EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Eindhoven University of Technology

MASTER

Simplexys for windows

de Bruin, J.L.M.

Award date:
1997

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

7264

Eindhoven University of Technology
Faculty of Electrical Engineering
Division of Medical Electrical Engineering

Simplexys for Windows

by J.L.M. de Bruin

This report was submitted in partial fulfilment of the requirements for the degree of Master of Electrical Engineering at the Eindhoven University of Technology.
The work was carried out from April 1994 until February 1995 under responsibility of prof.dr.ir. J.E.W. Beneken and under supervision of dr.ir. J.A. Blom.

The department of Electrical Engineering cannot be held responsible for the contents of training and graduation reports.

Summary

This report discusses the development of a Windows version for Simplexys, an expert system toolbox.

During the last eight years Simplexys has been developed at the Division of Electrical Engineering of the Eindhoven University of Technology. This toolbox consists of a programming language, a rule compiler, an inference engine (reasoning mechanism) and some debugging tools. To assist the programmer in using the toolbox, a user interface is available. The tools and the user interface are designed for a DOS environment.

Graphical environments become more and more popular. For DOS based machines Windows has become the most popular one. Due to this rising popularity, a next step in the development of Simplexys is the creation of a Windows version.

To fulfill this task, first some research was done into 'What is Windows'. This research showed the essential ideas behind Windows and provided useful programming techniques. This knowledge is used to develop a Windows interface for controlling the elements of the toolbox. Also some chances are made in the existing tools to be compatible with Windows.

The Windows version of Simplexys was successfully developed. The created Windows interface resembles other Windows applications, which makes it easy to use. To be flexible, the Windows interface allows the creation of DOS applications too.
Because Windows provides tools for easy data exchange and handling multimedia, the applicability of Simplexys will increase.

Contents

1 Introduction .... - 1 -

2 SIMPLEXYS, an expert system toolbox .... - 3 -

2.1 The Simplexys language .... - 3 -

2.2 The rule compiler .... - 5 -

2.3 The semantics checker .... - 6 -

2.4 The protocol checker .... - 6 -

2.5 The option builder .... - 6 -

2.6 The Simplexys inference engine .... - 7 -

2.7 The simulate and explain facility .... - 7 -

2.8 Programming a Simplexys expert system .... - 8 -

3 Windows .... - 9 -

3.1 Some Windows Essentials .... - 9 -

3.1.1 The graphical user interface (GUI) .... - 9 -

3.1.2 Windows and Multitasking .... - 10 -

3.1.3 Programming within Windows .... - 11 -

3.1.4 The message-driven architecture of Windows .... - 12 -

3.1.5 Windows and Dynamic-Link Libraries .... - 14 -

3.2 A sample application .... - 15 -

3.2.1 A Windows Skeleton .... - 15 -

3.2.2 A window-handling routine .... - 19 -

3.2.3 The Resource WorkShop .... - 22 -

3.2.4 The 'About Box' .... - 23 -

3.2.5 The main part .... - 24 -

3.3 Adding a Help system .... - 25 -

3.3.1 Designing Help topics .... - 26 -

3.3.2 Access to Help .... - 26 -

3.3.3 Creating the Help topic files .... - 29 -

4 The Windows version of Simplexys . . . . - 31 -
4.1 The Shell . . . . - 31 -
4.1.1 Error handling . . . . - 36 -
4.1.2 Introduced routines and variables . . . . - 38 -
4.2 The Expert System . . . . - 40 -
4.2.1 The DOS-version . . . . - 40 -
4.2.2 The Windows-version . . . . - 40 -
4.2.3 Memory usage . . . . - 44 -
4.3 The Simulate and Explain facility . . . . - 46 -
4.3.1 The DOS-version . . . . - 46 -
4.3.2 The Windows-version . . . . - 47 -
4.3.3 Changes in the DLL . . . . - 49 -
4.3.4 Memory usage . . . . - 52 -
4.4 New programming elements . . . . - 53 -

5 Conclusions and recommendations . . . . - 55 -

Appendix A
References . . . . - 57 -

Appendix B
List of files . . . . - 59 -

1 Introduction

At the Division of Medical Electrical Engineering an expert system toolbox, called Simplexys, has been developed. Because Simplexys is mainly designed for monitoring tasks in medical engineering where efficiency and performance are required, key goals during development were execution speed, compactness and correctness.

The toolbox consists of a rule compiler, a semantics and protocol checker, an inference engine (reasoning mechanism) and some debugging tools. To be flexible, also an interface to existing programming languages is available. To assist the programmer in controlling these tools, a user interface was created. The just mentioned tools run in a DOS environment.

During the last few years, graphical environments have become more and more popular. For DOS-based machines, Windows has become the most popular environment. To be able to develop Simplexys applications for Windows, the toolbox needed to be adapted to this environment.

This report will first discuss the elements of the Simplexys toolbox. In chapter 3 the graphical environment Windows is presented. The ideas behind Windows are introduced and some programming techniques are explained. Chapter 4 presents the created user interface and the changes made to the existing tools. Finally chapter 5 gives some conclusions and remarks.

2 SIMPLEXYS, an expert system toolbox

The SIMPLEXYS toolbox consists of a collection of tools that assist in the design of expert systems. The purpose of the expert systems toolbox is to develop computer programs that are capable of 'human reasoning' or 'thinking'. The expert system evaluates input data and intermediate results using the implemented knowledge. After evaluation, conclusions can be derived. In the following sections, the tools of SIMPLEXYS will be described. A thorough treatment can be found in [BLOM, 1990].

2.1 The Simplexys language

The Simplexys language is designed to formally describe human knowledge. It is a superset of the Pascal programming language. This offers an intuitive interface to Pascal procedures, that can be used for data acquisition or to display results. Programs written in this language are called 'knowledge bases' and describe the domain knowledge available in the expert system. The simplicity of the language makes the programs easy to understand, even for non-expert system programmers.

Simplexys is based on a three-valued logic. A rule's conclusion can have the value TR, the conclusion is true, FA, the conclusion is false or PO, the conclusion is unknown (there is no information available that allows to decide whether the value is true or false). This type of logic is introduced to have a better approximation of human reasoning.

An essential part of a knowledge base is the RULES section. This part contains the rules of the knowledge base, where each rule represents a 'chunk' of knowledge.

Within Simplexys five primitive rule types are available:

1 FACT rules, that denote constants.

2 ASK rules, that ask questions from the user.

3 TEST rules, that test externally supplied data.

4 MEMO rules, that remember results.

5 STATE rules that denote the current context.

To combine the results of other rules, another rule type is available:

EVAL rules, that calculate higher level conclusions given an expression that combines other rules.

The rules defined in the knowledge base are connected into a semantic network. This collection of nodes and links, represented by the rules and the interrelation between several rules, connects the 'chunks' of knowledge.

Within Simplexys rules are evaluated only once, in a recursive way. To determine the value of EVAL rules, first the values of the rules that define the EVAL rule must be obtained. This process repeats itself, until the primitive rules are reached.

To specify that a certain action has to be taken if a rule gets a matching conclusion, a construction with THEN, ELSE or IFPO (if possible) is used. This kind of construction is called a THELSE.

To evaluate expressions, Simplexys uses a logic that is very much like boolean logic. Simplexys expressions consists of two entities, propositions and operators. Propositions are indicated by a name (p, q, animal, etc) and operators are indicated by special symbols (not, and, or, etc.). The operators are monadic (having one argument) or dyadic (having two arguments). Parentheses can be used to form sub-expressions.

In Figure 2.1 a small example is shown. To obtain a conclusion for rule 1, first the primitive rules 2 and 3, and rule 4 have to be evaluated. Rule 4 gets its value by evaluating primitive rules 5 and 6. The semantic network is shown in Figure 2.2.

```
Rule1: 'Rule text 1'
Rule2 and Rule3 and Rule4

Rule4: 'Rule text 4'
Rule5 and Rule6
```

Figure 2.1 Sample rules

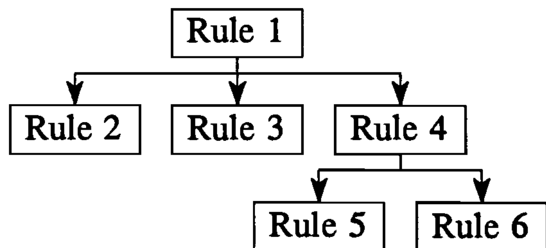


Figure 2.2 Semantic network of rules

Another essential section of the knowledge base is the PROCESS section. In this section the dynamics of the process are described. All state transitions or ON-statements are inserted here. ON-statements are used to define context switches.

2.2 The rule compiler

The Rule Compiler translates the knowledge base into an internal representation that can be processed by the inference engine. To prevent an enormous amount of searching through the knowledge base at run time, searching and matching in SIMPLEXYS is done by the compiler. The internal representation consists of a set of tables, in which the relations between the rules are represented by pointers. At execution time, these tables can be traversed at high speed.

To represent the information, the rule compiler creates six qqq-files:

- rinfo.qqq: containing the arrays and tables used to represent the rules and their mutual connectivity
- rtest.qqq: containing the test sections, used in the test rules
- rhist.qqq: containing information about the history
- rdodo.qqq: containing the DO sections
- rinex.qqq: containing the initialization and exit sections
- ruses.qqq: containing the Turbo Pascal units used in the knowledge base.

Besides this, the rule compiler checks the knowledge base for some semantic and syntactic errors and generates warnings for incorrect or questionable higher level constructions that cannot be checked at compile time.

2.3 The semantics checker

The semantics checker performs several semantics checks on the file rinfo.qqq. If errors are detected, appropriate error messages are created. The semantics checker is a powerful tool for partially proving correctness. Semantics errors, that can be detected, are self-referencing evaluations, loops in thelses, conflicting thelses, thelses to successors and predecessors, and unconnected non-STATE rules.

The semantics checker runs without user interaction. It provides additional checks, besides those performed by the rule compiler. Running the program is not always necessary and can be skipped if the structure of the knowledge base remains unmodified.

2.4 The protocol checker

The protocol checker is used to detect errors in the PROCESS description of the knowledge base. The ON-statements define the protocol. The protocols are translated into Petri nets, that are ideally suited to describe systems in which concurrent events can occur. The tests performed by the protocol checker are quite extensive and cover syntax, topology, connectedness and dynamic errors.

Just like the semantics checker, the protocol checker performs additional test not performed by the rule compiler, runs without user interaction and may be skipped if the structure of the protocol in the knowledge base remains unmodified.

2.5 The option builder

The option builder is used to determine a number of runtime options. The options are used in the inference engine. Options that can be set are about:

Time: It is possible to choose between real time or simulated time. The option 'simulated time' lets the user specify the time between two successive runs of the final system.

Debugging: This option makes it possible to see intermediate results of the inferencing process, which can be very helpful during design and testing phases.

Dump results: This option makes it possible to store all results to disk to investigate the results of the inferencing process at leisure. During design and testing phases, this can be helpful.

2.6 The Simplexys inference engine

The inference engine is the experts system's reasoning mechanism. The purpose of the inferencing process is to derive all possible conclusions (or goals) about data offered to the system. Generally, the data will change over time, and the analysis will need to be repeated. Each evaluation of all the goals is called a run.

The Simplexys inference engine is written entirely in Pascal. Compiling the qqq-files and the inference engine into one executable program, a ready to run expert system is built. The Pascal compiler is used to check for errors in the Pascal code of the knowledge base.

The created expert system is able to 'reason' by itself, using the knowledge specified in the knowledge base. The experts system starts with evaluating STATE rules. At the end of a run, when all the goals are evaluated, the ON-statements are executed, possibly causing a context switch. As long as at least one STATE rule is TR, a new run is started.

2.7 The simulate and explain facility

The design of a correct knowledge base is difficult. During the design process, testing and debugging are the most difficult parts, because experts and knowledge engineers want to know how the expert system came to a decision. To perform these tasks, some debug tools have been developed.

The simulate facility provides a tool to 'trace' through the inferencing process. The infor-

mation, obtained during the inferencing process, is stored to disk. The simulate facility traces this information and makes it possible to visualize the information. The explain facility can be used for examination of the evaluation structure of the process. It shows the links between rules graphically.

2.8 Programming a Simplexys expert system

To develop an expert system with the Simplexys expert system toolbox, the following steps must be taken:

1. Acquire the knowledge to be implemented.
2. Implement the knowledge using the SIMPLEXYS language. The result of this process is a 'knowledge base' or 'knowledge program'. If the knowledge cannot be formalized, go back to step 1.
3. Generate the internal format of the knowledge. This translation from knowledge base to internal representation is done through compilation by the rule compiler. Also some additional syntactic and semantic checks can be performed. If errors or contradictions are signalled, go back to step 2 resp. step 1.
4. Select any runtime debug options.
5. Compile the expert system. The Simplexys expert system and the output files, generated by the rule compiler, are combined by the Pascal Compiler. This produces a ready to run expert system.
6. Run the expert system.
7. Correct imperfections and errors by repeating steps 1 to 5.

Whether an expert system is 'finished' is hard to say, because some errors or limitations will only show up in the long term and the system will never be 'complete'. Practically, the system is finished, when it is sufficiently 'useful'.

3 Windows

Since its introduction in 1985 Microsoft Windows has emerged as the most popular graphical user interface environment for DOS-based computers. Windows provides a multitasking (see 'Windows and Multistasking'), graphical-based windowing environment that runs programs especially designed for Windows. The goal of Windows is to enable a person who has basic familiarity with the system to sit down and run virtually any application without prior training. Learning a new program can be restricted to *what* the program does, not *how* the user must interact with the program.
What Windows is depends, to some extent, upon whether you are an end user or a programmer. For the end user, it is a shell with which he or she interacts to run applications. For the program developer, Windows provides a collection of API (Application Program Interface) functions that allow the use of menus, dialog boxes and other components of a friendly user interface. Windows also contains an extensive graphics programming language.

3.1 Some Windows Essentials

3.1.1 The graphical user interface (GUI)

The concepts behind the graphical user interface date form the mid-1970s, with the pioneering work done at the Xerox Palo Alto Research Center (PARC). The work done at Xerox PARC was brought into the mainstream and popularized by Apple Computer, Inc, with their introduction of the Macintosh. Since the introduction of the Macintosh, graphical user interfaces have become more and more popular.

Graphical user interfaces make use of graphics on a bitmapped video display. Graphics provides better utilization of screen real estate, a visually rich environment for conveying information. In earlier days, the video display was used solely to echo text that the user

typed using the keyboard. In a GUI, the video display is used interactively. The video display shows various graphical objects in the form of icons, which are small symbols used to represent functions or programs, and input devices, such as buttons and scroll bars. The theory behind the use of graphical objects is found in the old adage:

'a picture is worth a thousand words.'

Using the mouse or the keyboard, the user can directly manipulate these objects on the screen. The interaction between the user and a program becomes more intimate. Rather than the one-way cycle of information from the keyboard to the program to the video display, the user directly interacts with the objects on the display.

The basic idea behind a window-based user interface is to provide the equivalent of a desktop on the screen. On a desk one may find several different pieces of paper, one on top of another, often with fragments of different pages visible beneath the top page. The equivalent of the desktop in Windows is the screen. The pieces of paper are the windows on the screen. Selecting a window and making it current, which means to put the window on top of all the other windows, is like moving the pieces of paper on the desktop. In short, it is possible within Windows to control the surface of the screen the way you control the surface of your desktop.

3.1.2 Windows and Multitasking

Windows is a multitasking operating system. As multitasking system, it is somewhat unique in that it uses *non-preemptive multitasking.* Non-preemptive multitasking means that each program running in the system retains use of the processor until it relinquishes it.

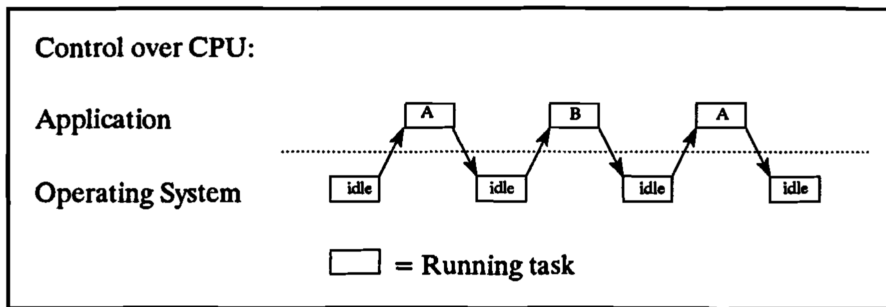


Figure 3.1 Non-preemptive multitasking

This differs radically from the type of multitasking done by other operating systems that employ preemptive task switching based upon time slices. In preemptive task switching, the operating system simply stops executing one program and moves on to the next, in a cyclic way.

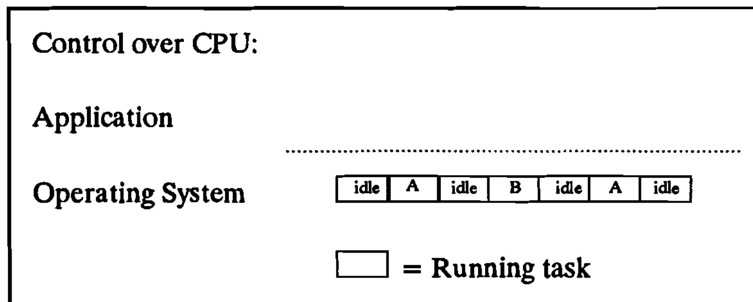


Figure 3.2 Preemptive multitasking

So one of the most important rules that a Windows program must follow is to return control to Windows when it is inactive. This allows Windows to grant the processor to another task. If this is not done consequently, it is possible for a program to monopolize the processor, effectively halting all other tasks[1].

3.1.3 Programming within Windows

Programs written for Windows do not directly access the hardware such as the graphics display or the I/O ports. Windows virtualizes the hardware. A program written in Windows will run with any hardware, such as video boards or printers, for which a Windows device driver is available. The program does not need to determine what type of device is at-

[1]It is announced that in the new version of Windows, 'Windows 95', preemptive multitasking is implemented, which prevents these problems.

tached to the system, as is shown in Figure 3.3. Windows performs this task. So the programs written in Windows will run on almost every system, independent of the hardware installed.

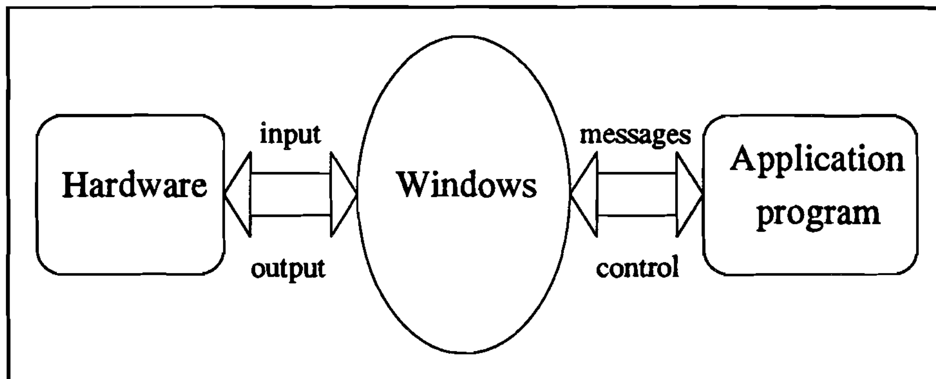


Figure 3.3 Position of Windows

For programming graphics, Windows includes a graphics programming language (called the Graphics Device Interface, GDI) that allows the easy display of graphics and formatted text.

3.1.4 The message-driven architecture of Windows

When starting to write a Windows application, one must bear in mind that the application will not have full control over the machine all the time. Instead, it is given control by Windows a great number of times. In particular, Windows has control during the time the machine is waiting for a user response. Communication between Windows and an application is performed by a message-driven architecture. It is Windows that calls your application. A Windows application waits until it is sent a message by Windows. The message-driven architecture is presented in Figure 3.4. If any kind of input occurs, Windows receives this input. Within Windows this input is translated into a message.

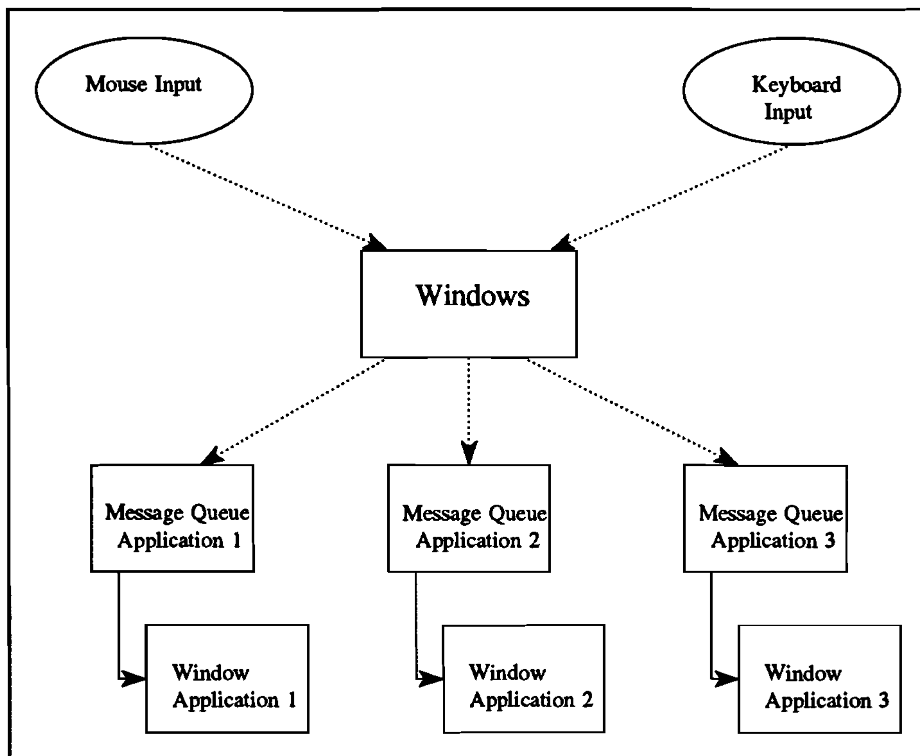


Figure 3.4 Message driven architecture

This message is passed to the application through a special function that is called by Windows. The messages sent to an application are stored in a message queue associated with the application until they can be processed. This prevents that messages will be lost because an application is busy processing another message. The message will wait in the queue until the application is ready for it. After receiving the message, the application is expected to take appropriate action. It is the message-based interaction with Windows that dictates the general form of all Windows programs.

This architecture is different from a conventional program, running under DOS, where a main part exists that is called once and does not return until program execution terminates. Any functions or procedures in such a program are called directly by the main part or by other functions. It is the program that requests such things as input and output.

3.1.5 Windows and Dynamic-Link Libraries

A Dynamic-Link Library (DLL) is an executable module containing code or resources (see 'The Resource Workshop') for use by other applications or DLLs. Conceptually, a DLL is similar to a unit in Pascal: both have the ability to provide services in the form of procedures and functions to a program. There are, however, many differences between DLLs and units. In particular, units are statically linked, whereas DLLs are dynamically linked. DLLs permit several Windows and DOS protected-mode applications to share code and resources.

When a program uses a procedure or function from a unit, a copy of that procedure or function's code is statically linked into the program's executable file.
In contrast to a unit, the code in a DLL is not linked into the program that uses the DLL. Instead a DLLs code and resources are in a separate executable file with a .DLL extension. This file must be present when the client program runs. The procedure and function calls in the program are dynamically linked to their entry points in the used DLLs. Another difference between units and DLLs is that units can export types, constants, data and objects whereas DLLs can export procedures and functions only.

A DLL does not have to be written in the same language as the calling application. This makes DLLs ideal for multi-language programming projects. Using DLLs also economizes the memory usage, because the DLL is loaded into memory only when it is needed.

For a module to use a procedure or function in a DLL, the module must import the function using an external declaration. In imported procedures and functions, the **external** directive takes the place of the declaration and statement parts that would otherwise be present. Imported procedures and functions use the far call model, but otherwise they behave no differently than normal procedures and functions.

3.2 A sample application

To illustrate the essential steps in the creation of a Windows application, a sample program is used. The sample program creates a window like Figure 3.5 and uses the menu structure of Figure 3.6.

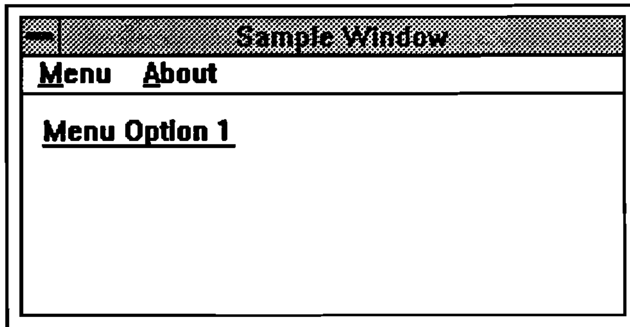


Figure 3.5 Output window of example program

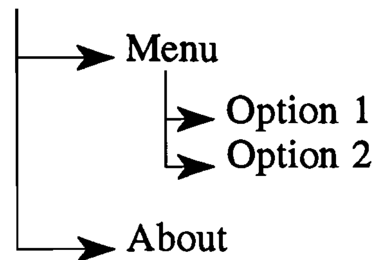


Figure 3.6 Menu structure

The available options are:

- Option 1: show the text 'Menu Option 1'
- Option 2: show the text 'Menu Option 2'
- About: show the 'About box'

The following sections describe the program parts, necessary to run the sample program.

3.2.1 A Windows Skeleton

All Windows programs must perform some general steps. These steps are:

1. Define a window class
2. Register that class with Windows
3. Create a window of that class
4. Display the Window
5. Begin running the message loop

When a Windows program first begins to execute, it will need to define and register a

window class. Registering the window class tells Windows about the form and function of the window. However, registering the window class does not cause a window to come into existence. For registering a procedure named *Register* is introduced[1].

```
{---Register-------------------------------------------------------}
PROCEDURE Register(P: Pointer; Name: PChar; Menu: PChar);
VAR
  WndClas: TWndClass;                 { Structure for class information }

BEGIN
  IF hPrevInst <> 0 THEN Exit;{ Check if instance is already available }
  WndClas.Style := CS_HReDraw OR CS_VReDraw;{ Window will be repainted }
                                         { if size is changed        }
  WndClas.lpfnWndProc:= P;            { Set pointer to handling routine }
  WndClas.cbClsExtra := 0;            { No additional allocation needed }
  WndClas.cbWndExtra := 0;            { No additional allocation needed }
  WndClas.hInstance := HInstance;           { Define handle to Instance }
  WndClas.hIcon := LoadIcon(0, Idi_Application); { Define default icon }
  WndClas.hCursor := LoadCursor(0, Idc_Arrow); { Define default cursor }
  WndClas.hbrBackground := GetStockObject(White_Brush);
                                        { Define default background }
  WndClas.lpszMenuName := Menu;               { Define menu structure }
  WndClas.lpszClassName := Name;                  { Define class name }
  IF NOT RegisterClass(WndClas) THEN
  BEGIN
    MessageBox(GetFocus, 'Can not Register Class', 'Error ', MB_OK);
    Halt;
  END;
END;
{---Register END---------------------------------------------------}
```

Figure 3.7 Procedure Register

The procedure needs three parameters. The meaning of the parameters is as follows:

P: Points to a window function. This is the function that handles all the input and output of the window class to be defined.

Name: Points to a null-terminated string that specifies the name of the window class.

Menu: Points to a null-terminated string that specifies the resource name of the class menu (as the name appears in the resource file). The resource file is defined with the Resource Workshop of Borland Pascal, which will be discussed later.

[1]All the code parts presented in this report are written in Borland Pascal

When the procedure is called, the registration process is started. The procedure initiates a window class with some common features and provides each window class with a pointer to a routine that will handle all messages delivered to windows of that class.

After defining and registering a window class, the application can create a window of that class using the *Create* function.

```
{---Create-----------------------------------------------------------}
FUNCTION Create(Name:PChar; Style:Longint; X1,Y1,Width,Height:Integer;
                Parent: Word): Word;
VAR
  Wnd: Word;                                       { Handle to window }
BEGIN
  Wnd := CreateWindow(Name, Name, Style, X1, Y1, Width, Height,
                      Parent, 0, hInstance, nil);     { Create window }
  ShowWindow(Wnd, SW_ShowNormal);                { Make window visible }
  UpDateWindow(Wnd);                                 { Repaint window }
  Create := Wnd;                             { Give handle to function }
END;
{---Create END-------------------------------------------------------}
```

Figure 3.8 Function Create

To call the function *Create* seven parameters are needed. These parameters are:

Name: Points to a null-terminated string specifying the window class. It must be a name registered with the *Register* procedure. This string also represents the initial window name.

Style: The style of the window to be created. This parameter can be a combination of the window styles and control styles. Some often used styles are:
WS_HSCROLL (a window with a horizontal scroll bar),
WS_MAXIMIZEBOX (a window with a maximize button),
WS_OVERLAPPED (a window with a title and a border),
WS_SYSMENU (a window with a system-menu box in its title bar).

X1: The initial X-position of the window. If this value is CW_USEDEFAULT, Windows selects the default position for the window's upper-left corner and ignores the y parameter.

Y1: The initial Y-position of the window.

Width: The width, in device units, of the window. If width is CW_USEDEFAULT, Windows selects a default width and height for the window (the default width extends from the initial X-position to the right

edge of the screen, and the default height extends from the initial Y-position to the top of the icon area). If the width is set to CW_USEDEFAULT, Windows ignores Height.

Height: The height, in device units, of the window.

Parent: The parent or owner window of the window being created. A valid window handle must be supplied when creating a child window or an owned window. An owned window is an overlapped window that is destroyed when its owner window is destroyed, hidden when its owner is minimized, and that is always displayed on top of its owner window.

The function creates the window, shows it at the given position and returns a number that can be used for referencing the created window.

Because Windows communicates with a program by sending it messages, all Windows applications must establish a *message loop*. This loop reads any pending message from the application's message queue and then dispatches that message back to Windows, which calls the program's window function with that message as a parameter. This may seem to be an overly complex way of passing messages, but it is the way that all Windows programs must function. A reason for this is scheme is to force the application to return control to Windows from time to time, thus supporting Windows' non-preemptive multitasking. The task of a message loop is performed by the procedure *Loop* and presented in Figure 3.9 .

```
{---Loop-----------------------------------------------------------}
PROCEDURE Loop;
VAR
  Msg: TMsg;
BEGIN
  WHILE GetMessage(Msg, 0, 0, 0) DO
  BEGIN
    TranslateMessage(msg);
    DispatchMessage(msg);
  END;
END;
{---Loop END-------------------------------------------------------}
```

Figure 3.9 Procedure Loop

The procedure *Loop* initiates the message loop, which revolves around three functions:

GetMessage(), TranslateMessage () and DispatchMessage ();

GetMessage receives an event message from the Windows kernel. The message is passed to TranslateMessage(), where any key-down messages are translated to their appropriate ASCII equivalents. Finally the message is handed to DispatchMessage (), which calls the window's associated handling routine, and passes in detailed information about the message as arguments.

3.2.2 A window-handling routine

A window-handling routine is built around a case statement. The routine is called by the procedure *Loop* and the posted message is passed as a parameter. A typical Windows event message is composed of four fields:

- a 16-bit handle to a Window, that points to the window the message belongs to
- a 16-bit message type, which identifies the kind of message
- a 16-bit and a 32-bit parameter, that contain data specific to the message.

Within the case statement, the messages which need a specific action, are stated. The handling routine for the sample program is shown in Figure 3.10.

The routine shows some frequently used messages.

wm_Command: This message is sent to a window when the user selects an item from a menu, when a control sends a notification message to its parent window, or when an accelerator keystroke is translated. WParam specifies the control or menu item identifier.

wm_LButtonDown: This message is sent when the user presses the left mouse button. The horizontal position of the cursor is given by LOWORD(lParam) and the vertical position by HIWORD(lParam).

wm_Move: This message is sent after a window has been moved. The new x-position is (int) LOWORD(LParam) and the new y-position is (int) HIWORD(LParam).

wm_Paint: This message is sent when Windows or an application makes a request to repaint a portion of an application's window.

```
{---HandlingRoutine-----------------------------------------------------}
FUNCTION HandlingRoutine(Window: HWnd; Message, WParam: Word;
                         LParam: Longint): Longint; EXPORT;

CONST
  X = 10;                    { initial X position }
  Y = 10;                    { initial Y position }

VAR
  DC  : HDC;                 { handle type for device context handles }
  PS  : TPaintStruct;        { structure that can be used to }
                             { paint the client area of the window }
  AboutProc : TFarProc;      { Pointer to a procedure }

BEGIN
  HandlingRoutine := 0;
  CASE Message OF
    wm_Command:
      BEGIN
        CASE WParam OF
          idm_MenuOption1:                         { Option 1 }
            BEGIN
              StrCopy (TextArray, 'Menu Option 1');
              InvalidateRect (Window, NIL, TRUE);
            END;
          idm_MenuOption2:                         { Option 2 }
            BEGIN
              StrCopy (TextArray, 'Menu Option 2');
              InvalidateRect (Window, NIL, TRUE);
            END;
          idm_About:                               { About }
            BEGIN
              AboutProc := MakeProcInstance(@About, HInstance);
              DialogBox(HInstance, 'AboutBox', Window, AboutProc);
              FreeProcInstance(AboutProc);
            END;
        END;
      END;
    wm_LButtonDown:                        { Left mouse button clicked }
      Exit;
    wm_Move:                               { Window moved }
      Exit;
    wm_Paint:                              { Repaint of Window }
      BEGIN
        DC := BeginPaint (Window, PS);

        TextOut (DC, X, Y, TextArray, StrLen (TextArray));
        MoveTo  (DC, X, Y+15); LineTo  (DC, X+95, Y+15);

        EndPaint (Window,PS);
      END;
    wm_Destroy:                            { Quit application }
      BEGIN
        PostQuitMessage(0);
        Exit;
      END;
  END;
  HandlingRoutine := DefWindowProc(Window, Message, WParam, LParam);
END;
{---HandlingRoutine END-------------------------------------------------}
```

Figure 3.10 A sample window-handling routine

wm_Destroy This message is sent when a window is being destroyed. It is sent to the window procedure of the window being destroyed after the window is removed from the screen. This message is sent first to the window being destroyed and then to the child windows as they are destroyed.

ad wm_Command:
To start the program, it is assumed that a resource file is available with some menu options (MenuOption1, MenuOption2 and About). Depending on the menu option last chosen the text printed on the screen will be 'Menu Option 1' or 'Menu Option 2'. More information about creating a menu structure can be found in the section 'The Resource Workshop'.
If About is chosen, a dialog box is presented with some information about the program. The dialog box needs its own window-handling routine. To initiate the dialog box, three lines are executed. In the first line, AboutProc gets the pointer to the window-handling function, called About. HInstance identifies the instance associated with the desired data segment. The second line initiates the dialog box. The dialog box is called 'AboutBox' (defined in the used resource file). It is a child window of the active window and the window-handling routine is referenced by AboutProc. The instance is referred by HInstance.

ad wm-Paint:
If this message is sent a device context DC is defined. In this case, the device context is a display context. The specified window is prepared for painting and the structure PS is filled with information about the painting. Within this display context, one can write to screen (TextOut) or draw graphics (LineTo).

ad wm_Destroy:
This message executes the command PostQuitMessage (0). The command posts a message to Windows indicating that an application is requesting to terminate execution (quit).

3.2.3 The Resource WorkShop

The Resource Workshop is used to define resources, such as menu's and dialog boxes (Menu's are lists of commands the user can choose from and dialog boxes give the user a way to interact with an application.). Resources contain data that define the visible portions of a Windows program. The data is stored in the program's executable (.EXE) file, but separately from the program's normal data segment. Resources are designed and specified outside the program code, then added to the program's compiled code to create a program's executable file. Within the Resource Workshop it is possible to create the resources interactively.

The resource file can be stored in two ways. As a file with the extension .rc, containing ASCII text, or as file with the extension .res, containing a compiled version of the ASCII text. To create an executable application, a compiled version of the resource file is needed.

To define the menu structure of the sample program, the resource script of Figure 3.11 is used. The menu structure is graphically designed and the script is created by 'The Resource Workshop'. Saving the menu structure with the extension .res creates the compiled resource file.

```
SAMPLEMENU MENU
BEGIN
     POPUP "&Menu"
     BEGIN
          MENUITEM "Option &1", 100
          MENUITEM "Option &2", 101
     END

     MENUITEM "&About", 110
END
```

Figure 3.11 Resource script menu structure

Because Windows application's resources are separate from the program code, it is possible to make significant changes to the interface without even opening the file that contains the program code. It is also allowed for different applications to share the same set of resources. To include a specific resource file in the Pascal code, the {$R *FileName*} directive is used. *FileName* refers to the resource file to be used.

3.2.4 The 'About Box'

The window-handling function of the dialog box is presented below.

```
{---About----------------------------------------------------------}
FUNCTION About(Dialog: HWnd; Message, WParam: Word;
           LParam: Longint): INTEGER; EXPORT;
BEGIN
  CASE Message OF
    wm_InitDialog:
      BEGIN
        SetWindowPos (Dialog, 0, 200, 100, 0, 0, SWP_NOSIZE);
        Exit;
      END;
    wm_Command:
      CASE WParam OF
        idOk, id_Cancel:
          BEGIN
            EndDialog(Dialog, 1);
            Exit;
          END;
      END;
  END;
  About := 0;
END;
{---About END------------------------------------------------------}
```

Figure 3.12 Window-handling routine dialog box

The structure of this window-handling routine is almost the same as the window-handling routine presented in Figure 3.10 . The window-handling routine of a dialog box misses the statement DefWindowProc, which calls the default window procedure. The default window procedure provides default processing for any window messages that an application does not process. This function ensures that every message is processed. It should be called with the same parameters as those received by the window procedure.

When the dialog box is initiated, the messages are sent to the window-handling function. The first time this routine is called, wm_InitDialog will be executed. The command SetWindowPos moves the window to the position (200, 100). After pressing the OK button (idOK) or when the window is closed (id_Cancel), the dialog box will disappear.

The About dialog box is presented in Figure 3.13 This box is designed with the Resource Workshop and saved as a .res file.

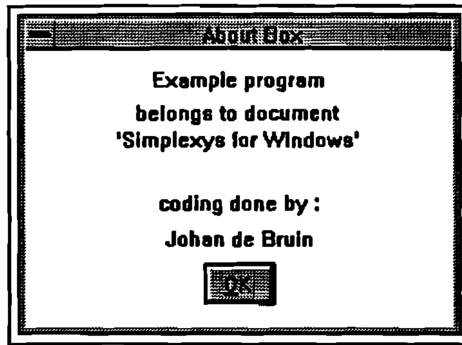


Figure 3.13 About dialog box

3.2.5 The main part

To get a working example program, also some declarations and a main part are needed. The code used to initiate the window and to start the message loop is given in Figure 3.14 After compiling all the code parts, the sample program is ready to run.

```
PROGRAM WindowsExample;
{ Sample program              }
{ Johan de Bruin, February 1995 }

USES
  WinProcs, WinTypes, Strings;

CONST
  idm_MenuOption1 = 100;          { Used for handling Menu Option 1 }
  idm_MenuOption2 = 101;          { Used for handling Menu Option 2 }
  idm_About       = 110;          { Used for handling About }

{$R Example}          { Resource file containing the menu definition }

VAR
  ExampleWindow : hWnd;                      { Handle for Window }
  TextArray     : ARRAY [0..13] OF CHAR;     { Array used for storing }
                                             { some characters        }


{---Main BEGIN-------------------------------------------------------}
BEGIN
  {Initialization of TextArray}
  StrCopy (TextArray, '');

  {Register Window Class}
  Register (@HandlingRoutine, 'Sample Window', 'SampleMenu');
  {Create Window}
  ExampleWindow := Create('Sample Window', WS_SYSMENU,
                          10, 10, 300, 150, 0);
  {Start MessageLoop}
  Loop;
END.
{---Main END---------------------------------------------------------}
```

Figure 3.14 Declarations and main part of example program

3.3 Adding a Help system

A Help system provides the users with online information about an application. Creating a Help system requires two skills: writing the Help text and programming the Help system to ensure that it works properly with the application.

To the user, the Help system is part of the application, and consists of text and graphics displayed in the Help window. To the programmer, Windows Help is a standalone Windows application. It is a program the user can run like any other application. The program can call the WinHelp function to ask Windows to run the Help application and specify which topic to display in the Help window. The topics of the Help system are stored in text files that include special codes.

During the creation of a Help system, it is useful to pay some attention to the following issues:

- The audience for the application: The audience determines what kind of information must be available in a Help System and how the information needs to be presented.
- The content of the Help topics: The topics used in the Help system should be numerous enough and specific enough to provide the users with the help they need.
- The structure of the topics: Topics can be related hierarchically. Each successive step takes the user one level down in the hierarchy of the Help system until the user reaches the topic information.

  Because users often know which feature they want help with, they can usually find what they want faster using the search feature than they can by moving through the hierarchical structure.
- The use of context-sensitive topics

  Windows Help supports context-sensitive Help. When written in conjunction with programming of the application, context-sensitive Help lets the user press F1 in an open menu to get help with the selected menu item. Alternatively, the user can press SHIFT+F1 and then click on a screen region or command to get help on that item.

3.3.1 Designing Help topics

The appearance of the Help information to the users depends on the layout of the Help topic. The Windows Help application supports text attributes and graphic images that can be used to design the Help window.
Research on screen format and Help systems has produced general guidelines for presenting information to users. Some results are summarized below:

- Use language appropriate for the audience the Help system is created for.
- Use a minimum of text. Studies have indicated that reading speed decreases by 30 percent when users read online text rather than printed text.
- Use short paragraphs. Online users become overloaded with text more easily than do readers of printed material.
- Use whitespace to help group information visually. Whitespace is important to making online text more readable. Users tend to think there is more information on a screen then exists.
- Use highlighting techniques judiciously. Using many devices will decrease the effectiveness of the visual presentation.
- Use graphics to support the explanation of visual events. Using appropriate images can help to explain some elements of the application.
- Be consistent in your design. Consistent titling, highlighting, fonts and positioning of text in the window is essential to an effective Help system.

3.3.2 Access to Help

Users may access help from the Help menu, the Help key (F1), or Help mode.
To enter Help mode, the user presses SHIFT+F1, which changes the mouse pointer to the Help pointer. To cancel Help mode, the user presses ESC.
In Help mode, the user positions the pointer over the interface element for which help is desired. When the user clicks the mouse button, the Help window appears with information appropriate to that element. Keyboard access to menu items is also available in Help mode. Choosing a menu item with the keyboard in Help mode displays Help information for the

menu item instead of initiating the item.

When using dialog boxes a Help command button can be provided in the dialog box.

To implement Help mode, the program must respond to F1 or SHIFT+F1 and the help-cursor must appear, if necessary. The program code that performs this task is shown in Figure 3.15.

```
wm_KeyDown:
  IF WParam = vk_F1 THEN
  { If Shift-F1, turn help mode on and set help cursor }
    IF GetKeyState(VK_Shift) < 0 THEN
      BEGIN
        Help := True;
        SetCursor(HelpCursor);
      END
      ELSE { If F1 without shift, call up help main index topic }
      BEGIN
        WinHelp(Window, HelpFileName, Help_Index, 0);
        Exit;
      END
      ELSE { Escape during help mode: turn help mode off }
      IF (WParam = vk_Escape) AND Help THEN
      BEGIN
        Help := False;
        SetCursor(hCursor(GetClassWord(Window, GCW_HCursor)));
        Exit;
      END;
wm_EnterIdle:
  IF ((WParam = msgf_Menu) AND ((GetKeyState(VK_F1) AND $8000) <> 0))
  THEN BEGIN
    Help := True;
    PostMessage(Window, WM_KEYDOWN, VK_RETURN, 0);
    Exit;
  END;
```

Figure 3.15 Code for entering Help mode

As long as the user is in Help mode, the cursor needs to be set to the help cursor. Otherwise Windows will reset the cursor to the default cursor of the class. To perform this task the code of Figure 3.17 is included. The helpcursor (Figure 3.16) is defined in the resource file and loaded with the command:

HelpCursor := LoadCursor (HInstance,'HelpCursor');

Figure 3.16 Help-cursor

```
wm_SetCursor:
  IF Help THEN
  BEGIN
    SetCursor(HelpCursor);
    Exit;
  END;
wm_InitMenu:
  IF Help THEN
  BEGIN
    SetCursor(HelpCursor);
    Exit;
  END
  ELSE
  HandlingRoutine := 1;
```

Figure 3.17 Code for setting help-cursor

Finally the wm_Command of the window-handling routine has to be extended. If in Help mode WinHelp must be called with the requested context.

```
wm_Command:
  { Was F1 just pressed in a menu, or are we in help mode Shift+F1 ? }
  IF Help THEN
  BEGIN
    CASE WParam OF
      idm_MenuOption1,
      idm_MenuOption2,
      idm_About: HelpContextId := wParam;
    ELSE
      HelpContextId := 0;
    END;

    IF HelpContextId = 0 THEN
      MessageBox(Window, 'Help not available for this item',
                 'Help', Mb_Ok)
    ELSE
    BEGIN
      Help := False;
      WinHelp(Window, HelpFileName, Help_Context, HelpContextId);
      Exit;
    END;
  END
  ELSE { if not in help mode }

  { old code of wm_Command }
```

Figure 3.18 Extended version of wm_Command

The WinHelp function starts Windows Help and passes optional data indicating the nature of the help requested by the application. The function uses four parameters.
The first parameter refers to the window requesting Help. The WinHelp function uses this handle to keep track of which applications have requested Help. The second parameter points to a null-terminated string containing the path and the name of the help file that the Help application is to display. The third parameter is the type of help requested. If this parameter is HELP_CONTEXT, context sensitive Help is available and the last parameter

is used. If this parameter is HELP_CONTENTS, the Help contents topic is displayed. The last parameter is ignored. If used, the last parameter refers to the requested Help topic.

3.3.3 Creating the Help topic files

Help topic files are text files that define what information is presented to the user when using the Help system. The topic files can define various kinds of information, such as an index to information about the system, a list of commands, or a description of how to perform a task. Besides the specific text for the topics, a topics file contains control codes that determine how the user can move from one topic to another.

The text files for the Help system need to be written in a Rich Text Format (RTF) editor, which has the capability to create footnotes, underlined text, and strikethrough (or double-underlined) text that indicate the control codes. The RTF capability allows to insert the coded text required to define Help terms, such as jumps, keywords and definitions. Word processors like Word 6.0 or WordPerfect 6.0 can perform this task.

Normally a topic file contains multiple Help topics. To identify the topics within the file and to control the structure of the file, some control codes are used:

- *Hard page break* separates the topics
- *Pound sign (#) footnote* defines a context string that uniquely identifies a topic. Because hypertext relies on links provided by context strings, topics without context strings can only be accessed using keywords or browse sequences
- *Dollar sign ($) footnote* defines the title of a topic. Titles are optional.
- *Letter "K" footnote* defines a keyword the user uses to search for a topic. Keywords are optional.
- *Plus sign (+) footnote* defines a sequence that determines the order in which the user can browse through topics. Browse sequences are optional.
- *Strikethrough or double-underlined text* indicates the text the user can choose to jump to another topic.
- *Underlined text* specifies that a temporary or "lookup box" box will be displayed

when the user clicks the mouse button or presses the ENTER key.

- *Hidden text* specifies the context string for the topic that will be displayed when the user chooses the text immediately preceding it.

The last step in creating a Help function is to compile the created Help text. This task is performed by the Help Compiler included in Borland Pascal.

The Help text created for the example is shown in Figure 3.19. Four pages are defined. The first page contains the index, and allows jumps to the specific topics. The other pages contain the specific topic information.

```
                                                        page 1_
#$KMain Index

To get help on an item, click the item or underlined text.
To get help on Help, press F1.

The available options are:
   -      Option 1idm_MenuOption1: Help for Option 1
   -      Option 2idm_MenuOption2: Help for Option 2
   -      Aboutidm_About: Help for About

# Main_Index
$ Main Index
K Main Index
                                                        page 2_
#$KOption 1

Help Text Option 1.

# idm_MenuOption1
$ Option 1
K Option 1
                                                        page 3_
#$KOption 2

Help Text Option 2.

# idm_MenuOption2
$ Option 2
K Option 2
                                                        page 4_
#$KAbout

Help Text About.

# idm_About
$ About
K About
```

Figure 3.19 Help text for example
(Italic is used to show the hidden text)

4 The Windows version of Simplexys

In this chapter the created program parts for the Windows version of Simplexys will be discussed. First the shell, which controls the creation, testing and debugging of an expert system, is presented. Than the expert system and the debug facilities are discussed. Finally the introduced programming elements are explained.

4.1 The Shell

The main goal of the shell is to assist the expert system programmer in creating, testing and debugging the expert system. To fulfill these requirements, a list of necessary steps is needed. To come to a good design, it is essential to know in what order the steps are handled. The steps necessary to develop an expert system are depicted in Figure 4.1.

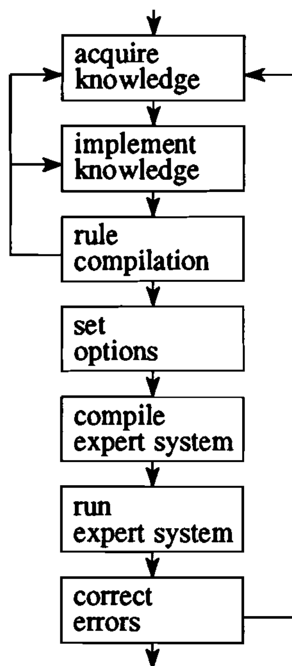


Figure 4.1 Necessary steps to develop an expert system

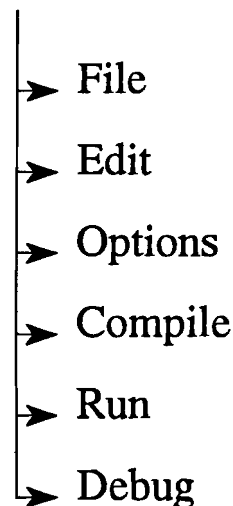


Figure 4.2 Derived menu structure

The menu structure to execute these steps is defined in Figure 4.2. The item Compile combines rule compilation and compilation of the expert system. When the shell is started, the window of Figure 4.3 appears. If the items are executed from left to right, all the necessary actions are taken to create an expert system.

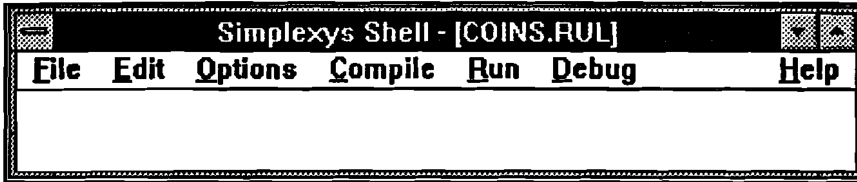
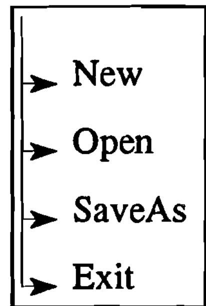


Figure 4.3 Window showing the menu structure

Item File:

Activating File shows a submenu with the options New, Open, SaveAs and Exit. New creates a new rule base. Open and SaveAs show the dialog boxes for selecting and saving a file and Exit leaves the program. The used dialog boxes are the common dialog boxes. These dialog boxes are provided by Windows and used in almost every Windows program. After choosing a rule base, the filename is presented in the window title between brackets.



To use a common dialog box the following steps have to be taken.

- Declare a structure to hold information about a file and an array to hold the filename.

  OpenFile : TOpenFileName;

  FilePathName : Array [0..128] of Char;

- Initialise the structure with arguments as available data block, owner window, default extension, default path name, structure sizes and an array that contains or receives the chosen filename.
- Call the common dialog box with the created structure. To do this, the following commands are available:

  GetOpenFileName(OpenFile) for opening a file

  GetSaveFileName(OpenFile) for saving a file

The exact program code is shown in Figure 4.4 After selecting a filename, the filename is stored in the array FilePathName.

```
FillChar (OpenFN, SizeOf(TOpenFileName), #0);  { initialize structure }
StrCopy (FilePathName, '*.*');                 { initialize filename }

WITH OpenFN DO
BEGIN
  hInstance    := HInstance;                  { datablock containing template }
  hwndOwner    := Window;                     { owner window of dialog box }
  lpstrDefExt  := 'rul';                      { default extension }
  lpstrFile    := FilePathName;               { buffer that contains filename }
  lpstrFilter  := '*.*';                      { file filter }
  flags        := ofn_FileMustExist;          { initialization flags }
  lStructSize  := SizeOf(TOpenFileName);      { length of the structure }
  nMaxFile     := SizeOf(FilePathName);       { size of the buffer pointed }
                                              { to by the lpstrFile        }
END;

GetOpenFileName(OpenFN);                      { invoke dialog box }
```

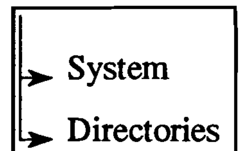
Figure 4.5 Program code to invoke a common dialog box

item **Edit:**

If a rulebase is active and the item Edit is chosen, the standard editor of Windows is started. A new window is created and the selected rulebase is opened and can be changed.

item **Options:**

Activating Options shows two new items. The first item, System, shows the dialog box of Figure 4.5. Here the options for the expert system can be set.



In this dialog box three different types of buttons are used:

OK — A *command button* contains a label that specifies the action or response represented by that button. The user can activate the button by clicking the mouse.

◇ — An *option button* represents a single option in a limited group of mutually exclusive options. Within such a group only one option can be selected. A black dot denotes the selected option.

☐ — A *check button* controls options that can either be on or off. A ✔ appears when the option is selected.

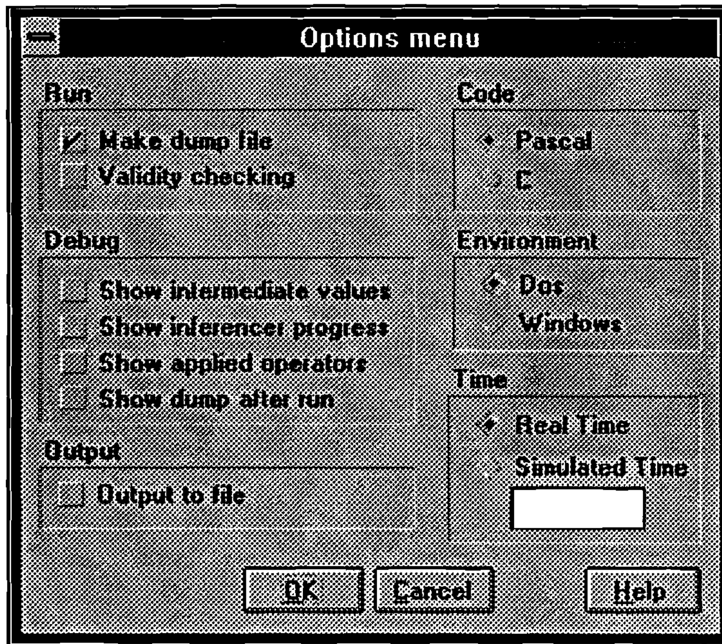


Figure 4.7 Dialog box showing system options

After activating or deactivating an item, Windows updates the screen automatically. When the dialog box is closed, the settings need to be checked. To do so, the following steps are needed:

- identify the object (item in dialog box)
- check the state of the object
- repeat for every object

The command to check the state of an item is *IsDlgButtonChecked.* The command returns the value 0 if the item is not checked, and ≠ 0 if the item is checked.

```
IsDlgButtonChecked (Dialog,        { identifies dialog box }
                    id_DumpFile)   { identifies item 'Make dump file' }
```

The second item provided by Options is Directories. Activating Directories shows a dialog box to set the directories, used by the Simplexys toolbox. Specifying the directories incorrectly can cause error messages.

To store the settings of System and Directories, an initialisation file is used. Because Windows uses initialisation files frequently, some commands are provided to use these files: *WritePrivateProfileString, GetPrivateProfileString* and *GetPrivateProfileInt.*

item **Compile:**

The item Compile starts the compilation process. First the Rule compiler is started. The Rule compiler is written for a DOS environment. Windows allows DOS programs to run in the 'multitasking' environment, if a so called pif file is created. In this file some system parameters are defined. These parameters provide Windows information about the memory usage and the priority of the program. To start the program, Windows uses the *ShellExecute* command. To start the rule compiler the following command is needed:

```
ShellExecute (Window,           { owner window }
              'open',           { operation type }
              'Ruc41.pif',      { file to open }
              Params,           { string containing paramaters of file }
              SimplexysPath,    { string containing default directory }
              SW_MINIMIZE);     { window is minimized }
```

When a program is started, a window is created. This window is present as long as the process is running. After the process has stopped, the window disappears. The availability of the window indicates that the process is still working. Testing the availability of the window is performed by the *FindWindow* command. The command returns 0, if the window is not available. To check if rule compilation has finished, the following command is used.

```
FindWindow (NIL,        { search for all window classes }
           'RUC41');    { name of window to look for }
```

After running the rule compiler, the semantics checker and the protocol checker can be executed. To run the check programs, the Pascal code of the programs has to be compiled with the qqq-files, containing specific information about the expert system. The Pascal compiler is a command line compiler and specific to the compilation, options are set. To start the compilation process, a batch file is created and started. Checking the progress of the process is performed by checking the availability of the window.

The generated check programs are executed as described for the rule compiler. If no errors are detected, the expert system and the debug system can be built. The inference engine, together with the qqq files, must be compiled. This is also done with the Pascal compiler. A batch file, containing the specific compilation options, is created and started.

If the compilation processes have run without error, the menu-items **Run** and **Debug** become active.

items **Run** and **Debug:**
With Run the created expert system is started and with Debug the debug facility is started. The structure of these parts is discussed later.

4.1.1 Error handling

During the development of an expert system several errors can occur. Some of these errors occur due to errors in the defined rule base (protocol or semantics errors) and others due to errors in the included Pascal code.

If the rule compiler detects an error, the compilation process is stopped and the editor is started. A message box appears showing the error and the offending line, and the cursor jumps to this line. After editing the rule base, the editor must be closed and the compilation process can be started again.

Other errors in the rule base can be detected during tests done by the semantics checker and the protocol checker. During execution these check mechanisms produce a textfile with information about the performed checks. The results from these tests are presented in a dialog box like Figure 4.6.

In the left upper corner some statistics, created by the rule compiler, are shown. The other information is about the tests. To show the result of the tests, a green ✔ (test passed) and a red X (test failed) are used.

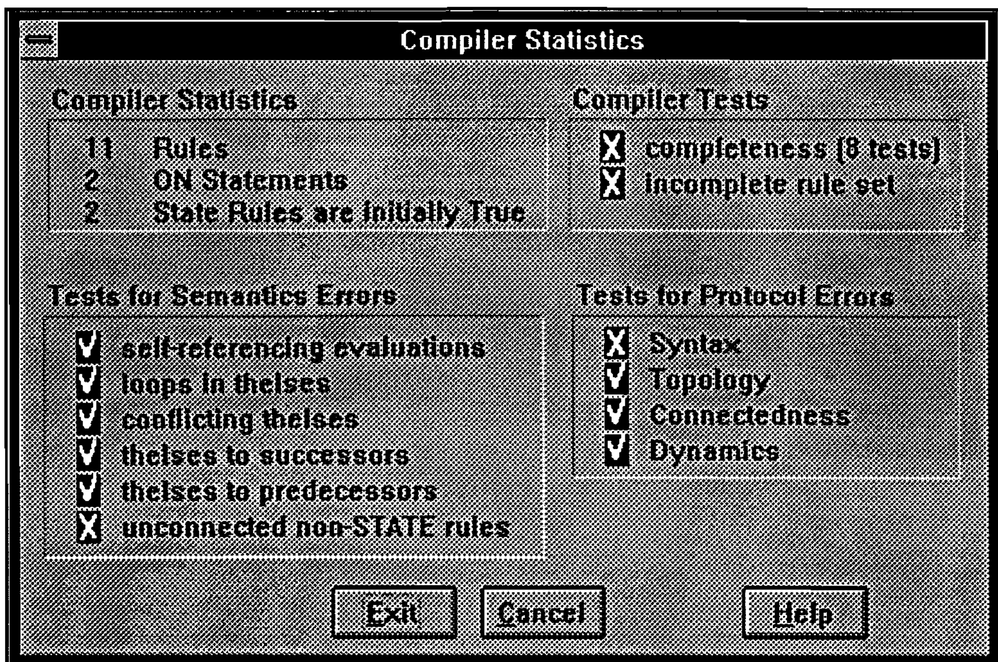


Figure 4.8 Dialog box showing results of performed checks

Drawing the markers in the dialog box is not directly possible. The dialog box contains several objects as text lines, buttons and group boxes. To draw a marker first the object, which needs a marker, must be initialized. Then a brush needs to be defined with the selected color. This brush must also be set active. Now the marker can be drawn and the object can be released. The code performing these tasks is shown in Figure 4.7.

```
hCtrlBlock := GetDlgItem (Window, idNumber);      { identify object }
hdcon := GetDC (hCtrlBlock);                      { Select object }

hndPen := CreatePen (PS_SOLID, 2, RGB (255,0,0)); { Create red pen }
SelectObject (hdcon, hndPen);                     { Select pen }
MoveTo (hdcon, 2,2);                              { Draw X marker }
LineTo (hdcon, 9,13);
MoveTo (hdcon, 2,14);
LineTo (hdcon, 9,2);
DeleteObject (hndPen);                            { Release pen }

ReleaseDC (hCtrlBlock, hdcon);                    { Release object }
```

Figure 4.9 Code for drawing markers

In the presented code, hCtrlBlock is a handle to an object, hdcon is a handle to a device context (in this case a display) and hndPen is a handle to a type for pen drawing tools.

When a test has failed and the red X marker is shown, the user can click on that error to get specific information about it. If a critical error is found, the button Exit appears, as shown in Figure 4.6. If no critical errors are detected, this button starts the building of the expert and the debug systems.

Detecting and handling errors in the Pascal code, included in the rulebase, is more difficult. The Pascal code is transferred to several qqq-files. These files are included in the inference engine and compiled by the command line Pascal compiler. The results of this compilation process are printed directly to screen. To check the results of the compilation process, the output is redirected to a file. This file is checked and if an error has occurred, the file is presented in a window. The file containing the rule base is also opened and ready for editing. Because the Pascal compiler reads the Pascal code from the qqq-files, it is not simply possible to jump to the line with the error. To solve this problem, the rule compiler could provide the qqq-files with information about the linenumbers as used in the original rul-file.

4.1.2 Introduced routines and variables

To fulfill the tasks described, a wide range of constants, variables, types, functions and procedures is needed. They will be summarized below, besides the already mentioned ones.

The introduced constants are:

TimerInterval sets the interval time used by some timers to test a process.

A set starting with *id*, *idm*, *Option* and *Dir* identifies menu items and objects in dialog boxes.

The introduced type is:

```
ErrorLst = RECORD
    Text      : STRING;          { Error message }
    ErrorType : INTEGER;         { Type of error }
    Next      : ErrorLstPtr;     { Pointer to next line }
    Prev      : ErrorLstPtr;     { Pointer to previous line }
END;
```

The meaning of the variables is:

ActivityWindow, *ErrorWindow* and *ShellWindow* are handles to windows.

ActionMessage contains the message for displaying in the activity window.

WaitCursor and *HelpCursor* are handles to the Wait- and the Helpcursor icons.

Help is TRUE if help-mode is activated, otherwise it is FALSE.

ScrollBar defines the position of the scrollbar.

Lines defines the number of visible lines.

State defines the compilation state. Possible states are:

NewFile: The selected rulebase is new or has been changed. The rule compiler must be executed.

AllCompiled: Nothing has changed. No compilation is needed.

BuildOnly: The rulebase is unchanged, but the system settings are changed. The expert system and the debug system must be recompiled.

A set of strings to store various file and path names.

A set of booleans to store the settings of the debug options.

A set of pointers to handle the errorlist.

The function of the routines are:

ErrorWindowProc, *CompilerStatsProc*, *ShellWindowProc*, *ActivityWinProc*, *Compile-WinProc*, *OptionsWinProc* and *DirectoryWinProc* are window-handling routines.

GetDefaults and *WriteDefaults* are used to read and write the settings used by the Simplexys Toolbox.

GetOptions and *SaveOptions* are used to read and write the debug options.

GetFileName and *SaveFileName* are used to handle the Open File and the Save File dialog boxes.

TestFile is used to check if a file is available.

SearchError checks if an error has occurred during rule compilation or during the protocol or semantics check.

CompileError checks for errors generated by the Pascal compiler.

ReadCompilerInfo reads the information generated by the rule compiler.

InvalidateWindow updates the screen partially.

4.2 The Expert System

4.2.1 The DOS-version

The existing DOS-version of the expert system is built around the program part sim41.pas. This file contains the inference engine. After compiling the inference engine and the qqq-files containing specific information about the expert system, the complete system is built and ready to run.

4.2.2 The Windows-version

The structure of the Windows version can be found in Figure 4.8. The figure shows three blocks.

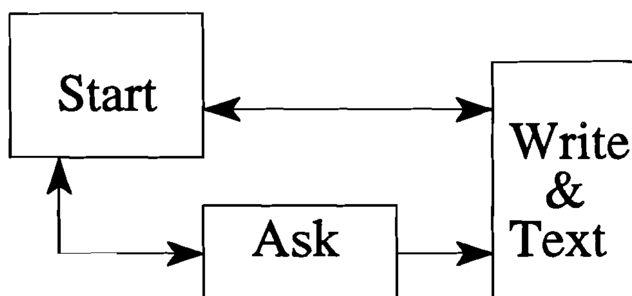


Figure 4.10 Windows structure of Expert System

If the block 'Start', together with the qqq-files, is compiled the system is ready to run. The block 'Start' contains the window-handling routines and the inference engine. The block 'Ask' contains the input routines and the block 'Write & Text' contains functions and procedures to write output to a file and to store text in a buffer. The arrows indicate that there is communication between the blocks.

The block 'Start'

The block 'Start' consist of two files. The first file, startup.pas, contains the window-handling routine. During compilation, the inference engine (sim41.pas) is included. When

the program is started, a window like Figure 4.9 is created.

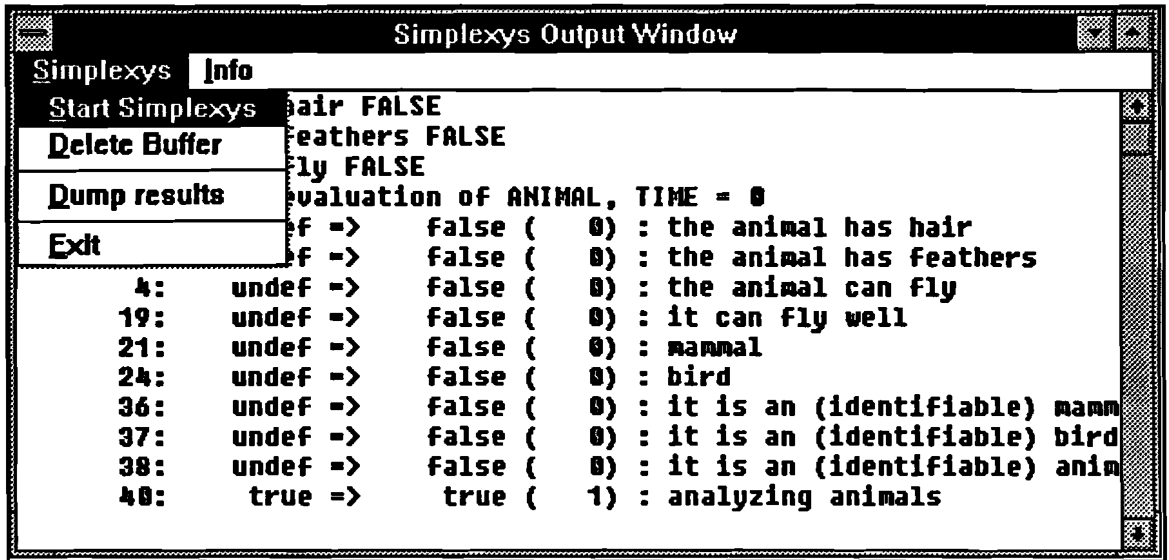
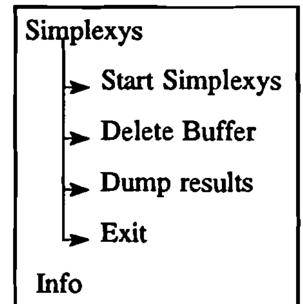


Figure 4.11 Sample Simplexys output window

The available options are:

Start Simplexys:	Restart the application.
Delete Buffer:	Clear display buffer.
Dump results:	Give an overview of reached goals
Exit:	Closes application.
Info:	Shows information about the expert system.



All the output generated during the inferencing process is directed to the window. To be able to scroll through the generated output, the window provides a scrollbar. The routines for handling this scrollbar are a part of the window-handling routine belonging to the window.

The file sim41.pas has changed a little. The routines to handle input are not included any more. The routines are called from the block 'Ask'. Also the main part of the DOS-version is transferred to a procedure in the Windows version. This procedure is started from the window-handling routine in startup.pas.

The new types, variables, and routines introduced in startup.pas are:

Variables:

SimplexysWindow is the handle to the output window.

Lines holds the number of lines visible in the output window.

NumberOfEntries holds the number of lines in the text buffer.

Scrollbar holds the index of the scrollbar.

Running is TRUE, if the Simplexys system is active and FALSE if the Simplexys system is inactive.

FirstEntry, LastEntry and PointedEntry are pointers to the first, the last and the actual line in the text buffer.

TextString and *Convert* are variables used for creating strings to use with WritText.

Types:

```
List = RECORD
  Text : STRING[80];          { Storage for text }
  Next : ListPtr;             { Pointer to next line }
  Prev : ListPtr;             { Pointer to Previous line }
END;
```

The introduced functions are:

Simplexys: This routine starts the inferencing process.

The routines defined in the blocks 'Ask' and 'Write & Text'.

The Block 'Ask'

The routines used for data input during the inferencing process are stored in the block 'Ask'. This block is a DLL. The defined routines are called by the block 'Start'.

Creating a separate DLL for the ask-routines makes it possible for the programmer to change the appearance of the routines without changing the main expert system. Several sets of input routines can be used, without compiling the expert system again. Only the DLL containing the ask-routines needs to be replaced.

When the block 'Start' initiates the ask-routines, a pointer to the function in the DLL is provided. The created ask-routines use dialog boxes like Figure 4.10. For input some pushbuttons are provided, but it is also possible to use the keyboard. The dialog boxes, created for the ask-routines, are stored in the file ask.res.

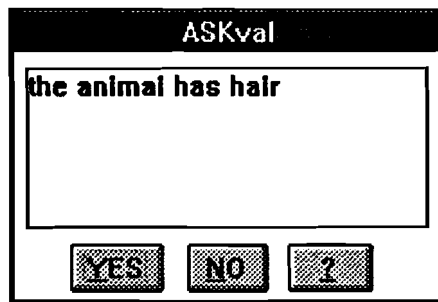


Figure 4.13 Ask input box

Introducing a DLL for the ask-routines gives also an advantage when using multiple programming languages. The functions and procedures need to be defined once in the DLL and can be used in all the programming languages. So when the C-version of Simplexys is adapted to Windows, the ask-routines designed for the Pascal version can be used without any changes.

The functions and procedures, defined in the DLL 'Ask' are summarized below:

- *InitAsk*: initializes the DLL with the debug options used in Sim41 and passes the handle of the output window to the DLL
- *ASKyn*: ask-routine with answers yes or no
- *ASKval*: ask-routine with answers true, possible and false
- *ASKint*: ask-routine requesting an integer value
- *ASKword*: ask-routine requesting a word value
- *ASKreal*: ask-routine requesting a real value

The block 'Write & Text'

The output to screen and file is performed by the block 'Write & Text'. This block is a

DLL and can be accessed by the blocks 'Start' and 'Ask'. The routines cannot be stored in the part 'Start', because 'Ask' must have access to them also (for a DLL it is not possible to use functions or procedures of a calling part). To store text in the used textbuffer a linked list is created in the DLL Text. The command *WriteText (Text: String)* is used for storing information in the textbuffer. The pointers to the first and last line in the buffer are passed to the block 'Start' when a display command is sent.

The WriteText command does not perform a repaint of the screen, because some actions write more lines at once. If a repaint of the screen would be provided after every command, the screen would start to flicker. Requesting a repaint of the screen is done by the procedure *InvalidateRect (Window, Rect, Erase)*. The parameter Window identifies the window that needs to be repainted. Rect points to a structure that contains information about the part that needs to be updated. If Rect is NIL, the whole window is updated. Finally Erase specifies whether the background is to be erased. If Erase is TRUE, the background is erased, otherwise the background remains unchanged.

The DLL 'Write & Text' exports the following functions and procedures:

InitText:	Passes the handle of the output window to the DLL.
WriteText:	Puts a string in the text buffer.
GetEntries:	Gets a pointer to the first and the last entry in the textbuffer and gives the number of lines in the buffer.
DeleteEntries:	Deletes all the entries in the buffer.
WriteFile:	Writes output to file.
OpenWriteFile:	Opens the output file.
CloseWriteFile:	Closes the output file.

4.2.3 Memory usage

When the DOS- and Windows-version are compiled, the sizes for data- and codesegment of table 1 are created.

Table 1 Data comparison DOS and Windows version

Rulebase	animals.rul		coins.rul	
	DOS	Windows	DOS	Windows
Code segment (in bytes)	15728	15419	16240	17936
Data Segment (in bytes)	5168	4964	2572	2368

The knowledge bases animals.rul and coins.rul are small examples with 40 and 7 rules. The sizes for DOS are obtained by compilation of the file sim41dos.pas, which creates an executable DOS-file, and the sizes for Windows come from the file startup.pas, which is a Windows application.

4.3 The Simulate and Explain facility

4.3.1 The DOS-version

In the DOS-version of Simplexys the simulate and explain facility are separate programs. The structures of these programs is shown in Figure 4.11.

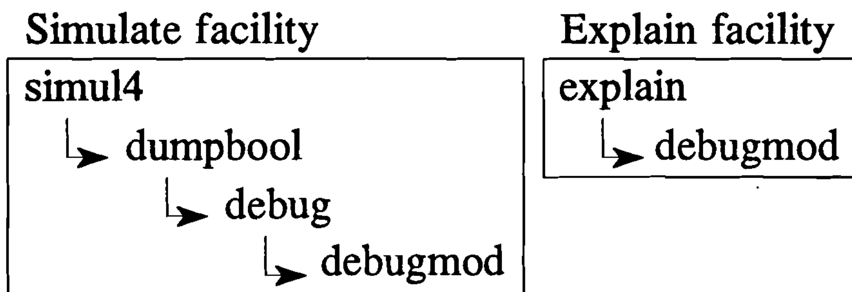


Figure 4.14 DOS-structures of Simulate and Explain facility

The simulate facility consists of the files simul4, dumpbool, debug and debugmod. The explain facility consists of the files explain and debugmod. The files simul4 and explain are the main parts. They include the other files.

The file simul4 provides the inference engine. The file dumpbool contains the procedures and functions to simulate the process with the data stored in the dumpfile. The file debug contains the routines for tracing a simulation block and the file debugmod contains the routines for changing the display options and explaining the evaluation structure of the rules. This file also includes the routines for handling keyboard input. It controls the different processes and appropriate actions are taken until the program is finished. Finally, the file explain makes it possible to examine Simplexys rulebases.

To simulate a specific expert-system, some qqq-files (rinfo.qqq, ruses.qqq, options.qqq, rdodo.qqq, rtest.qqq and hist.qqq) and the file simplex.sav with information about saved runs are needed. To use explain, the file rinfo.qqq is needed. After compiling the files simul4 and explain, the simulate facility and the explain facility are created and ready to run. For more details see [Philippens, 1990].

4.3.2 The Windows-version

In Windows the simulate and explain facilities are combined into a single program. The structure of the Windows program is shown in Figure 4.12.

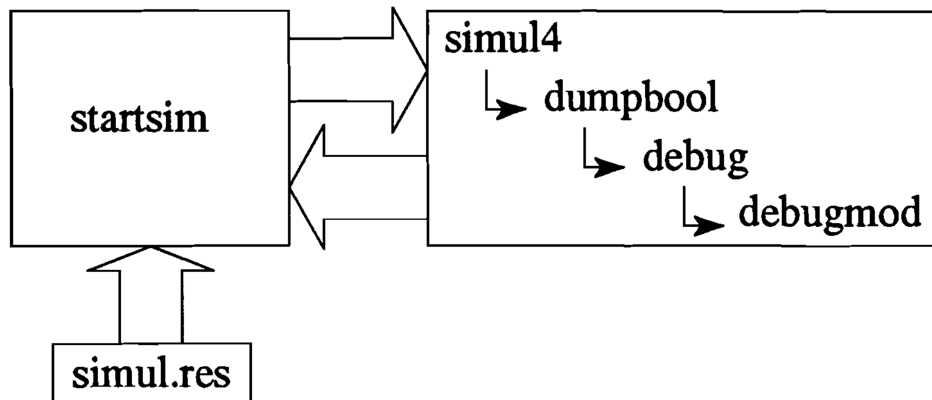


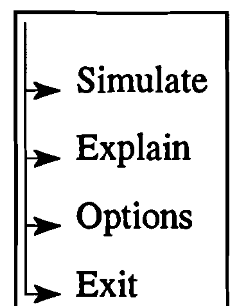
Figure 4.15 Windows-structure of simulate and explain facility

The program is started by the file startsim. This part initializes the used window classes and makes some of them visible. Also the message loop is started. Information about the used menu structure and the used dialog boxes is stored in the file simul.res. The part startsim handles all user input and output to screen. It also controls the simulate and explain processes.

The files simul4, dumpbool, debug and debugmod are placed in a DLL. Using a DLL gives the advantages of an own data segment. In the DLL, the data segment is used to store data for the simulate and explain processes. The variables needed to provide input and output are declared in startsim.

When the program is started, a window like Figure 4.13 appears.
Four menu items are shown:

Simulate: After selecting the rules to examine, the simulation process is started
Explain: Two options are available: displaying rule information or displaying tree information.
Options: Settings of simulation time and some screen options can be changed.
Exit: Quits the program.



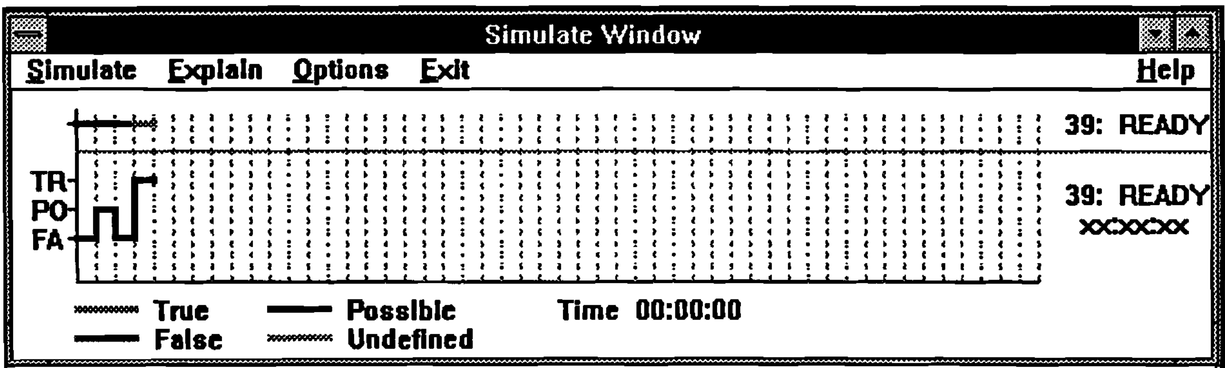


Figure 4.17 Options of showing tracer information

item **Simulate:**

Activating simulate starts the simulate facility. This options is only active if the file simplex.sav is available. First a dialog box like Figure 4.14 is presented. This dialog box shows specific rule information. By changing the rulenumber or using the buttons << and >>, a specific rule can be displayed. The rule is selected by activating the OK button. After selecting some rules, tracer information about these rules is shown.

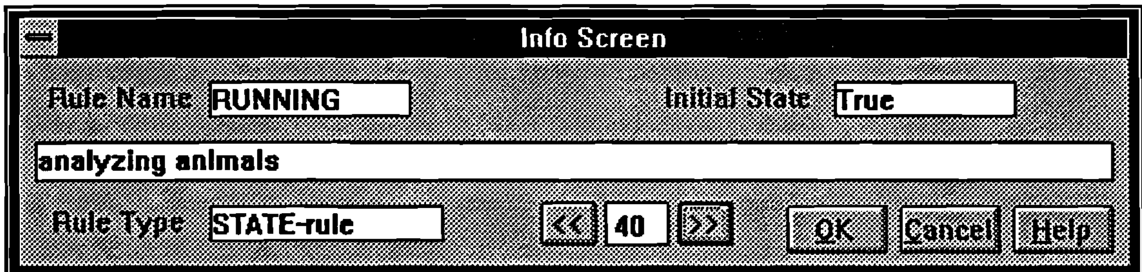


Figure 4.18 dialog box showing rule information

It is possible to show the rule value of a single run in two ways. First, the value of a run is represented by a color. Every possible value has a own color. A block of several runs appears as a single line with changing colors. The other option shows the value of a rule by changing the position of the line. The height is an indication for the value. These display options are shown in Figure 4.13.

item **Explain:**

Activating the item explain makes it possible to show rule information or to show tree information. To show rule information, the dialog box of Figure 4.14 is used. Activating the OK button returns to the main menu. Selecting a rule is done as described for the

option simulate.

To show tree information, a separate window is used. This window includes a scrollbar to view large trees. To store tree information, a linked list with pointers is used.

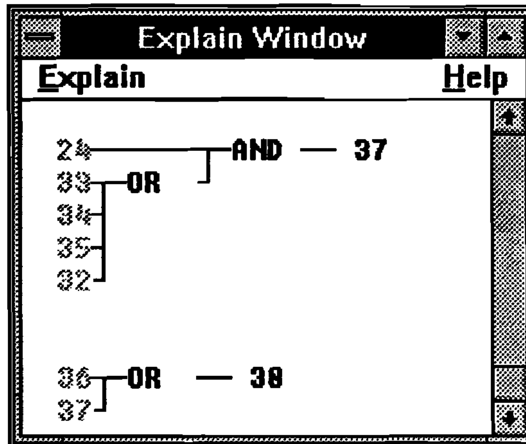


Figure 4.19 Window for showing trees
The affected rules are represented by their rule-number. To evaluate rule 37, rules 24 and 33 or 34 or 35 or 32 need to be evaluated.

Every line of the tree is stored in a record. Using this structure makes it possible to store tree information about several rules, without decreasing the size of the datasegment. The routines for using the scrollbar are included in the window handling routine.

item **Options:**

This option allows the user to define some settings. 'Block Options' allows to select the part that needs to be displayed in tracer mode. 'Screen options' allows to select the way of displaying tracer information (by color or by line) and how many rules are examined at the same time. Finally, 'Explain Window' enables the window to show tree-information. If tracer information is shown, clicking on a rule shows the tree-information about that rule.

4.3.3 Changes in the DLL

In the DOS version simul4 was an executable program with a main part. In the Windows version, this part is moved to a DLL. The main part of the DOS program is transferred to a function, called Simplexys.

The control over the several processes is placed in the part startsim. Also the output to screen is provided by that part. The program parts that provide screen output and perform process control are not needed any more in the DLL. To specify the working mode a variable Mode is introduced. Setting this variable makes the specific routines for the requested action active. The values used to set Mode are shown in table 2.

Table 2 Values of variable Mode

Mode	Function
Mode_Tracer	Get tracer information of selected rule
Mode_Tree	Show tree of selected rule
Mode_End	End application

Information created in tracer mode was stored in an array and short rule information was directly printed to screen. Because the DLL does not perform screen output any more, it is not necessary to store information about more than one rule. The array is deleted and all the information created to show tracer info is stored in a record.

If a rule needs to be examined, the procedure Simplexys is called with the variable Mode set for the requested action. When the asked information is created, the record is transferred to startsim. The created data is not needed any more in the DLL. If more rules need to be shown, the procedure Simplexys must be called for every rule. The data, used to display more rules is stored in startsim.

To be able to create all the information and to pass this information to the module startsim, some new constants, variables, types, functions and procedures are introduced.

The usage of the constants is as follows:

Mode_Tracer, *Mode_Tree* and *Mode_End* are used for setting the operation-mode.

SizeBlock determines the maximum number of runs in a block.

The meaning of the variables is:

BlockCount determines the simulationblock that is in process.

SimulationBlock points to the simulation block the tracer asked for.

ColorOfText stores the actual color of the text. This makes it possible to show the value of a rule by color.

ShowRule holds the rulenumber whose information is asked for by startsim.

PointedRun holds the number of the run in the actual block.

Mode holds the operation mode of Simplexys.

Convert, *ConvertString* and *ConvertPChar* are used for building a string to show with a MessageBox.

GlobalWindow is the handle to the main window in startsim.

TracerInfo is the record in which the tracerinformation is stored.

The introduced types are:

```
TracerRec = RECORD              { Storage of Tracer Info }
  id     : INTEGER;             { Rule Number }
  Values : ARRAY [0..SizeBlock] OF ShortInt;
                                { TracerValues of actual block }
  Hist   : LONGINT;             { History information }
  Info   : STRING[25];          { Short RuleInformation }
END;

TreeBuf = RECORD                { storage for line of tree }
  Text  : STRING[80];           { string for line }
  Color : ARRAY [1..80] OF INTEGER;
                                { Character color }
  Next  : TreeBufPtr;           { Pointer to next line }
  Prev  : TreeBufPtr;           { Pointer to previous line }
END;
```

The new introduced procedures and functions are summarized below. Most of the routines are defined to allow startsim having access to variables declared in the DLL.

Simplexys:	Starting the process to build tracer or tree information.
GetOptions:	Get the options used by the procedure Simplexys.
SetOptions:	Set the options for use with Simplexys.
SetShowRule:	Sets the rule whose information is asked.
GetNumberOfRules:	Gets the number of rules used in the expert system.
GetRuleInfo:	Gets the rule information of the rule pointed to by showrule.
GetTracerInfo:	Gets the tracer information of the rule pointed by showrule.

GetProces: Gets number of simulated runs in actual block.

SetWindow: Passes handle of main window to DLL. So MessageBoxes can be shown.

SetPointedRun: Sets the number of the run in the actual block whose information is wanted.

GetTimeOfRun: Gets the time of the pointed run in the actual block.

SetSimulationBlock: Sets the simulation block whose information is wanted.

GetMaxBlocks: Gets the maximum number of blocks in the simulation process.

SetMode: Sets the operation mode for Simplexys.

WriteTreeBuf: Adds a string to the open line of the TreeBuffer.

WritelnTreeBuf: Adds a string to the open line of the TreeBuffer and opens a new line.

4.3.4 Memory usage

When the DOS- and Windows-version are compiled the sizes for data- and codesegment of table 3 are created.

Table 3 Data comparison DOS and Windows version

Rulebase	animals.rul		coins.rul	
	DOS	Windows	DOS	Windows
Code segment (in bytes)	42460	29026	42544	28929
Data Segment (in bytes)	25364	18098	19070	13422

The sizes for DOS are obtained by compilation of the DOS file simul4.pas, which creates an executable file, and the sizes for Windows come from the changed file simul4.pas, which creates a DLL.

4.4 New programming elements

When programming a rulebase, a wide range of commands can be used. These commands are provided by the Simplexys language or by the used programming language. The creation of a rulebase for a DOS or a Windows environment differs only in the way input and output to screen is managed.

To get input, the Simplexys language provides some ask-routines. These routines are defined for both the DOS version and the Windows version. To usage of these routines is independent of the environment. During compilation it is defined which set of routines must be used.

The output to screen is more difficult. Within the DOS version, *Write* and *Writeln* are used for output to screen. The parameters for these commands can be strings and/or numbers. Type conversion is provided automatically.

In Windows, the above mentioned commands have no function. To write something to the output window, the command *WriteText (TextString: String)* is defined. Because the used parameter is a string, the programmer must take care of the necessary type conversion. It is also possible to define a new window. The screen handling is performed by the window handling routine of the new window.

The created user interface is suited to develop both Windows and DOS applications. To combine both versions in a single rulebase, the {$IFDEF} compiler statement of the Pascal compiler can be used. When compiling a DOS system, the DOS directive is set. For a Windows system, the WINDOWS directive is set. The code specific for DOS is placed between {$IFDEF DOS} and {$ENDIF} and the code specific for WINDOWS is placed between {$IFDEF WINDOWS} and {$ENDIF}. Using this technique, universal rulebases can be defined.

After successfully developing an expert system, the system needs to be exported. The files needed to run the created expert system are summarized in table 4.

Table 4 Files belonging to expert system

	Files	Function
DOS	sim41dos.exe	executable expert system
Windows	startup.exe	executable, containing expert system
	write.dll	dll containing write routines
	text.dll	dll containing text routines
	ask.dll	dll containing ask routines

Using the debug option 'dump results', the file simplex.sav is created. This file is needed to run the simulate facility. The structure of the file is independent of the used environment. This makes it possible to examine the results of both versions with the Windows simulate facility. The files needed to run the simulate and explain facility are shown in table 5.

Table 5 Files belonging to debug system

Files	Function
startsim.exe	executable, controlling the debug processes
simul4.dll	dll, containing the debug system
bwcc.dll	dll, containing custom classes

5 Conclusions and recommendations

With the Windows version of Simplexys it is possible to develop expert systems specially suited for the Windows environment. To create these systems, a user interface is implemented to run the tools of the Simplexys toolbox. The user interface allows an easy access to the rulebases and editing these rulebases. Further the rulebases can be compiled, checked, executed and debugged. To reach flexibility, it is also possible to develop expert system running in a DOS environment. To provide the user with information about the menu commands, a context sensitive help-function is implemented.

The inference engine has changed a little. Within Windows, input and output is handled differently. To be independent of the environment (DOS or Windows), the routines for handling input are defined for both. During compilation, the required set of routines is used.

The input routines for the Windows version are placed in a DLL. This makes the routines accessible for other programming languages too. When the C-version of Simplexys is adapted to Windows, these routines can be used, without any changes.

The debug utilities are combined into one program. The created information is stored and presented graphically. In order to have no performance decrease (smaller data segment), the debug facilities are placed in a DLL. The routines for displaying and handling the several debug processes are placed apart. To exchange data a procedural interface is used.

In order to improve the Windows version of Simplexys some recommendations can be made:

- The created Windows expert systems are specially set up for Windows. During development of an expert system, some processes use DOS programs (Rule compiler, additional checkers). These programs run in a so called DOS-box. To make the toolbox more efficient, these processes can be adapted to Windows.

- To give the user more support while developing an expert system, the help function can be extended with references about the Simplexys language.
- To simplify finding errors in the rul-file, the rule-compiler could provide the qqq-files with information about the original linenumbers in the rul-file. After detecting an error in a qqq-file, the information about the linenumbers provides sufficient information to find the line with the error in the original rul-file.
- Because Borland Pascal allows only a data segment of 64 kb, the additional check programs (semantics and protocol checker) cannot test large rule bases. Because C is more flexible in memory management, it could be possible to perform these checks with a C program.

Appendix A

References

Blom, J.A.
The Simplexys Experiment: Real time expert systems in patient monitoring
Thesis: Eindhoven University of Technology, 1990

Blom, J.A. and G.A. van Poppel
SIMPLEXYS Reference
Faculty of Electrical Engineering, Eindhoven University of Technology, 1993
Internal manual

Blom, J.A. and G.A. van Poppel
Using SIMPLEXYS
Faculty of Electrical Engineering, Eindhoven University of Technology, 1993
Internal manual

Borland International
Borland Pascal manuals
Borland international, Inc., 1992

Edson, D.
Writing windows applications from start to finish
New York: M&T Books, 1993

Hair, P.J.A. de
Realisatie van een uitlegfaciliteit voor Simplexys Expertsystemen
Thesis: Eindhoven University of Technology, 1988

Lammers, J.O.
The Use of Petri Net Theory for Simplexys Expert Systems Protocol Checking
Faculty of Electrical Engineering, Eindhoven University of Technology, 1990
EUT Report 90-E-238

Lutgens, J.M.A.
Knowledge Base Correctness Checking for Simplexys Expert Systems
Faculty of Electrical Engineering, Eindhoven University of Technology, 1990
EUT Report 90-E-240

Microsoft Corporation
The Windows interface: an application design guide
Redmond: Washington, Microsoft, 1992

Petzold, C.
Programming Windows, The Microsoft guide to writing applications for Windows 3.1
Redmond, Washington: Microsoft, 1992

Philippens, E.H.J.
Designing Debugging Tools for Simplexys Expert Systems
Faculty of Electrical Engineering, Eindhoven University of Technology, 1990
EUT Report 90-E-234

Poppel, G.A. van
A user interface for SIMPLEXYS Experts Systems
Faculty of Electrical Engineering, Eindhoven University of Technology, 1993

Swan, T.
Programmeren in Turbo Pascal voor Windows
Schoonhoven: Academic Service, 1991

Appendix B

List of files

ASK.DLL	dll containing ask routines
ASK.PAS	source code of ask routines
ASK.RES	resource information for ask routines
BATCH.BAT	batch file for compilation process
BATCH.PIF	file containing system options
BWCC.DLL	dll containing custom classes
CHK41.PAS	source code semantics checker
CHK41.PIF	file containing system options to run semantics checker
DEBUG.PAS	source code debug facility
DEBUGMOD.PAS	source code debug facility
DUMPBOOL.PAS	source code debug facility
HELP.HLP	compiled version of help text
HELP.HPJ	project file for help text
HELP.RTF	help text (Rich Text Format)
HELPIDS.H.	list of identifiers to get context sensitive help
PET41.PAS	source code protocol checker
PET41.PIF	file containing system options to run protocol checker
RUC41.EXE	rule compiler
RUC41.PAS	source code rule compiler
RUC41.PIF	file containing system options to run rule compiler
SIM41.PAS	source code inference engine used in Windows version
SIM41DOS.PAS	source code inference engine used in DOS version
SIMPLEX.RES	resource information expert system
SIMSHELL.EXE	executable shell
SIMSHELL.PAS	source code shell
SIMSHELL.RES	resources for shell
SIMUL.RES	resources for debug facility
SIMUL4.PAS	source code dll for debug facility
STARTSIM.EXE	executable startup part debug facility
STARTSIM.PAS	source code of executable startup part debug facility
STARTUP.PAS	source code startup part of expert system (Windows version)
TEXT.DLL	dll containing text routines
TEXT.PAS	source code text routines
WRITE.DLL	dll containing write routines
WRITE.PAS	source code write routines