   GetTextMetrics(hdc, &tm);
   yinc = tm.tmHeight + tm.tmExternalLeading;
   DeleteDC(hdc);
   strcpy(apchIEout, "I am chiu"nThis is prototype of Inference Engine"n");

}


// Define TMainWindow destructor
TMainWindow::~TMainWindow()
{
   ReleaseDC(HWindow, hdc);

}


void TMainWindow::SetupWindow()
{
   TDialog::SetupWindow();
   SetMenu(HWindow, LoadMenu(GetWindowWord(HWindow,
GWW_HINSTANCE), "IE"));
}


LPSTR TMainWindow::GetClassName()
{
      return "MainWindow";
}


void TMainWindow::GetWindowClass(WNDCLASS _FAR & AWndClass)
{
   TDialog::GetWindowClass(AWndClass);
   AWndClass.hIcon = LoadIcon(AWndClass.hInstance, "IE");
```

```
    AWndClass.lpszMenuName = (LPSTR)"IE";
    AWndClass.hCursor = LoadCursor(AWndClass.hInstance, "IE");


}


// Declare TDLG_STARTDlg, a TDialog descendant
class TDLG_STARTDlg : public TDialog
{
public:
    int select;
    TDLG_STARTDlg(PTWindowsObject AParent, LPSTR AName);
    virtual void Ok(RTMessage Msg) = [ID_FIRST + IDOK];
    virtual void Cancel(RTMessage Msg) = [ID_FIRST + IDCANCEL];
    virtual void RTDLG_ALL_INSTANCES(RTMessage Msg)
                            = [ID_FIRST + 233];
    virtual void RTDLG_SELECTED_INSTANCES(RTMessage Msg)
                            = [ID_FIRST + 234];
    virtual void RTDLG_ADD_INSTANCES(RTMessage Msg)
                            = [ID_FIRST + 235];


};


// Define TDLG_STARTDlg, a TDialog constructor
TDLG_STARTDlg::TDLG_STARTDlg(PTWindowsObject AParent,
                    LPSTR AName) :TDialog(AParent, AName)
{
 select = 1;
}
void TDLG_STARTDlg::Ok(RTMessage)
{
   char ch[3];
   HWND hwnd;

   //Notify parent that selection has been made.
   strcat(apchIEout, "You select ");
   itoa(select, ch, 10);
   strcat(apchIEout, ch);
   strcat(apchIEout, ""n");
   Destroy();
}
```

```
void TDLG_STARTDlg::Cancel(RTMessage Msg)
{
      switch(Msg.LP.Hi)
      {
        case BN_CLICKED :
  int Selection = MessageBox(HWindow, "Exit Program?", "Good Bye",
                                      MB_YESNO | MB_ICONQUESTION);
   if (Selection == IDYES)
   PostQuitMessage(0);
//              TDialog::Cancel(Msg);
   }
}

void TDLG_STARTDlg::RTDLG_ALL_INSTANCES(RTMessage Msg)
{
   switch(Msg.LP.Hi)
   {
        case BN_CLICKED :
             select = 1;
        break;
   }
}

void TDLG_STARTDlg::RTDLG_SELECTED_INSTANCES(RTMessage Msg)
{
      switch(Msg.LP.Hi)
      {
        case BN_CLICKED :
             select = 2;
        break;
      }
}

void TDLG_STARTDlg::RTDLG_ADD_INSTANCES(RTMessage Msg)
{
   switch(Msg.LP.Hi)
      {
        case BN_CLICKED :
             select = 3;
        break;
      }
}
```

```cpp
// Declare TDLG_PROMPTDlg, a TDialog descendant
class TDLG_PROMPTDlg : public TDialog
{
public:

    TDLG_PROMPTDlg(PTWindowsObject AParent, LPSTR AName);
    ~TDLG_PROMPTDlg();
        virtual void Ok(RTMessage Msg) = [ID_FIRST + IDOK];
        virtual void Cancel(RTMessage Msg) = [ID_FIRST + IDCANCEL];
        virtual void RTDLG_PROMPTIDHELP(RTMessage Msg)
                        = [ID_FIRST + IDHELP];
};


// Define TDLG_PROMPTDlg, a TDialog constructor

#pragma argsused
TDLG_PROMPTDlg::TDLG_PROMPTDlg(PTWindowsObject AParent, LPSTR
                        AName) :TDialog(AParent, AName)
{
}


// Define TDLG_PROMPTDlg destructor
TDLG_PROMPTDlg::~TDLG_PROMPTDlg()
{
}


void TDLG_PROMPTDlg::Ok(RTMessage Msg)
{
   switch(Msg.LP.Hi)
   {
        case BN_CLICKED :
                TDialog::Ok(Msg);
   }
}


void TDLG_PROMPTDlg::Cancel(RTMessage Msg)
{
   switch(Msg.LP.Hi)
      {
        case BN_CLICKED :
                int Selection =
                   MessageBox(HWindow,
                   "Exit Program?",
```

```
                    "Good Bye", MB_YESNO | MB_ICONQUESTION);
                    if (Selection == IDYES)
                    PostQuitMessage(0);
                        TDialog::Cancel(Msg);
//          }
}


void TDLG_PROMPTDlg::RTDLG_PROMPTIDHELP(RTMessage Msg)
{
    switch(Msg.LP.Hi)
    {
        case BN_CLICKED :
        MessageBox(HWindow,
                    "this program prompt the quations "
                    "for you to input the symptom "
                    "work with inference engine",
                    "HELP", MB_OK );
        break;
        }
}




void TMainWindow::LOADWMELEMENTS(RTMessage)
{
}

void TMainWindow::SAVEWMELEMENTS(RTMessage)
{
}



void TMainWindow::SPECIFYRULEFILE(RTMessage)
{
 char fileName[MAXPATH];
 strcpy(fileName,"*.rul");
 if(GetApplication()->ExecDialog(new TFileDialog(
                this,
            IE_FILEOPEN,
            fileName)) == IDOK)
 {
   strcat(apchIEout, "File you select ");
   strcat(apchIEout, fileName);
   strcat(apchIEout, ""n");
```

```
  }
  else
  {
    int Selection =
    MessageBox(HWindow,
             "Exit  Program?",
             "Good  Bye", MB_YESNO  |  MB_ICONQUESTION);
    if (Selection == IDYES)
               PostQuitMessage(0);
  }
}


void TMainWindow::FORWARDCHAINER(RTMessage)
{
  PTWindowsObject PTWndObj;
  // Create modeless dialog
  if((GetApplication()->ExecDialog(
        new TDLG_STARTDlg(this, "DLG_START"))) == IDOK)
    {
    }
}


void TMainWindow::BACKWARDCHAINER(RTMessage)
{
}


void TMainWindow::BROWSEWORKINGMEMORY(RTMessage)
{
}


/* void TMainWindow::WARNING(RTMessage, LPSTR str)
{
 MessageBox(HWindow, str, "Warning", MB_OK  |  MB_IDSTOP);
} */


void TMainWindow::PROMPT(RTMessage , LPSTR str)
{
       // Execute modal dialog
 char buf[20];

 strcpy(buf, "");
 if(GetApplication()->ExecDialog(new TInputDialog(
```

```
            this,
            "promptDialog_popup",
            str,
            buf,
            sizeof (buf)
            )) == IDOK)
  strcat(apchIEout, ""nResopnse: ");
  strcat(apchIEout, buf);
}

void TMainWindow::WMPaint(RTMessage Msg)
{
  HBRUSH hbrush;
  PAINTSTRUCT ps;
  DWORD dwSize;
  RECT r;
  HDC hdc;
  int xend, yend;
  int i = 0;
  hdc = BeginPaint(HWindow, &ps);

  GetClientRect (Msg.Receiver, &r);
  xposition = r.right;
  xend = r.left;
  yposition = r.top;
  yend = r.bottom;
  hbrush = CreateSolidBrush(0xFF80FF);
  SelectObject(hdc, hbrush);
  Rectangle(hdc, xposition, yposition, xend, yend);
  DeleteObject(hbrush);
  SelectObject(hdc, GetStockObject(WHITE_BRUSH));
  RoundRect(hdc, 10, 30, 425, 290, 15, 15);
  SetTextColor(hdc, RGB(0, 128, 0));
  SetBkMode(hdc, TRANSPARENT);
  xposition = 25;
  yposition = 50;
  while(apchIEout[i] != NULL)
  {
        if(apchIEout[i] == '"n')
          {
            yposition += yinc;
```

```
              xposition = 25;
            }
         else
         {
           TextOut(hdc, xposition, yposition,
                              &apchIEout[i], 1);
           dwSize = GetTextExtent(hdc, &apchIEout[i], 1);
           xposition += LOWORD(dwSize);
         }
         i++;
     }
  EndPaint(HWindow, &ps);
}

void TMainWindow::WMCommand(RTMessage Msg)
{
 PAINTSTRUCT ps;
 HMENU hmenu;
 HWND  hwnd;
 HDC hdc;
 char buffer[80];

 switch(Msg.WParam)
        {
        case WM_PAINT:
                WMPaint(Msg);
                break;
//      case ID_WARNING:
//              xc_warning(Msg, "test");
//              break;
        case IDM_LOADWMELEMENTS:
                LOADWMELEMENTS(Msg);
                break;
        case IDM_SAVEWMELEMENTS:
                SAVEWMELEMENTS(Msg);
                break;
        case IDM_QUIT:
                CloseWindow();
                break;
        case IDM_SPECIFYRULEFILE:
                SPECIFYRULEFILE(Msg);
                SendMessage(HWindow, WM_COMMAND,
```

```
                              ID_FILESELECTED,  0L);
               break;
        case IDM_LEX:
               LEX(Msg);
               break;
        case IDM_RULEPRIORITY:
               RULEPRIORITY(Msg);
               break;
        case IDM_BACKWARDCHAINER:
               BACKWARDCHAINER(Msg);
               break;
        case IDM_BROWSEWORKINGMEMORY:
               BROWSEWORKINGMEMORY(Msg);
               break;
        case IDM_AHELP:
               WinHelp(GetFocus(),szHelpFileName,HELP_INDEX,0L);
               break;
        case ID_FILESELECTED:
               hmenu = GetMenu(Msg.Receiver);
               EnableMenuItem(hmenu, IDM_FORWARDCHAINER,
                   MF_BYCOMMAND | MF_ENABLED);
               break;
        case IDM_FORWARDCHAINER:
               FORWARDCHAINER(Msg);
               hmenu = GetMenu(Msg.Receiver);
               EnableMenuItem(hmenu, IDM_FORWARDCHAINER,
                   MF_BYCOMMAND | MF_GRAYED);
//             SendMessage(HWindow, WM_COMMAND,
//                 IE_FORWARDCHAINING, 0L);
//             break;
        case IE_FORWARDCHAINING:
               PROMPT(Msg, "Enter some data:");
               break;
        default:
               MessageBox(HWindow, "Feature not implemented",
                         GetApplication()->Name, MB_OK);
    }
}

/*************************************************
/
```

```
 * Tie2App  method  implementations:
 **************************************************/


// Construct the Tie2's MainWindow of type TMainWindow
void Tie2::InitMainWindow()
{
   MainWindow = new TMainWindow(NULL,
"INFERENCE_ENGINE_MONITOR_WIN");
}


// Main program
int PASCAL WinMain(HANDLE hInstance,
                   HANDLE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)
{
HANDLE hBorLibrary;

   hBorLibrary = LoadLibrary("bwcc.dll");

   if(hBorLibrary <= 32)
       MessageBox(NULL, "Unable to load Borland Controls", "System Error",
                            MB_OK | MB_ICONHAND);
   Tie2 ie2 ("COSMOS: Inference Engine Monitor", hInstance, hPrevInstance,
        lpCmdLine, nCmdShow);
       MakeHelpPathName(szHelpFileName);
  ie2.Run();
  if(hBorLibrary > 32)
        FreeLibrary(hBorLibrary);

   return ie2.Status;
}
void MakeHelpPathName(char * szFileName)
{
   strcpy(szFileName, "helpex.hlp");
   return;
}
```

```
/*****************************************************************
 *
 *    Source File: ie2.h
 *    Date:          Wed Jul 01 04:26:36 1992
 *    Author:    Ching-Jenq Chiu
 *    Supervisor: D. Wang
 *
 *****************************************************************/

// Defines for menu item IDs
#define   IDM_LOADWMELEMENTS            101
#define   IDM_SAVEWMELEMENTS            102
#define   IDM_QUIT                      103
#define   IDM_SPECIFYRULEFILE           104
#define   IDM_LEX                       105
#define   IDM_RULEPRIORITY              106
#define   IDM_FORWARDCHAINER            107
#define   IDM_BACKWARDCHAINER           108
#define   IDM_BROWSEWORKINGMEMORY       109
#define   IDM_AHELP                     110
#define   IE_FILEOPEN                   111
#define   ID_CONFLICTRESOLUTION         112
#define   ID_FILESELECTED               113
#define   IE_FORWARDCHAINING            114
#define   UI_WARNING                    115
```

```
/****************************************************************
 *
 *      Source File: ie2.rc
 *      Date:          Wed Jul 01 04:26:44 1992
 *      Author:    Ching-Jenq Chiu
 *      Supervisor: D. Wang
 *
 ****************************************************************/

#include <d:"borlandc"include"windows.h>
#include <d:"borlandc"include"bwcc.h>
#include "ie2.h"
#include "ie.mnu"
#include "ie.dlg"
IE            ICON    IE.ICO
IE            CURSOR IE.CUR
```

```
/*****************************************************************
*
*    Source File:    ie2.mnu
*    Date:    Wed Jun 01 06:26:38 1992
*    Author:    Ching-Jenq Chiu
*    Supervisor: D. Wang
*
*****************************************************************/
IE MENU
BEGIN
    POPUP       "&File"
    BEGIN
        MENUITEM    "&Load WM Elements",IDM_LOADWMELEMENTS
        MENUITEM    "&Save WM Elements",IDM_SAVEWMELEMENTS
        MENUITEM  SEPARATOR
        MENUITEM    "&Quit",        IDM_QUIT
    END
    POPUP       "&Inference"
    BEGIN
        MENUITEM    "&Specify Rule File",IDM_SPECIFYRULEFILE
        POPUP       "&Conflict Resolution Strategy"
        BEGIN
            MENUITEM    "&LEX",     IDM_LEX
            MENUITEM    "&Rule Priority",IDM_RULEPRIORITY
        END
        MENUITEM  SEPARATOR
         MENUITEM    "&Forward Chainer",IDM_FORWARDCHAINER,
                    GRAYED
        MENUITEM    "&Backward Chainer",IDM_BACKWARDCHAINER,
                    GRAYED
    END
    POPUP       "Browse"
    BEGIN
        MENUITEM    "&Browse Working Memory",
                    IDM_BROWSEWORKINGMEMORY
    END
        MENUITEM    ""a&Help",IDM_AHELP
END
```

```
/*****************************************************************
*
*    Source File:    ie.dlg
*    Date:    Wed Jun 01 06:26:38 1992
*    Author:    Ching-Jenq Chiu
*    Supervisor: D. Wang
*
*****************************************************************/
#ifndef WORKSHOP_INVOKED
#include <windows.h>
#include <bwcc.h>
#endif
#include <d:"borlandc"owl"include"owlrc.h>


INFERENCE_ENGINE_MONITOR_WIN DIALOG 12, 33, 219, 160
CAPTION "COSMOS: Inference Engine Monitor"
STYLE WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU |
      WS_MINIMIZEBOX | WS_MAXIMIZEBOX
BEGIN
      LTEXT "FC Conflict Resolution Strategy :", -1, 3, 4, 105, 10,
           WS_CHILD | WS_VISIBLE | WS_GROUP
      LTEXT "", ID_CONFLICTRESOLUTION, 110, 4, 62, 10, WS_CHILD |
           WS_VISIBLE | WS_GROUP
END


IE_FILEOPEN DIALOG DISCARDABLE LOADONCALL PURE MOVEABLE
      5, 17, 165, 166
CAPTION "fileSelectionDialog_popup"
FONT 8, "Helv"
CLASS "BorDlg"
STYLE WS_TILED | WS_CAPTION | WS_SYSMENU | DS_SETFONT |
      DS_MODALFRAME
BEGIN
      CONTROL "", -1, SHADE_CLASS, 1, 4, 2, 156, 30
      CONTROL "", ID_FNAME, "EDIT", WS_CHILD | WS_VISIBLE |
           WS_BORDER | WS_TABSTOP | ES_AUTOHSCROLL,
           24, 116, 128, 14
      CONTROL "", ID_FPATH, "STATIC", WS_CHILD | WS_VISIBLE |
           WS_GROUP, 10, 20, 146, 9
```

```
CONTROL "", ID_FLIST, "LISTBOX", LBS_STANDARD |
      WS_CHILD | WS_VISIBLE | WS_HSCROLL, 9, 47, 64, 51
CONTROL "", ID_DLIST, "LISTBOX",  LBS_STANDARD |
      WS_CHILD | WS_VISIBLE | WS_HSCROLL, 91, 47, 64, 51
CONTROL "", IDOK, BUTTON_CLASS, BS_DEFPUSHBUTTON |
      WS_CHILD | WS_VISIBLE | WS_TABSTOP, 9, 137, 37, 24
CONTROL "", IDCANCEL, BUTTON_CLASS, BS_PUSHBUTTON |
      WS_CHILD | WS_VISIBLE | WS_TABSTOP, 118, 138, 36, 24
CONTROL "&Path:", -1, "STATIC", SS_LEFT, 8, 8, 38, 12
CONTROL " &Files", -1, SHADE_CLASS, 1, 4, 34, 74, 67
CONTROL " &Directories", -1, SHADE_CLASS, BSS_GROUP |
      WS_GROUP, 86, 35, 74, 67
CONTROL " Input the rule file name:", -1, SHADE_CLASS,
      BSS_GROUP | WS_CHILD | WS_VISIBLE, 4, 105, 156, 28
CONTROL "", -1, SHADE_CLASS, 2 | WS_GROUP, 0, 134, 165, 2
```

END


```
DLG_START DIALOG 14, 21, 221, 117
CAPTION "startDialog_pupop"
FONT 8, "Helv"
CLASS "BorDlg"
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION |
      WS_SYSMENU
BEGIN
      CONTROL "Initializing the working memory with All instances", 233,
      "BorRadio", BS_AUTORADIOBUTTON | WS_CHILD | WS_VISIBLE
      | WS_TABSTOP, 14, 24, 174, 12
      CONTROL "Initializing the working memory with Selected instances",
      234, "BorRadio", BS_AUTORADIOBUTTON | WS_CHILD |
      WS_VISIBLE | WS_TABSTOP, 14, 40, 192, 10
      CONTROL "Adding Selected instances to the working memory ", 235,
      "BorRadio", BS_AUTORADIOBUTTON | WS_CHILD | WS_VISIBLE
      | WS_TABSTOP, 14, 54, 188, 10
      CONTROL "Start inferencing by:", 236, "BorShade", BSS_GROUP |
      WS_CHILD | WS_VISIBLE | WS_GROUP, 8, 10, 206, 62
      CONTROL "Button", IDOK, "BorBtn", BS_PUSHBUTTON |
      WS_CHILD | WS_VISIBLE | WS_TABSTOP, 42, 88, 32, 20
```

```
        CONTROL "Button", IDCANCEL, "BorBtn", BS_PUSHBUTTON |
    WS_CHILD | WS_VISIBLE | WS_TABSTOP, 128, 88, 32, 20
        CONTROL "", 232, "BorShade", 2 | WS_CHILD | WS_VISIBLE,
    2, 82, 216, 2
END

SD_INPUTDIALOG DIALOG 14, 55, 135, 112
CAPTION "promptDialog_popup"
FONT 8, "Helv"
CLASS "BorDlg"
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION |
    WS_SYSMENU
BEGIN
        LTEXT "", ID_PROMPT, 8, 19, 114, 16, SS_LEFT
        CONTROL "Button", IDCANCEL, "BorBtn", BS_PUSHBUTTON |
    WS_CHILD | WS_VISIBLE | WS_TABSTOP, 49, 83, 32, 20
        CONTROL "Button", IDOK, "BorBtn", BS_PUSHBUTTON |
    WS_CHILD | WS_VISIBLE | WS_TABSTOP, 6, 83, 32, 20
        CONTROL "Help", IDHELP, "BorBtn", BS_PUSHBUTTON |
    WS_CHILD | WS_VISIBLE | WS_TABSTOP, 92, 83, 32, 20
        CONTROL "", 102, "BorShade", 2 | WS_CHILD | WS_VISIBLE,
    1, 74, 127, 2
        CONTROL "", ID_INPUT, "EDIT", WS_CHILD | WS_VISIBLE |
    WS_BORDER | WS_TABSTOP | ES_AUTOHSCROLL, 8, 41, 114, 18
END
```

```
/*********************************************************************
**
**                         C O S M O S
**
**    Name:        ie.h
**    Last version: 7/25/91
**    Author:  Bruno Fromont
**    Supervisor: D. Sriram
**            Copyright@ Intelligent Engineering Systems Laboratory. M.I.T
**
**    -----------------------------------------------------------------
**    Notes: Definition of the inference engine
**
*********************************************************************/


/*********************************************************************
**    This program comes without any warranty as to its performance, or
**    fitness for any particular purpose.
*********************************************************************/


#ifndef IENGINE
#define IENGINE

#include "root.h"
#include "classdec.h" //classdeclare.h

// An action is an instantiation of the RHS of a rule
// (an expression to evaluate(inside aNode) + bindings (inside tok))
class Action :public ObjectB {
 public:
  ActionNode *aNode;
  Token *tok;
  // Constructor
  Action(ActionNode *a,Token *t) {aNode = a ; tok = t;}
};

// The Conflict Set contains:
// - a list of the current actions to be fired (llaction)
// - the strategy to use to select among actions (strategy= RPR or LST)
// - the action which is being executed (the_action)
```

```
class ConflictSet {
  friend ActionNode;
  friend DestroyInConflictSet;
private:
  int strategy;
  Tlist(Action) llaction;
  Action* the_action;
public:
  // methods
  Action *select();
  int fire();
  void set_strategy(int s){strategy = s;}
  // Constructor
  ConflictSet() {llaction=new Tcell(Action);
              strategy = RPR;}
};


// The Ie class contains all the mechanism for the forward-chainer
// and the MORE algorithm.
class Ie{
 private:
  WorkingMemory *wmem;
  ReteNet *rete;
  int cycle_time;
  char *strategy;
 public:
        ConflictSet *cs;
// Constructor
  Ie();
  // methods for end user:
  int  parse(char *);
  void reset();
  void set_strategy(char *);
  void load_wme(root *);
  void load1_wme(root *);
  void run();

  // methods for interfacing with other problem-solvers
  void record_change_object(root *);
  void record_use_rule(char *);
```

```
    // other methods
    char *describe();
    void attach_rete(ReteNet *r){rete=r;}
    void remove_wme(WMelement *);
};


#endif //IENGINE
```

```
# ***********************************************************
#
#      Source File:    ie2.mak
#      Date:                  Wed Jun 08 06:26:38 1992
#      Author:                Ching-Jenq Chiu
#      Supervisor:     D. Wang
#
# ***********************************************************
.AUTODEPEND


#                       *Translator Definitions*
CC = bcc +ie2.cfg
TASM = TASM
TLINK = tlink


#                       *Implicit Rules*
.cpp.obj:
  $(CC) -c {$< }


#                       *List Macros*
Link_Exclude =   "
  ie2.res


Link_Include = "
  ie2.obj ie2.def


#                       *Explicit Rules*
ie2.exe: ie2.cfg $(Link_Include) $(Link_Exclude)
      $(TLINK)  /x/c/Twe/P-
/C/Lc:"BORLANDC"LIB;c:"BORLANDC"CLASSLIB"LIB;c:"BORLANDC"OW
L"LIB @&&|
c0wl.obj+
ie2.obj
ie2
                                        # no map file

owl.lib+
import.lib+
tclasdll.lib+
mathwl.lib+
crtldll.lib+
```

```
cwl.lib
ie2.def
|
  RC ie2.res ie2.exe

#                          *Individual File Dependancies*
ie2.res: ie2.rc
       RC -r -
Ic:"BORLANDC"INCLUDE;c:"BORLANDC"CLASSLIB"INCLUDE;c:"BORLAN
DC"OWL"INCLUDE -FO ie2.res ie2.RC

ie2.obj: ie2.cpp
```

```
;   ***********************************************************
;
;
;    Source File:    ie2.def
;    Date:                 Wed Jun 08 06:26:38 1992
;  Author:         Ching-Jenq Chiu
;  Supervisor:     D. Wang
;
;   ***********************************************************
;
```

```
NAME              ie2
DESCRIPTION       'INFERENCE ENGINE MONITOR'
EXETYPE           WINDOWS
STUB               'WINSTUB.EXE'
CODE              DISCARDABLE PRELOAD
DATA              PRELOAD
HEAPSIZE          4096
STACKSIZE         8192
                  ;Definition file code regeneration bracket
```

(RULE: ProblemDeadBattery1 10
IF
(CLASS: car OBJ: $x
((problem == "unknown") AND
((init_problem == "starting_system") AND
(headlights == "dim")))
)
THEN (
(MODIFY (OBJ:$x
(problem " has a dead battery")
)10000 0.001)
)
COMMENT:"")

(RULE: ProblemDeadBattery2 10
IF
(CLASS: car OBJ: $x
((problem == "unknown") AND
(headlights == "dead"))
)
THEN (
(MODIFY (OBJ:$x
(problem " has a dead_battery")
)13000 0.001)
)
COMMENT:"")

(RULE: ProblemBadIgnition 10
IF
(CLASS: car OBJ: $x
((problem == "unknown") AND
((init_problem == "fuel_or_ignition") AND
((headlights == "working") AND
(spark_plug_spark == "none"))))
)
THEN (
(MODIFY (OBJ:$x
(problem " has a bad ignition system")
)1000 0.001)
)

COMMENT:"")

(RULE: ProblemFuelSystem 10
IF
(CLASS: car OBJ: $x
((problem == "unknown") AND
((init_problem == "fuel_or_ignition") AND
((Fuel_gauge_reading == "full") AND
(carburetor_gas == "yes"))))
)
THEN (
(MODIFY (OBJ:$x
(problem " has a faulty fuel system")
)1000 0.001)
)
COMMENT:"")

(RULE: ProblemNoGas 10
IF
(CLASS: car OBJ: $x
((problem == "unknown") AND
((init_problem == "fuel_or_ignition") AND
(Fuel_gauge_reading == "empty")))
)
THEN (
(MODIFY (OBJ:$x
(problem " is out of gas")
)1000 0.001)
)
COMMENT:"")

(RULE: ProblemBadStarter 10
IF
(CLASS: car OBJ: $x
((problem == "unknown") AND
((init_problem == "starting_system") AND
(headlights == "working")))
)
THEN (
(MODIFY (OBJ:$x

```
(problem " has a bad starter")
)1000 0.001)
)
COMMENT:"")

(RULE: ProblemFloodedEng 10
IF
(CLASS: car OBJ: $x
((problem == "unknown") AND
((init_problem == "fuel_or_ignition") AND
((carburetor_gas == "yes") AND
(spark_plug_spark == "exists") )))
)
THEN (
(MODIFY (OBJ:$x
(problem "flooded_engine")
)1000 0.001)
)
COMMENT:"")

(RULE: ProblemFuelIgn 10
IF
(CLASS: car OBJ: $x
((problem == "unknown") AND
((init_problem == "unknown") AND
((ignition_key == "on") AND
(engine_turning_over == "yes") )))
)
THEN (
(READ PROMPT: "how are your headlights (working/dim/dead) ?", VAR:$h,
TYPE: s)
(READ PROMPT: "and spark plug spark (none/exists)?", VAR:$s, TYPE: s)
(READ PROMPT: "status of your fuel gauge (full/empty):", VAR:$g, TYPE: s)
(READ PROMPT: "smell on your carburetor (yes/no) ?", VAR:$carbu, TYPE:
s)
(MODIFY (OBJ:$x
(init_problem "fuel_or_ignition")
(headlights $h)
(spark_plug_spark $s)
(carburetor_gas $carbu)
```

```
(Fuel_gauge_reading $g)
)1000 0.001)
)
COMMENT:"")

(RULE: ProblemStartSystem 10
IF
(CLASS: car OBJ: $x
((problem == "unknown") AND
((init_problem == "unknown") AND
((ignition_key == "on") AND
(engine_turning_over == "no") )))
)
THEN (
(READ PROMPT: "what about your headlights (working/dim/dead)?", VAR:$h,
TYPE: s)
(MODIFY (OBJ:$x
(init_problem "starting_system")
(headlights $h)
)2000 0.001)
)
COMMENT:"")

(RULE: Start1 20
IF
(CLASS: COSMOS_START OBJ: $x
(init_status == 1)
)
THEN (
(MAKE (CLASS:car   OBJ:Acar
(problem "unknown")
(car_make "HONDA")
(ignition_key "off")
))
(MAKE (CLASS:mechanic   OBJ:Amec
(first_name "albert")
))
(MODIFY (OBJ:$x
(init_status 2)
)1000 0.001)
```

```
)
COMMENT:"")

(RULE: Start 20
IF
((CLASS: car OBJ: $x
((problem == "unknown") AND
((car_make == $mak) AND
(ignition_key != "on") ))
)AND
(CLASS: mechanic OBJ: $m
(first_name == $name)
))
THEN (
(DISPLAY "TheCar.dmp")
(PRINT "Hello sir, my name is ", $name, "."n")
(PRINT "Please turn on the key of your ", $mak, "."n")
(READ PROMPT: "Is the engine turning over (yes/no)? ", VAR:$ans, TYPE:
s)
(MODIFY (OBJ:$x
(init_problem "unknown")
(ignition_key "on")
(engine_turning_over $ans)
)2000 0.001)
(MODIFY (OBJ:$m
(job "grad student")
)2000 0.001)
)
COMMENT:"")

(RULE: FinalDiagnostic 20
IF
((CLASS: car OBJ: $c
(problem != "unknown")
)AND
(CLASS: car OBJ: $c
(problem == $problem)
))
THEN (
(PRINT "sir, I guess your car ", $problem)
```

```
)
COMMENT:"")

(RULE: ProblemNoGas 10
IF
(CLASS: car OBJ: $x
((problem == "unknown") AND
((init_problem == "fuel_or_ignition") AND
(Fuel_gauge_reading == "empty") ))
)
THEN (
(MODIFY (OBJ:$x
(problem " is out of gas")
)1000 0.001)
)
COMMENT:"")
```

BIBLIOGRAPHY

[1] Black, W. J. *Intelligent Knowledge Based Systems*. UK: Van Nostrand, 1986.

[2] Bridgeland, D., and L. Lafferty. "Scavenger: an Experimental RETE Compiler." *SPIE* vol. 635, Applications of Artificial Intelligence III, 1986.

[3] Brownston, L., R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Massachusetts: Addision-Wesley, 1985.

[4] Cosmos Group, "COSMOS: Design Document." *IESL Memo.* MIT, 1990

[5] Davis, R. "Interactive Transfer of Expertise: Acquisition of New Inference Rules." *Artificial Intelligence* (1979): 121-157.

[6] Ellis, M., and B. Stroustrup. "Annotated C++ Reference Manual." Massachusetts: Adddison-Wesley, 1990.

[7] Efstathiou, J. "Non-classical Logics and the Handing of Uncertainty." *British Computer Society Specialist Group on Expert Systems Newsletter*, 10, 1984.

[8] Forgy, C. "RETE: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem." *Artificial Inteligence* 19, 1982.

[9] Forsyth R. *Expert Systems, Principles and Case Studies* London: Chapman and Hall , 1984.

[10] Johnson E. and K. Reichard. "Power Programming MOTIF", Oregon: Management Information Source, 1991.

[11] Koen, B. V. "Toward a Definition of the Engineering Method", *The Bent*, Spring (1985):28-33.

[12] Lindsay, Robert K., B. G. Buchanan, E. A. Feigenbaum, and J. Lederberg. "Applications of Artificial Intelligence for Organic Chemistry." *The DENDRAL Project.* New York: McGraw-Hill, 1980.

[13] Lippman, Stanley B. "C++ Primer" Massachusetts: Addison-Wesley, 1989.

[14] MACSYMA group "The MACSYMA Reference Manual." Cambridge, Mass.: Computer Science Dept., Massachusetts Institute of Technology, 1974.

[15] McCord J. W. *Developing Windows Applications with Borland C++3.* Indiana: SAMS 1992

[16] Norton P., and P. Yao. *Borland C++ Programming for Windows.* New York: Bantam , 1992

[17] Reboh, R. "Knowledge Engineering Techniques and Tools in the PROSPECTOR Environment." Technical Report No. 243, SRI International, Menlo Park, CA, 1981.

[18] Sriram, D. *Computer-Aided Engineering: The Knowledge Frontier.* To be published, IESL, Massachusetts Institute of Technology, 1992.

[19] Stefik, M. "The Organisation of Expert Systems." *A Tutorial, Artificial Intelligence.* Vol. 18 No. 2, 1982.

[20] Stroustrup, B. *The C++ Programming Language.* Massachusetts: Addison-Wesley, 1986

[21] Swartout, W. "Explaining and Justifying Expert Consulting programs." *IJCAI* 7, (1974): 815-822.

[22] Young, D. *The X Window System Programming and Applications with Xt OSF/MOTIF Edition.* New Jersey: Prentice-Hall, 1990