A GRAPHICAL USER INTERFACE FOR CAD PROGRAMS

XI LU

A GRAPHICAL USER INTERFACE FOR CAD PROGRAMS

by

© Xi LU

A thesis submitted to the School of Graduate Studies in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Memorial University of Newfoundland

October 1995

St. John's Newfoundland

ISBN 0-612-17616-9

Canada

Abstract

The aim of this project is to outline the design and implementation of graphical user interfaces for engineering and scientific computer-aided design tools. Graphical interfaces are quite important for sophisticated software tools because they simplify the use of the tools and reduce the learning phase. A properly designed user interface "guides" a user through the interactions with design tools, clearly indicating the design decisions and supplying all relevant information which is helpful in making these decisions.

A package for graphical presentation and manipulation of numerical results is designed and implemented for existing computer aided design programs SPICE-PAC and FIT. The implementation is based on the Athena widget set under the X Window system.

The methodology developed during this project is applicable to many other tools, reducing their development time and simplifying the use of developed programs.

Acknowledgments

I wish to express my thanks to my supervisor, Dr. W.M. Zuberek, for his guidance, constructive suggestions and enthusiasm.

I am grateful to the Department of Computer Science for providing the environment for my M.Sc. program, and I would like to thank the system support staff for all their help and assistance that I needed during my work on this project.

Contents

List of Figures

Chapter 1

Introduction

1.1 Graphical User Interfaces

Due to the growing popularity and availability of computers, especially personal computers, user interfaces are becoming increasingly important in many applications. Since the user interface of a system is often one of the main features which determine the usefulness of a software system, graphical interfaces are becoming a "standard" for workstations and personal computer systems.

User interfaces which rely on windows, iconic (pictorial) representations of entities, pull-down or pop-up menus and pointing devices are now called *graphical user interfaces* (GUIs). They are characterized by:

1. Multiple windows allowing different information to be displayed simultaneously on the user's screen.

2. Iconic information representation.

3. Command selection via menus rather than a command language.

4. A pointing device such as a mouse for making selections from a menu or indicating an item of interest in a window.

5. Support for graphical as well as textual information display.

Graphical user interfaces simplify using the system; they allow the user to easily switch from one task to another as well as to interact with the application programs.

1.2 X Window System

The X Window System is an industry standard software system for graphical applications. One of the most important features of X is its unique device-independent architecture. X allows programs to display information containing text and graphics on any hardware device that supports the X protocol without modifying, recompiling, or relinking the application. This device independence, along with X's position as the industry standard, allows X-based applications to function in heterogeneous environments consisting of mainframes, workstations, and personal computers.

X provides a powerful platform that allows programmers to develop sophisticated user interfaces, portable to any system that supports X. X is based on a network transparent client-server model. The X server creates and manipulates windows in response to requests from clients, and sends events to notify clients of user input or change in a window's state. Clients can execute anywhere on a network, making X an ideal base for distributed applications.

X does not support any particular interface style, and strives to be policy-free. Applications are free to use the X primitives to define their own type of user interfaces.

The easiest way for a user to follow the basic guidelines is to use a higher level toolkit, such as Xt Intrinsic, Motif or Athena.

1.3 The Requirement

Several computer-aided design software tools have been developed without flexible user interfaces. For example SPICE-PAC is a collection of loosely coupled simulation "primitives" that can be composed in many different ways, as required by a particular application. However, it does not include an output module; all numerical results are returned as arrays of data, hence a graphical back-end with some post-processing capabilities, such as comparing results of different runs, storing and extracting results to/from files, and so on, is needed.

FIT is an interactive program for extraction of device parameters for SPICE-like circuit simulators. It is based on a circuit simulator (SPICE PAC) rather than an explicit set of model equations. The FIT program has some output facilities but they are inflexible and rather insufficient. Again, a graphical back-end is needed for FIT.

1.4 The Objective

The main objective of this project is to design and implement a package for graphical presentation and manipulation of (numerical) results for SPICE-PAC and FIT programs. A high level of parameterization is to be attempted to provide flexibility of the package. A simple interactive user interface is to be provided for refinements and manipulations of the graphics.

1.4.1 Designing a Graphical Back–End Package

The X-window system is to be used for implementation of a graphical back-end package. Since X clients are event-driven, the CAD programs need a different organization, more suitable for event-driven control.

Designing a graphical back-end package encompasses three aspects: application flow, layout policy, and construction tools. Understanding the application flow and selection of components such as menus, command buttons, etc., is the first step of a design. The next step is to determine the interface implementation policy which, in this case, is to use the Athena widget set.

1.4.2 Standardizing the Interface

Standardizing the user interface has many advantages. It provides consistent program structure, minimizes user learning time, and streamlines program development.

Fig.1.1 shows an outline of the main application window. It has the **Quit** button; the **File** menu which contains submenus providing selections for *loading*, *editing*, *printing* and *saving* files; the **Analysis** menu with submenus for *SPICE analyses*, *FIT analyses*, and so on; the **Display** menu containing a dialog display with functions for manipulating graphic displays. The display area is used for graphical presentation of results of different analyses.

1.4.3 High Level Parameterization

High level of parameterization means that the users can set resources to maintain system wide consistency between applications. Users can change the layout of win-

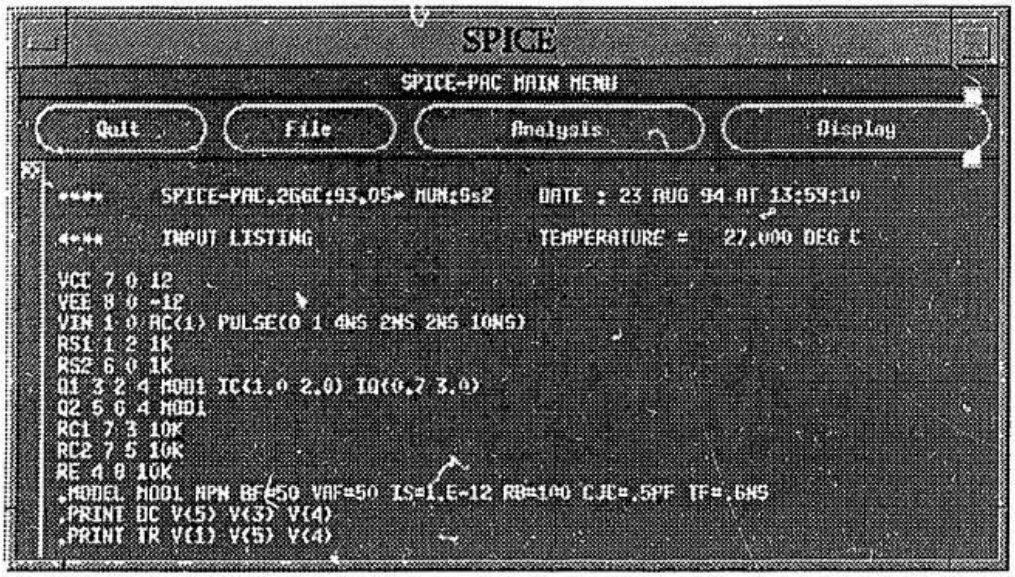

Fig.1.1. A standard layout of the GUI.

dows, modify window size and style, change the foreground and background colors or select a font which seems to be the most readable. Obviously, the users can only customize the Athena applications for those resources which are not hard-coded within the programs.

There are five basic ways to set resource values: four of them are external to the programs:

1. In a user resource file, which is a text file containing resource-setting commands.
2. In a class resource file, another text file containing resource-setting commands, but these commands apply to one application class only.
3. In the *RESOURCE_MANAGER property* of the root window.
4. In command-line parameters passed to the program.
5. Hard-coded in a program.

In this Athena application, resource values can be set in the user resource file ".Xdefaults" as well as in application default files, such as "Spice" or "Fit". For example, the "Spice" file can contain all resource-setting commands that apply to SPICE applications:

```
spice*labelString:        SPICE-PAC MAIN MENU
spice*width:              200
spice*height:             225
```

This file can be placed in any of the resource file locations such as *HOME/Class*. Finally, mechanisms for individual widgets are also developed.

1.4.4 The Widget Set

When developing an application, it is convenient to have libraries of routines that provide typical functions for the application. In this case, the Athena Widget Set is used, developed concurrently with the X Window system. The Athena Widget Set is intended to provide set of typical interface components.

For the graphical back-end package, the following widget classes are selected from the Athena set:

- *OverridShellWidgetClass* for pop-ups.
- *FormWidgetClass* for layout management.
- *BoxWidgetClass* for layout management.
- *ListWidgetClass* for the pop-up option selection list.
- *CommandWidgetClass* for buttons and menu panes.

- *LabelWidgetClass* for the menu titles, message area, and field labels.
- *FieldEdWidgetClass* for the field entry.
- *AsciiDiskWidgetClass* for help and so on.

1.4.5 Building an Application

To build an application, the program needs to be decomposed into several parts. The event-driven organization needs to be implemented; the application resource-gathering mechanism needs to be designed; the application structure needs to be built; the pop up warning system needs to be created; the pop-up file menu system needs to be constructed and the display window needs to be designed.

Fig.1.2 is a simple illustration of some capabilities of the Graphical Back End Package.

1.5 Overview of Thesis

This thesis is organized in 6 chapters and a series of appendices.

Chapter 2 describes the features of the X window system, basic concepts of object-oriented programming and the widget set. Chapter 3 describes the idea of the graphical user interface design in greater detail. Chapter 4 discusses the X window resources and the process of building the interface layout for an application. Chapter 5 discusses the implementation of the graphical user interface. Chapter 6 presents a number of conclusions related to this project.

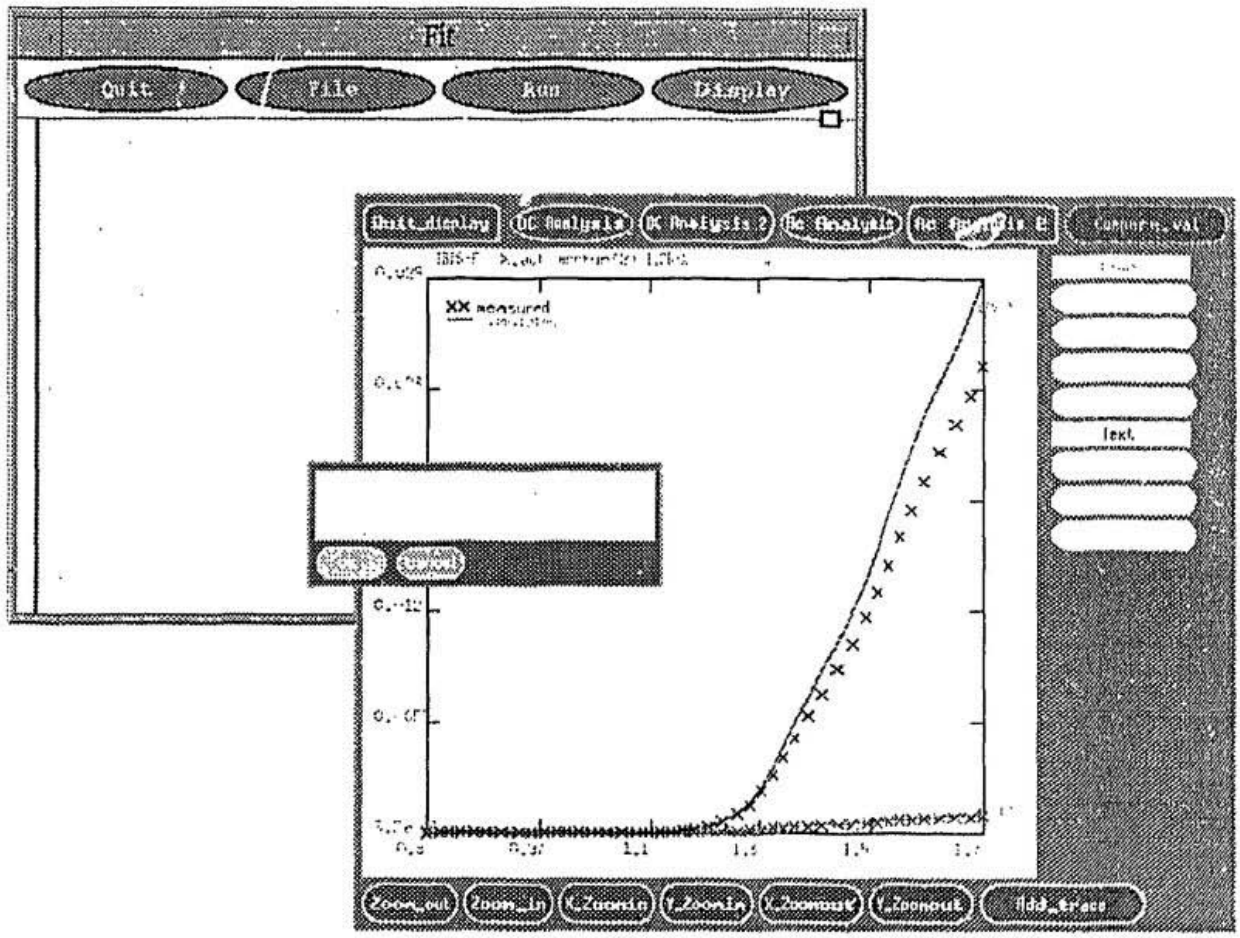

Fig. 1.2. A simple example.

Chapter 2

X Window System and Programming With X

2.1 The X Window System

2.1.1 Components of the X Window System

The X Window System operates with a bit-mapped graphic-display terminal. This type of terminal allows each individual *pixel* on the screen to be accessed and used to display a specific color or shade of gray. The pixels are the basic elements used to construct graphical images on the screen.

Communication between an X Window-based application (called a *client*) and a bit mapped graphic-display terminal is accomplished through a special software called the *X server*. The X client makes requests to the X server to receive input from the terminal's mouse or keyboard and to produce output on the terminal's screen. The X server is a true server program in the sense that it acts as an intermediary for any

client application that wants to use the resource of a graphic-display terminal.

Communication between the client and the server is accomplished using a special communication method called the *X protocol*. This protocol is a network transparent protocol that is used to send data between X clients and an X server. Through the X protocol, an X client can send requests to any server running on any computer connected to the network and cause its outputs and inputs to be sent to and received from any terminal to which that server is connected.

From the X client application developer's point of view, the X protocol is implemented as a series of *language bindings* or functions libraries. The library that implements the X protocol for the C programming language is called *Xlib*. Fig.2.1 shows the relationship between the various parts of the X Window system.

Xlib contains nearly 300 functions to create, move, resize, stack, and destroy windows; to draw lines, rectangles, arcs, and polygons; to use fonts, colormaps, graphic images, and cursors; and to execute a wide variety of other operations.

As can be seen in Fig.2.1, the X library is not the only library that is used to create an X Window client application. Programming with only Xlib has been compared to programming in an assembler language. One can get the basic tasks accomplished in Xlib, but the amount of code needed to produce a simple window with some text on the screen can amount to hundreds of lines. Because of this, another, higher-level, library is often used in combination with Xlib.

2.1.2 X Toolkit

X Toolkit, or *Xt* for short (sometimes referred to as *Intrinsic*), is a set of functions that X programmers can use to write X Window applications at a higher level. Usually,

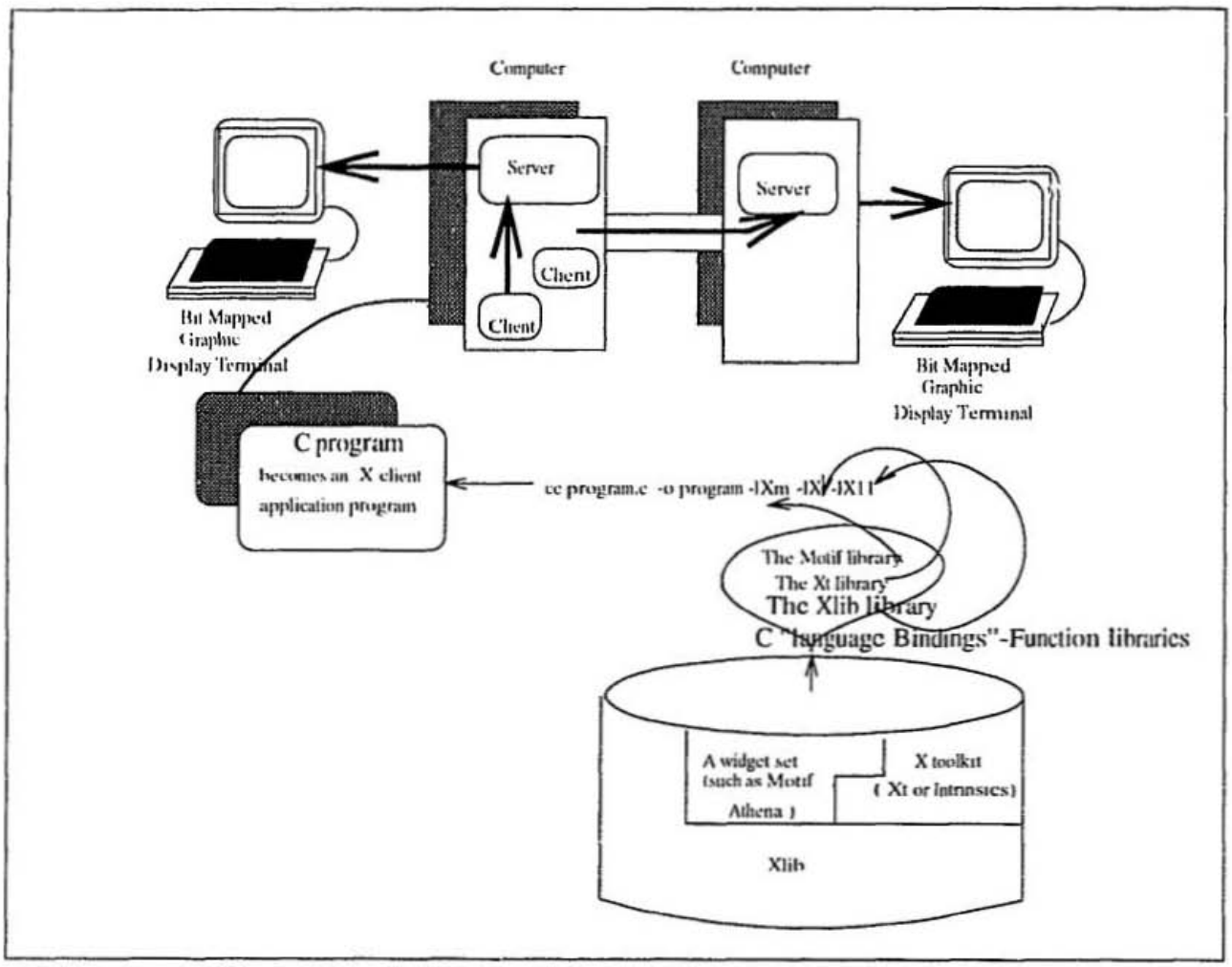

Fig.2.1. Relationship between components of the X Window system.

one Xt function call will translate to several Xlib function calls. In fact, most of the more common Xlib function sequences have their Xt function equivalents.

There are hundreds of Xt functions to perform typical X operations, to communicate input events from specific windows back to the client applications, to deal with events in the event queue, to perform interclient communication, to create and manage user interface objects called *widgets*.

A widget is a collection of one or more windows that are laid out in a way to form a graphics object. A widget definition also contains a set of procedures or functions that are invoked as a result of user input in the widget's window. Push buttons, scroll bars, text boxes, menus, and dialog boxes are all examples of widgets.

X Toolkit provides only a few specific widgets to use in (client) application programs. Consequently, when using Xt, the application developer will usually use a set of separately developed widgets referred to as a *widget set*.

2.1.3 Widget Sets

Widget sets are simply collections of graphics objects that are common to many GUI programs. Widget sets are not included in the X Window system. They are available from *MIT* (the Athena Widget set), *AT&T* (the Open Look widget set), *the Open Software Foundation* (the Motif widget set), and other developers.

Widget sets provide similar basic types of functionalities. Almost all widget sets provide push buttons, labels, text boxes, scroll bars, drawing areas, menus, and so on. The main differences between the widget sets are in the form of their *look and feel*.

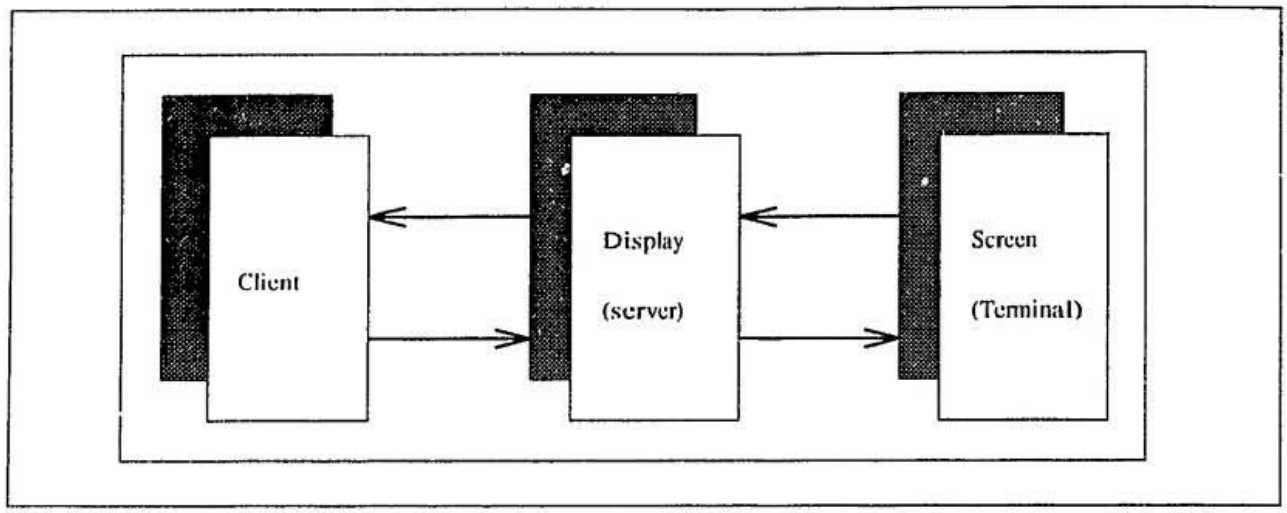

Fig.2.2. Client, server, display and screen in X.

2.1.4 Client, Server, Display and Screen

Whenever X refers to a display, it actually refers to a server. X uses the term *display* as another name for the server. Sending information to a display means using the X protocol for this transfer.

Whenever X refers to a screen, it means one of the physical screens on the terminal controlled by a server. When the server draws on the screen, it is satisfying an X protocol request from some X client application to cause some type of graphics object to appear on a terminal screen. Fig.2.2 illustrates these concepts.

Again, in X, a client application sends X protocol requests to the display. In response, the display causes the graphic-user interface to be drawn on a screen. When the user enters information using the mouse or keyboard, input events are sent from the screen to the display and the display sends the events to the appropriate client application.

The client application is executed within an infinite loop, waiting for input events from the screen. The server responds to the requests for services and passes the input

events to the clients.

2.1.5 X Window Concepts

An X window is a rectangular section of a terminal screen. A window is defined by a border, a background color or pattern, an X/Y-coordinate of its origin, a height, and a width. Whenever the X server takes control of a particular terminal, it installs a special window called the *root window* on the terminal screen.

Any new windows is installed within the root window as a *child* of the root window. Each child of the root window can be a simple window or a more complex window with children of its own. In fact, windows displayed on a particular screen in an X window application form a hierarchy as shown in Fig.2.3.

A window is actually allocated as a data structure within the X server. The X client holds an identifier of a particular window data structure. The window does not actually become visible on the terminal screen until the client tells the server to *map* the window.

When the client requests that a window be mapped, the server issues drawing instructions to cause a graphic representation to appear on the screen.

However, if the window, which is a child of some other window in the window hierarchy, is being mapped, it will not become visible itself until its parent has been mapped. Every ancestor of a window must be mapped and visible before the child's window can become visible. A window can be mapped in such a way that it will partially or completely obscure another window on the screen.

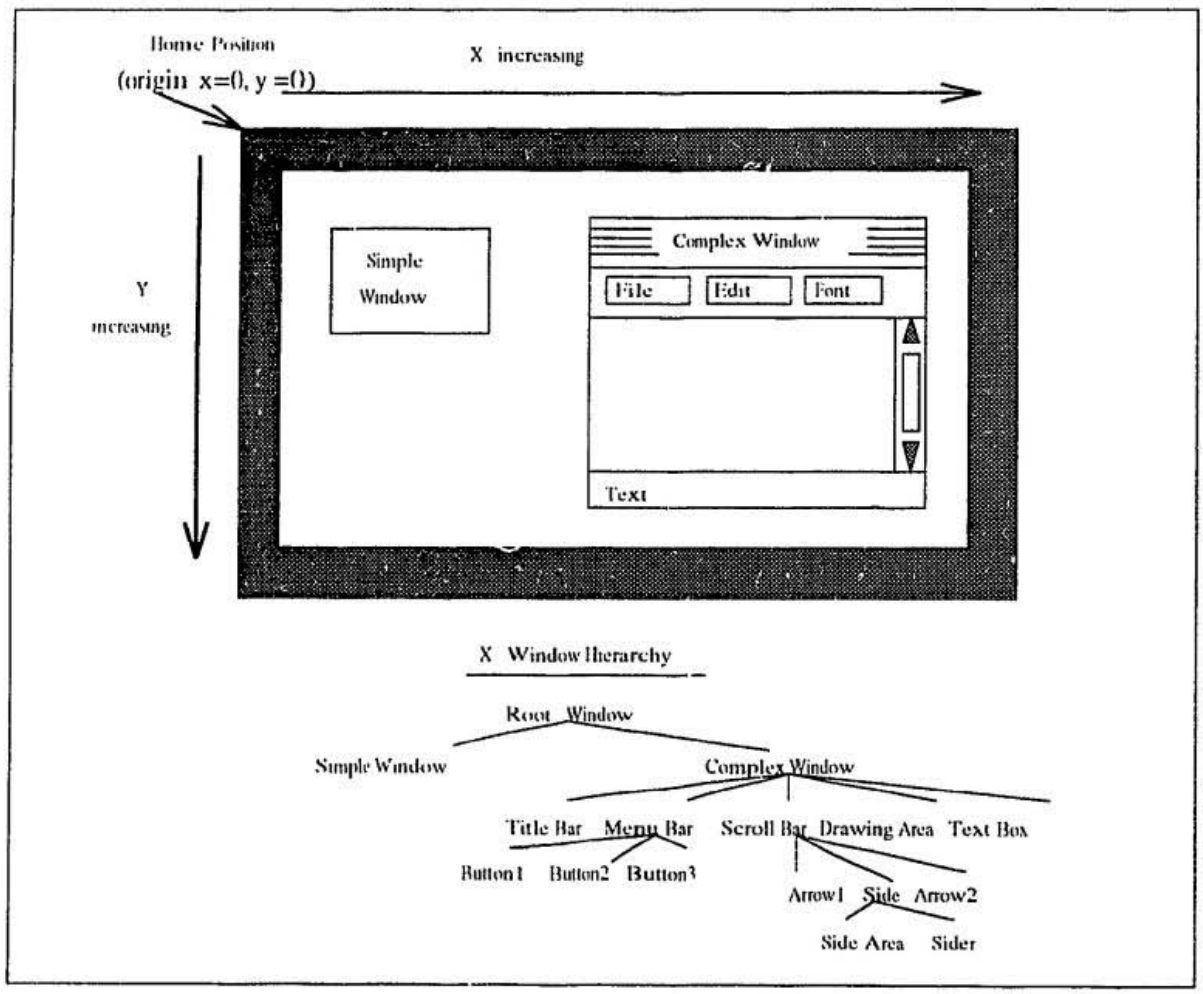

Fig.2.3. A typical X Window hierarchy.

2.1.6 Window Manager

The window manager is an important part of any X Window system. It is the window manager's responsibility to "manage" the terminal's screen "real estate". This means, that it is up to the window manager program to decide where and how new windows are placed on the root window. Typically, window managers also give users some additional control to manipulate the windows on the screen. Most window managers allow users to move input focus from one location to another, change the size of a window, and move input focus from one window to another. Many window managers also allow users to change the stacking order of the windows on the screen to make some previously hidden window visible again.

A window manager is just another X client. However, this client is given special privileges that allow it to intercept certain Xlib calls and deal with them internally. The window manager uses the same Xlib calls that any other client does, but it usually is developed to work with a specific widget set. So, in general, the Athena window manager is used with the Athena widget set and the Motif manager is used with the Motif widget set.

2.2 Graphic-User Interface Objects - Widgets

2.2.1 X and Object-Oriented Programming

X and Athena use *object-oriented* programming techniques to organize and classify the widgets in a way that will make them more useful to the application developer. The object (a widget in this case) is a set of procedures that can be thought of as a

"black box". It will accept specific inputs, perform operations on those inputs, and produce some related outputs in response.

Some goals of any object-oriented technology are:

1. To improve productivity by increasing software extensibility.

   Although the objects in X are predefined for a specific purpose, there are *hooks* into the objects which allow users to "extend" their basic operations with user supplied code

2. To improve productivity by increasing software reusability.

   The programmer doesn't have to "reinvent the wheel"; predefined objects are created for the most common functions needed in typical programs.

3. To control the complexity of software.

   To meet the demands of today's requirements, software systems have become more complex than ever before - especially with the needs of graphic-user interfaces and database systems. This complexity must be controlled so that the cost of software maintenance can be kept to a minimum.

4. To control the cost of software maintenance.

The features of an *object-oriented programming* system include:

- Data abstraction

  An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

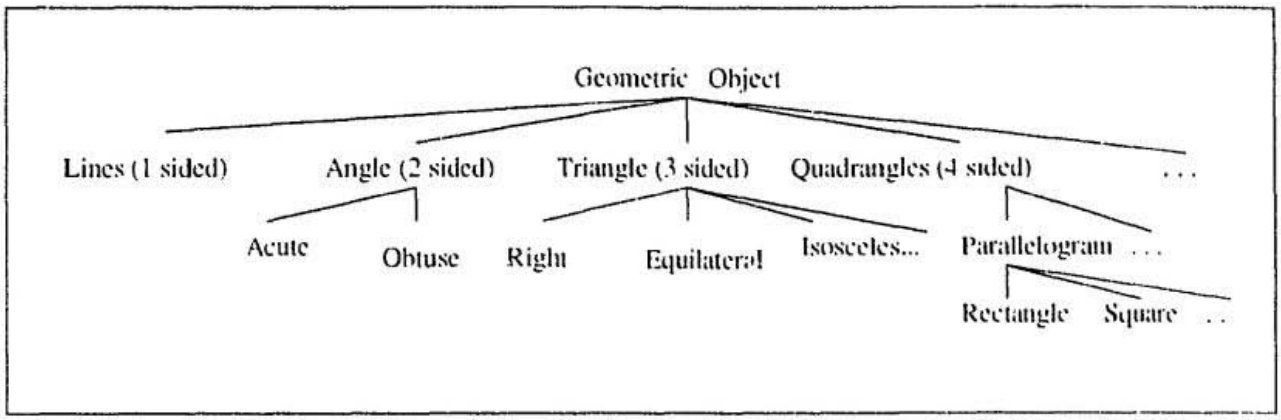

Fig.2.4. Abstract data type class hierarchy.

- Encapsulation

  Encapsulation is the process of hiding all those details of an object that do not contribute to its essential characteristics.

- Inheritance

  Inheritance is a relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes.

Fig.2.4 shows an instance of an object of type square which inherits its own local copy of all the features of square (such as side length) as well as features of type parallelogram (such as parallel sides), type quadrangle, and so on. If an instance of an object of type geometric object has specific attributes, an instance of an object of square will inherit the same attributes.

2.2.2 Widget Concepts

A widget is an object - an abstract data type. It is a collection of one or more X windows held together with a geometry for a specific look and a set of procedures that implement relevant operations. Both Athena and Xt have abstract data types (widgets) for a wide variety of user interface objects that can be used to accept input or supply output for a graphic-user interface.

To use a widget in the program, an instance of the widget data type is defined as an object that the program can use. As many instances of a widget can be defined as are needed by the application program. Each individual instance of the widget is a separate occurrence of an object of that widget data type. Each instance has its own appearance and purpose, which is completely separate and different from every other occurrence of that widget type in the program.

Each widget belongs to a class of related widgets that is organized into a hierarchy. When an instance of a widget is defined, it inherits attributes from all of its parents, grandparents, and other ancestors, all the way up to the root of the hierarchy.

2.2.3 Athena Widgets

The widgets from the Athena widget set which are used in the implementation of the graphical back end package:

- **Simple Widgets.** Each of these widgets performs a specific interface function. They are *simple* because they cannot have widget children - they may only be used as leaves in the widget tree. These widgets display information or handle a small amount of user input.

Command - a push button that, when selected, causes a specific action to take place. This widget can display a multi-line string or a bitmap image.

Grip - a rectangle that, when selected, causes an action to take place.

Label - a rectangle that may contain one or more lines of text or a bitmap image.

List - a list of text strings, presented in row column format that may be individually selected. When an element is selected, an associated action takes place.

Panner - a rectangular area containing a *slider* that may be moved in two dimensions. Notification of movement may be continuous or discrete.

Repeater - a push button that triggers an action at an increasing rate when selected.

Scrollbar - a rectangular area containing a *thumb* that, when slid along one dimension, causes a specific action to take place. A scrollbar may be oriented horizontally or vertically.

Simple - the base class for most of the simple widgets. It provides a rectangular area with a settable mouse cursor and a distinguished border.

- **Menus.** The Athena widget set provides single paned non-hierarchical pop up and pull down menus. There are three classes of *Sme objects* that may be used to build menus.

  Sme - the base class of all menu entries. It may be used as a menu entry to provide blank space.

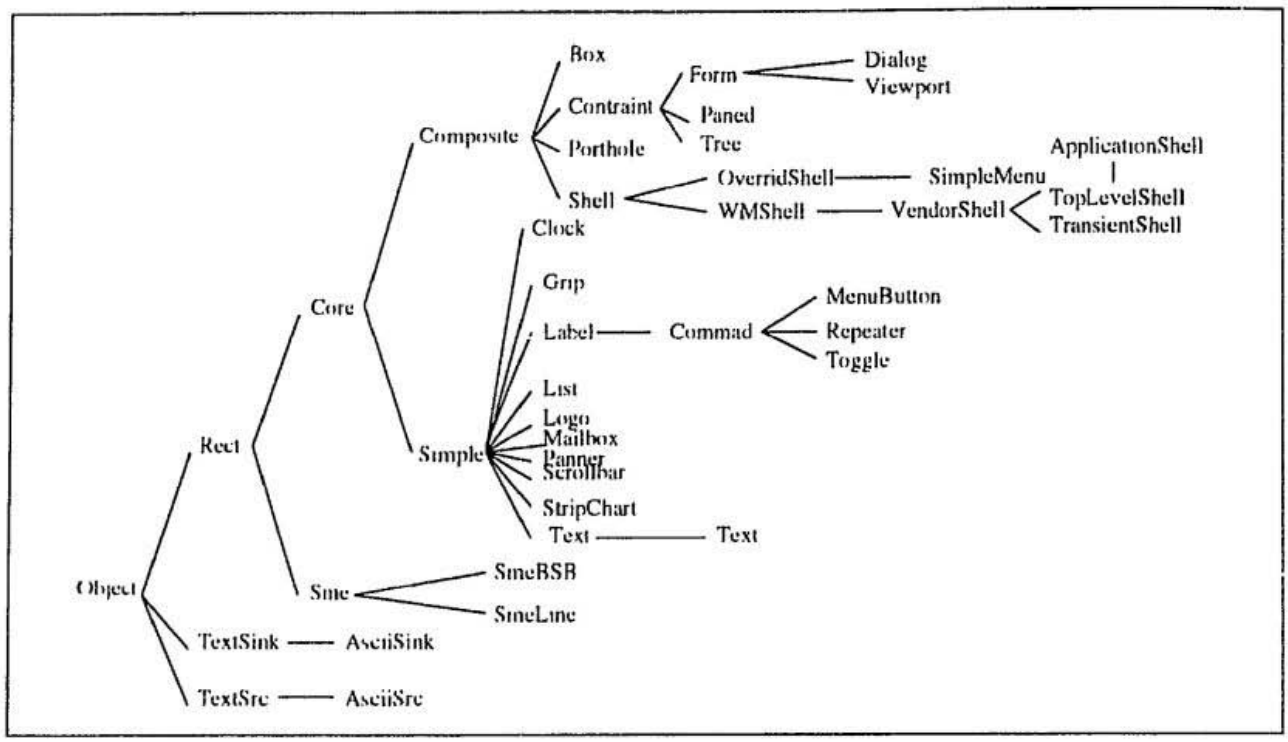

Fig.2.5. X/Athena widget class tree.

SmeBSB - this menu entry provides a selectable entry containing a text string. A bitmap can be placed in the left and right margins.

SmeLine - this menu entry provides an unselected entry containing a separator line.

- **Text Widgets.** The Text widget provides a window that will allow an application to display and edit one or more lines of text. Options are provided to allow the user to add scrollbars to the window, search for a specific string, and modify the text in the buffer.

  The Text widget is made up of a number of components. The modularization of functionality is intended to ease the customization. For most applications, the AsciiText widget is general enough to meet programmers' needs. More

flexibility, special features, or extra functionality can be added by implementing a new TextScore or TextSink widget, or by subclassing the Text widget.

- **Composite and constraint Widgets.**

  Box – this widget packs its children as tightly as possible in non-overlapping rows.

  Dialog – an implementation of a commonly used interaction widget which prompts the user for auxiliary input such as a filename.

  Form – a more sophisticated layout widget that allows its children to specify their positions relative to other children, or to the edges of the form.

  Paned – allows children to be tiled vertically or horizontally. Controls are also provided to allow the user to dynamically resize the individual panes.

  Porthole – allows viewing of a managed child which is as large as, or larger than its parent, typically under control of a Panner widget.

  Tree – provides geometry management of widgets arranged in a directed, acyclic graph.

  Viewport – consists of a frame, one or two scrollbars, and an inner window. The inner window can contain all the data that needs to be displayed. This inner window is clipped by the frame with the scrollbars controlling which section of the inner window is currently visible.

2.2.4 The X/Athena Widget Class Tree

Fig.2.5 shows the organization of the X/Athena widget class tree. This tree shows how widgets are organized into related classes. When an instance of a particular widget is encapsulated into an occurrence of a specific object in the program, it inherits attributes and features from all the widget classes that appear above it in the widget class tree.

Chapter 3

The Design of Graphical Interfaces

Several computer-aided design software tools have been developed which lack flexible user interfaces. SPICE-PAC and FIT are two of them. Since the user interface of a system is often the yardstick by which the system is judged, a flexible, simple and easy to use interface is an important aspect of the design. An interface which is difficult to use will, at best, result in numerous user errors. At worst, it will cause the software system to be discarded, irrespective of its functionality.

A badly designed interface can cause the user to make unnecessary and irritating errors. If information is presented in a confusing or misleading way, the user may misunderstand the meaning of an item of information and initiate a sequence of unwanted actions. From this point of view, the user interface is an important part of any software system.

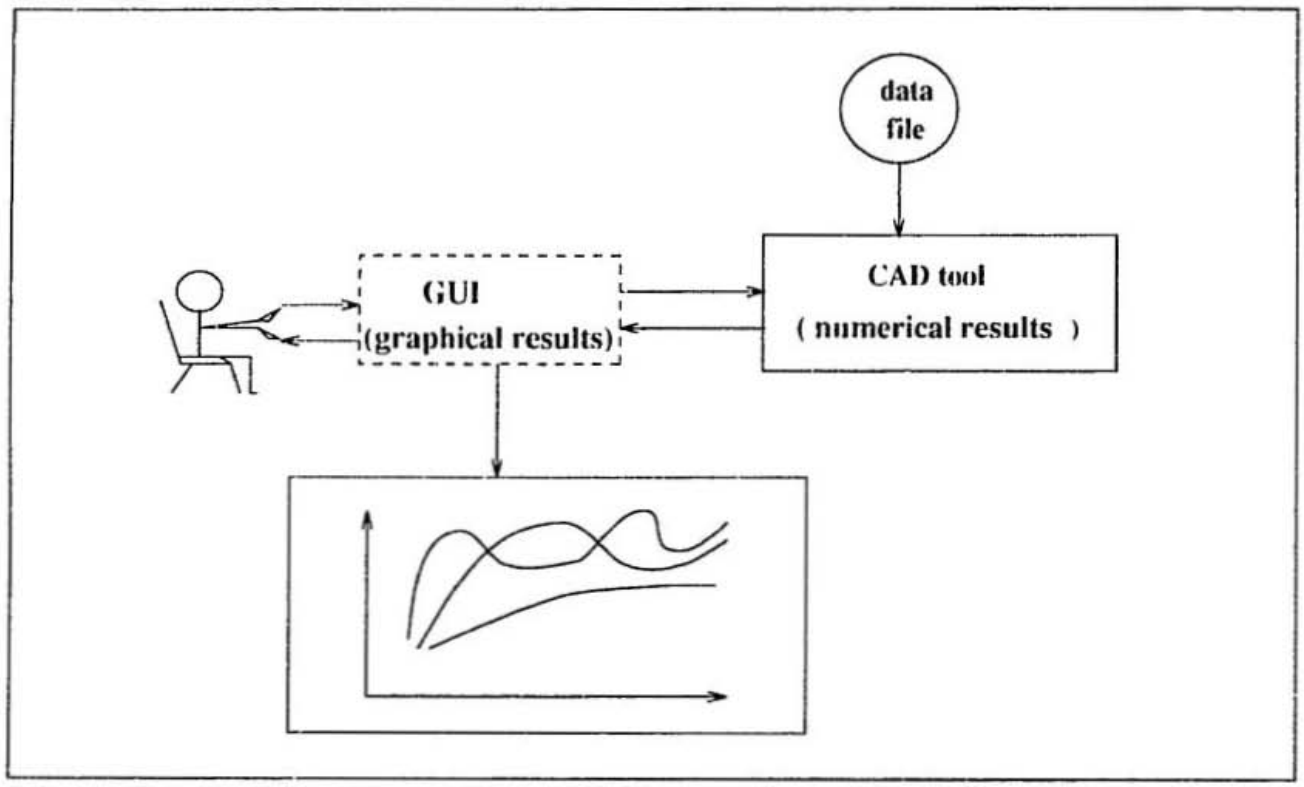

Fig.3.1. Interaction with a GUI.

3.1 Graphical Interface for CAD Tools

A graphical interface for SPICE-PAC/FIT provides a bridge linking the user with application programs which can make the users more confident. With the graphical user interface users are in complete control of what they do and when they do it. The organization of the interaction with a GUI is shown in Fig.3.1.

3.2 SPICE-PAC

SPICE-PAC is a simulation package that is upwardly compatible with the popular SPICE-2G circuits simulator. It accepts the same circuit description language (with only a few minor exceptions) and provides the same circuit analyses as SPICE, but it also supports a number of extensions and refinements which are not available in the original SPICE program. The most important difference between SPICE and SPICE-PAC is, however, in their internal organizations; SPICE is a program with one. fixed sequence of operations while SPICE-PAC is a collection of loosely coupled simulation "primitives" that can be composed in many different ways, as required by a particular application.

This flexibility of SPICE-PAC is quite important in "integrated" applications, i.e., applications in which circuit simulation is combined with other software tools, for example, optimization methods, statistical analysis, symbolic simulation, high-level (e.g., behavioral) simulation, and so on.

3.3 FIT

FIT is an interactive program for extraction of device parameters for SPICE-like circuit simulators. FIT is a simulation-based extractor, so explicit model equations need not be known as they are provided by the circuit simulation tool used, fitting can be performed not only for single devices but for functional blocks or whole circuits as well, and the same extractor can be used for a variety of devices and/or device models. The extractor supports numerical as well as symbolic simulation, so repeated analyses of linearized circuit (for frequency domain analyses) can be performed very efficiently using the symbolic functions generated from the Coates flowgraph representation of the circuit. Several optimization methods are built into the program to provide robust as well as efficient fitting of device characteristics. Flexibility is obtained by specification of extraction details in the data sets rather than the extraction procedure.

FIT is iterative, simulation-based and data-driven. The data-driven capability allows integrated parameter extraction [12] as well as selective extraction, performed on subsets of measurement data and subsets of parameters. Different extraction strategies can thus be developed for different types of devices and/or their models in order to perform the extraction of parameters efficiently.

3.4 Outline of the Graphical Interface

The graphical back-end package for SPICE-PAC/FIT consists of four parts. Fig.3.2 shows its general structure.

The **Quit** part is used to terminate execution of an application.

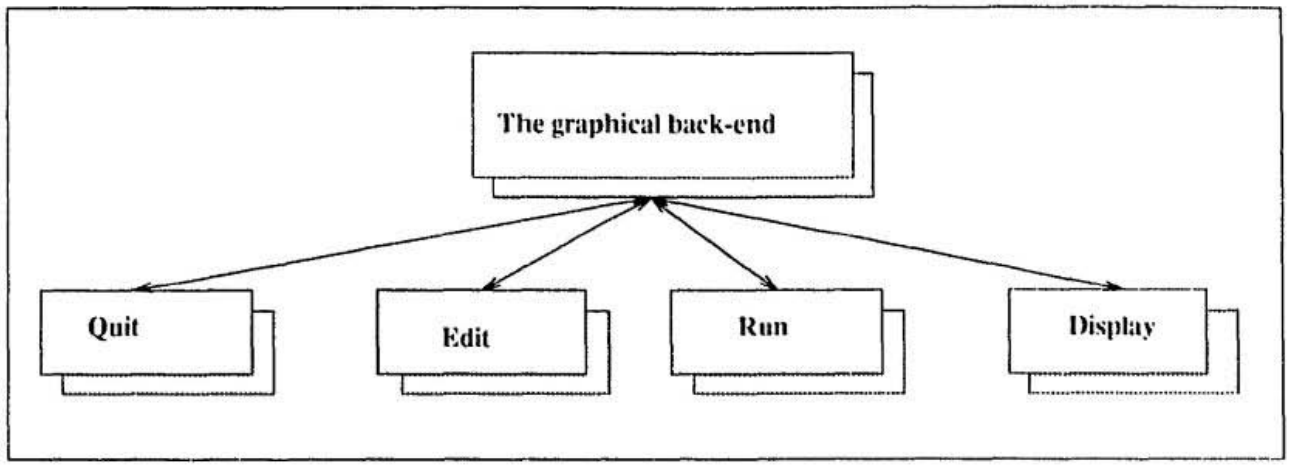

Fig.3.2. Main structure of the graphical interface.

The **Edit** part is provided to manipulate data files; it can load a file from different directories, edit a file and save or print a file. The edit part can distinguish the different data files such as SPICE-PAC data files or FIT data files.

The **Run** part executes the SPICE-PAC/FIT programs, provided that the required input and output data files are available.

The **Display** part displays data created by the SPICE-PAC and FIT programs. Graphical results can be presented either using default properties or properties selected by the user (color, line style, text color or text font). Results can be zoomed in and zoomed out. This part also allows the users to trace specific output variables, compare the results obtained from different analyses, etc.

3.5 Design Guidelines for Graphical Interfaces

One reason why Xt applications are so successful is that they follow the graphic-user interface design guidelines. A short overview of these guidelines follows

Every Xt application should be written in a way that gives the user the controls

needed to accomplish a given task. Users will have a feeling of control over an application if it is consistent and gives them the ability to directly manipulate the controls and other objects of the application. The application must also be flexible and allow the user to decide on any action that may have irreversible effects.

- **Consistency**

  Consistency means that similar controls will operate in a similar manner and have similar effects. If a Pushbutton in one application has a Label and clicking that button causes a DialogBox to pop up, any other application that uses the ellipsis on a pushbutton Label should do the same thing. Consistency also means that the same action will always produce the same result in many different applications. If you can click-hold-drag the title bar of one application to move its location on the screen, then any other application that has a title bar ought to work the same way.

  Another factor in the consistency of an application is placement of the user controls in the application's window. The controls and functions that are used most often should be presented first, at the top of the application, in a logical and straightforward order.

  Functions, which are not used frequently, should be hidden and only called up on an as-needed basis.

- **Direct manipulation**

  The experience of direct manipulation is defined as the connection of a user action in an application with an observable response from the application. In a

direct-manipulation user interface, users will experience the immediate visible results of their actions.

Immediate visible response is the most important aspect of the direct manipulation experience. The performance problems of slow hardware or of poor program design and implementation can make it difficult for a user to concentrate on the task for which the X application is used. No matter how interesting an application may look, if it is slow in its operation, it will be practically useless to customers.

- **Flexibilty**

  Each Xt application should be flexible with respect to the way the user chooses to interact with the application and flexiible in allowing the user to configure aspects of the application to fit his personal preferences.

  An Xt application should provide more than one way to get a task accomplished, so the user can choose the method that is most convenient to him. A user may point to a PushButton on a MenuBar, use a PulldownMenu, or select a PushButton to cause some action to take place. However, in addition to using the mouse to post a menu, the application can also define a keyboard character that will post the PullDown and another character to activate the desired option.

  A user should also be given some amount of control over the visual appearance of his applications. Not all users like a light-blue or pink background; some users may prefer gray PushButtons, while other users may like to see icons instead of Labels on their button widgets, etc.

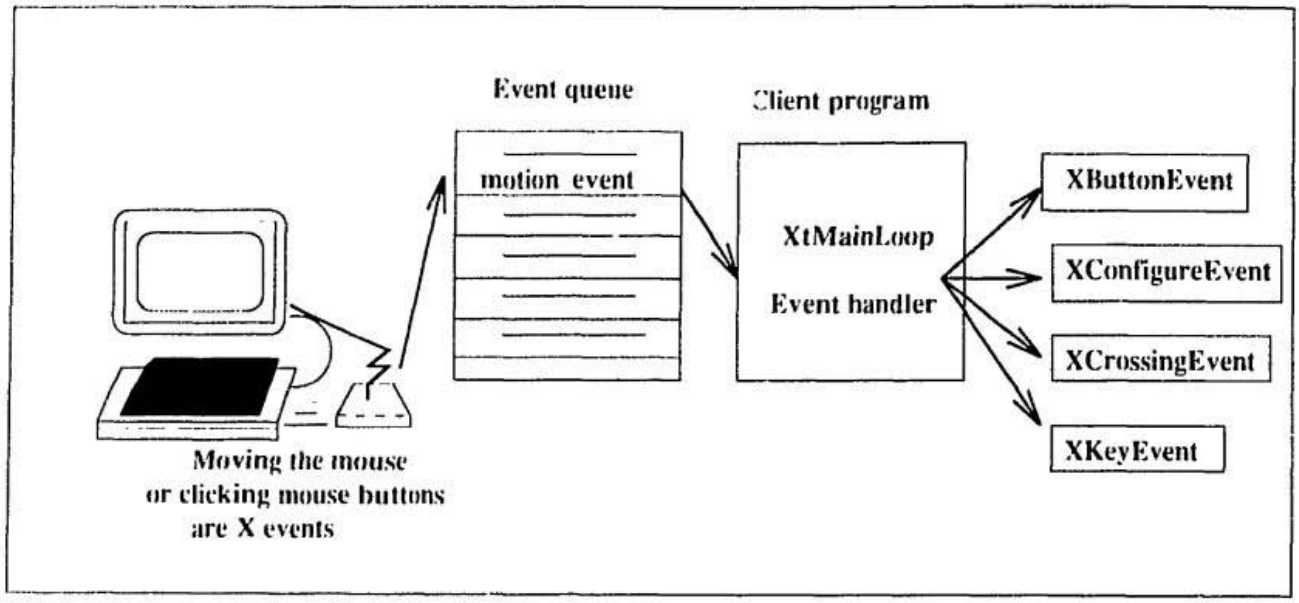

Fig.3.3. X event processing.

Explicit control is an important issue for those operations within the application that have irreversible effects. Before the user is allowed to destroy an object close a file before saving the changes, remove a file, or any other similar action the user should be given a chance to reconsider the action. In such cases, a DialogBox should pop up to require a confirmation or to dismiss the action.

3.6 Designing Xt Applications

X Window applications are event-driven programs. The structure of an event-driven program can be presented as a loop with a case structure in it. In each execution of the loop, an event is fetched from the event queue and used for selection of an action.

X events are the basic means of communication between a user and an application. X events are generated as a result of user input from the keyboard or the mouse, as shown in Fig.3.3.

```
eventloop()
{
    ...
    ...
    XNextEvent( theDisplay, &theEvent);
    swith ( theEvent.type)
            {
            case Expose:
                    Process when window is exposed    ;
                    break;
            case: ButtonPress:
                    Process when key was pressed;
                    break;
            case: Keypress:
                    Process when window is exposed;
                    break;
             }
  }
```

Fig.3.4. A simple example of the X event loop.

When an event is received from a client, the X server stores it in an event queue. Each client application has its own event queue to hold the events in the first-in/first-out (FIFO) order. When the client application needs a next input event from the user, it fetches it from the queue, decides which window (not widget) it belongs to, and executes the window-specific code for this event. At this point, the application-specific code takes over the processing of the event.

X client application programs are event driven — they are organized as infinite loops, waiting for input events, and executing actions corresponding to these events, as shown in Fig.3.4.

3.7 Design Process

The first stage in establishing a system data flow is to formulate a model of the "real world" entities which are represented in the system.

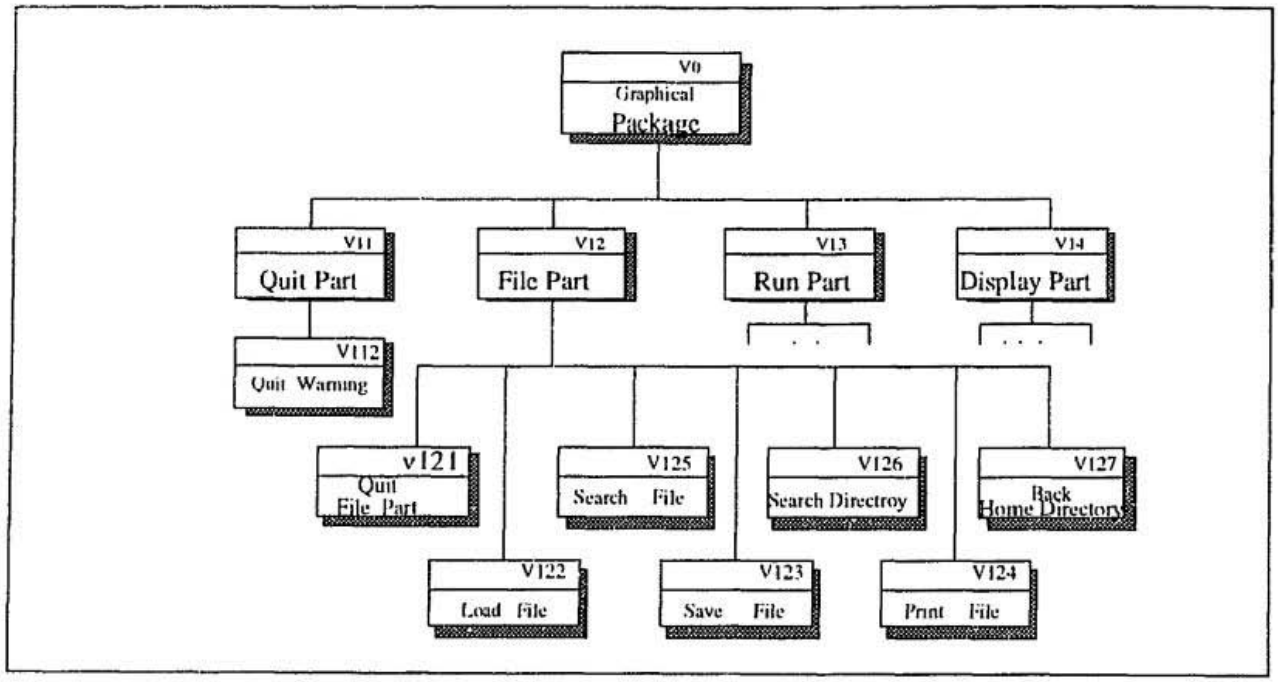

Fig.3.5. Main components of the graphical package.

3.7.1 Viewpoint Analysis

When formulating a system model, viewpoint structuring is an important stage which imposes a structure on the identified viewpoint clusters and represents this in a viewpoint structure diagram. Fig.3.5 shows the main components of the graphical back-end package and Fig.3.6 shows the main components of the **Display** part.

3.7.2 Identifying Main Functions

The next step of the building process is to identify the functions. The descriptions of main functions which support the layout of a GUI and the corresponding callback functions are as follows:

- *Function:* CheckFileType.

  Input: Path and name of the file.

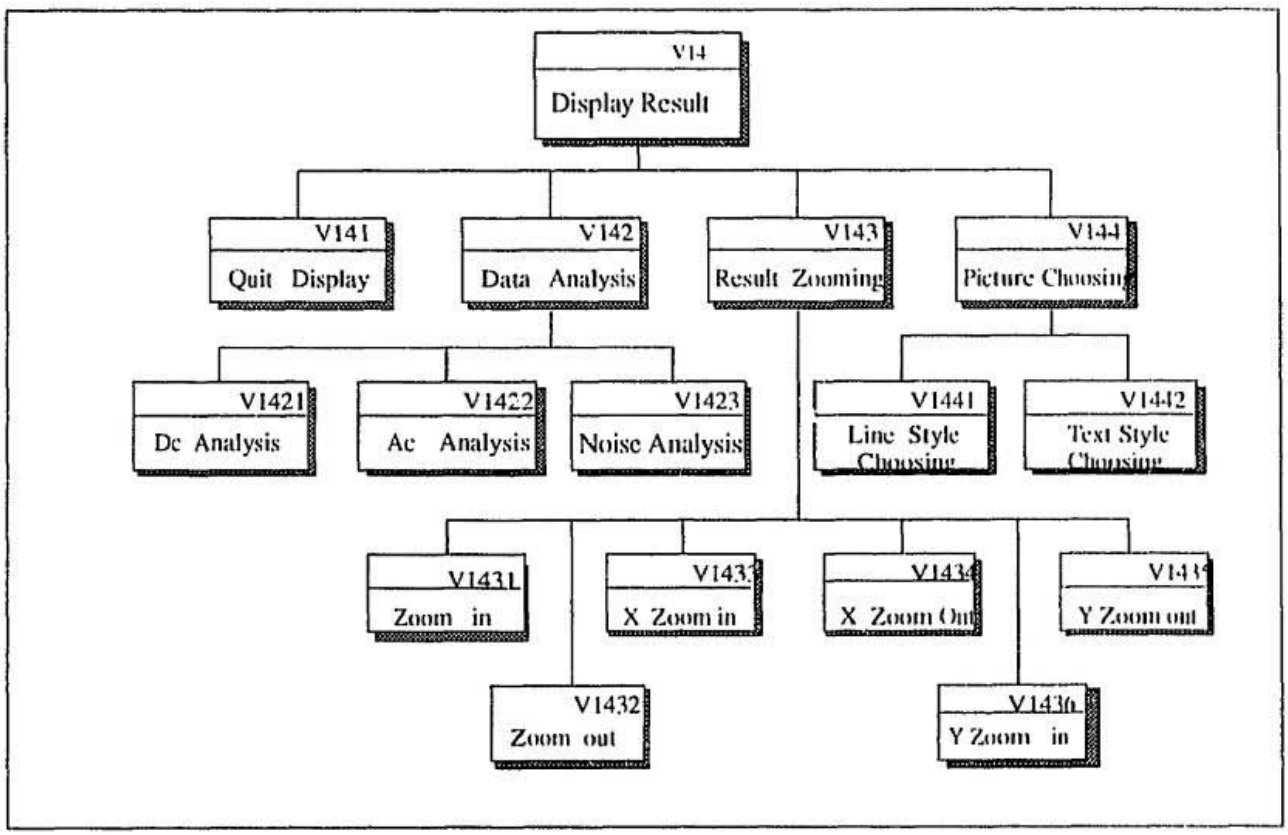

Fig.3.6. Main components of the Display part.

Output: Type of the file.

Description: CheckFileType returns an indication EXECUTABLE, DIRECTORY or NORMAL file, based on the type of the file returned by the stat() function. This function is system dependent.

- *Function:* FillFileList.

  Input: File list that holds file names, Dir list that holds subdirectory names, Label widget that holds current pathname and Current which is the current directory pathname.

  Output: None.

  Description: Fills in the directory and the file lists with the names of the directories and files in a given directory (Current). The Label widget is updated to the current pathname.

- *Function:* ChangeToParent.

  Input: Pathname.

  Output: None.

  Description: Changes the pathname to the parent's pathname; for example, it changes "/dir1/dir2/dir3/" to "/dir1/dir2/" .

- *Function:* DirCallback.

  Input: Widget, Client data and List data.

  Output: None.

  Description: Loads the files in a new directory.

- *Function:* GetCurrentDirectory.

  Input: Pathname.

  Output: None.

  Description: GetCurrentDirectory fills the pathname with the current directory name, or with "/" for the root directory to indicate a failure. If successful, it appends "/" to the pathname.

- *Function:* ReadFitData.

  Input: The name of the output file.

  Output: None.

  Description: Opens data file and reads the data into different data files. Intermediate data files, such as *AC file*, *DC file* are created.

- *Function:* Read AC file.

  Input: The name of input file.

  Output: None.

  Description: Opens the *AC file* and reads the data into corresponding arrays.

- *Function:* CreatePicture.

  Input: Maximum and minimum X and Y coordinate values of the display window.

  Output: None.

  Description: Displays the plot of specific data according to the size of the display window, determines the max and min values of the given data, and

finds the scale factor.

- *Function:* Tracing.

  Input: Widget, Closure, CallData.

  Output: None.

  Description: Checks if the selected variable is correct for the current analysis; if it is correct, a tracing window is popped up with the graphical representation of results, otherwise a warning message is displayed.

- *Function:* Comparevalue.

  Input: Origin X value, origin Y value, maximum X value, maximum Y value.

  Output: None.

  Description: Displays the values of the same output variable for different analyses.

3.7.3 Standardizing the Interface

The idea behind standardizing the user interface is to provide its consistent structure.

Fig.3.7 shows the OSF/Motif style of the window. As can be seen, the resize borders are on the outermost frame of the window (they are placed there by the OSF/Motif window manager). In the upper-left corner there is a button for the system menu, followed by an icon button, and lastly, a min/max button. Below this "system bar" is the application-specific "menu bar". When an item from the menu bar is selected, a menu will pull down beneath it. The "workspace" will hold additional pop up windows or perhaps static display windows.

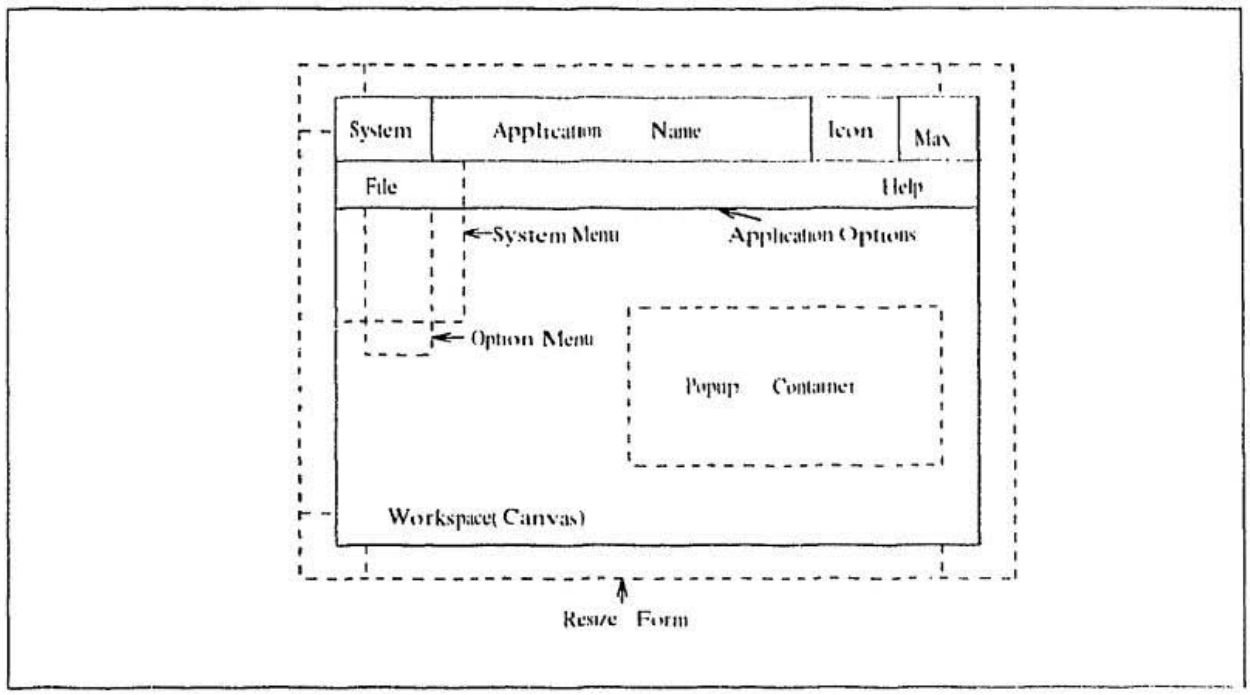

Fig.3.7. OSF/Motif style GUI.

The style adopted for the back-end package is shown in Fig.3.8. It has a title bar in the top of the screen and beneath it a "menu/system bar" with **Quit, File, Run** and **Display** items. The "workspace" for file viewing is the static area under the menu bar. This adopted style of the interface reflects the logical structure of basic capabilities of the programs controlled by the interface.

Program development is streamlined due to the interface "routines" or "objects" which are provided for simplifying the development of interfaces. Moreover, duplication of interface components is eliminated.

3.7.4 Choosing the Widgets

After identifying the main functions and standardizing the graphical interface, the widgets are selected for the application.

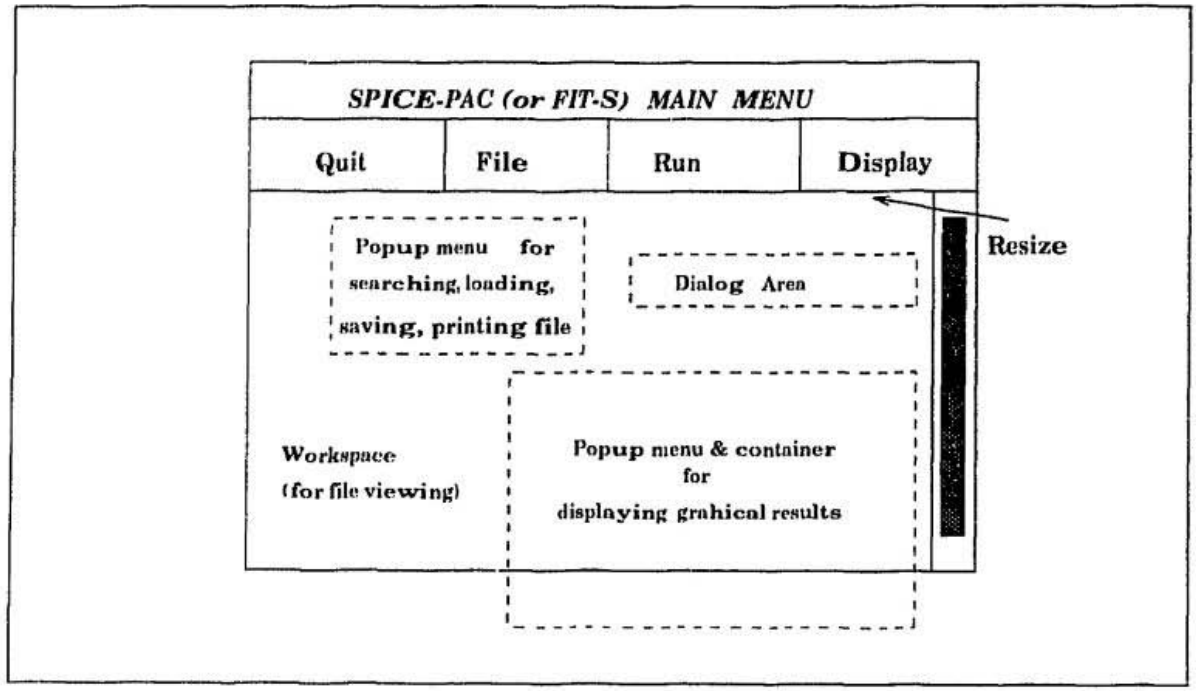

Fig.3.8. Adopted style of GUI.

The Athena widget set is used for the implementation of the interface. The principal reason of selecting the Athena widget set is that it is a commonly available public domain set while Motif is a commercial product.

The detailed descriptions of the widgets used in this application follows.

OverrideWidgetClass

This widget is a member of the class ShellWidgetClass. It is one of the widgets provided by the Intrinsics. Its role is to override window manager. It is used for creating pop-ups.

FormWidgetClass

This widget is a member of the class ConstraintWidgetClass. Essentially, it contains layout policies that are useful for assembling the interface. In this application, it is used for controlling the placement of widgets with respect to other widgets, controlling

resizing of the children and displaying the graphical results.

BoxWidgetClass

This widget is a member of the class CompositeWidgetClass. It assists in the layout, placing constraints on its children; it lays them out by filling in the row first, then continuing down.

ListWidgetClass

This widget is a member of the class SimpleWidgetClass. It provides a list of items and a callback mechanism for selected items.

CommandWidgetClass

This widget is a member of the class LabelWidgetClass. Its role is to provide a callback mechanism for LabelWidgetClass. It is a "button-like" widget.

LabelWidgetClass

This widget is a member of the class SimpleWidgetClass. It provides a textual label (justified) or a pixmap in a window. Its main use is to provide a message to the user. It is used in the menus and in the message area.

TextWidgetClass

The Text widget provides a window which allows an application to display and edit one or more lines of text. Options are provided to allow the users to add scrollbar to its window, search for a specific string, and modify the text in the buffer.

3.7.5 General Requirements

The system menu contains **File, Display and Run** items. The file display area is also contained in the main menu. The file can be viewed as well as edited at the same time.

When the user clicks on the **Quit** button, a pop up dialog with a warning message shows up. When the user clicks on the **File** button, the menu of the file manager pops up with buttons: **Load, Save** and **Print** for performing operations on files, **Cancel** for closing the file main menu, and **Home** for going back to the user's home directory.

There are two lists in the main file menu, one list for files in the current directory, the other list for subdirectories under the current directory. Whenever the user clicks on the file list or types in a file name as *current file name*, the label of the current file name changes immediately. Whenever the user clicks on a directory list, the contents in file list changes to show the files in the new directory. When the **Save** button is selected, the file system saves the current file, and a warning message "Are you sure to change the file?" appears. When the user clicks on **Load**, the file system loads the file in the file display area. When the user chooses **Print**, the current file is printed.

When the user clicks on the **Display** button, the main menu of the **Display** part shows up with a group of command buttons for different analyses, a group of command buttons for choosing the style and color for graphical presentation of results and a set of buttons for zooming the picture. There also is a workspace for displaying the results, and command buttons for comparing results of different analyses. When the user selects comparing results, another display window pops up to present the graphical results of different analyses.

Chapter 4

Building the Interface

Setting the application resources and building the application structure are the initial steps of developing a graphical user interface. The next sections provide a detailed description of these steps.

4.1 Application Resource Setting

4.1.1 Resources

One of the most powerful features of Xt is the *resource manager*, a collection of mechanisms for getting resources from a variety of locations and converting a resource to the correct representation for the client. For example, if we want to set the font for the Athena Label widget, we can simply set a resource XtNfont for the widget Label, and the resource manager will perform all operations which are required to use the required font in strings within this widget.

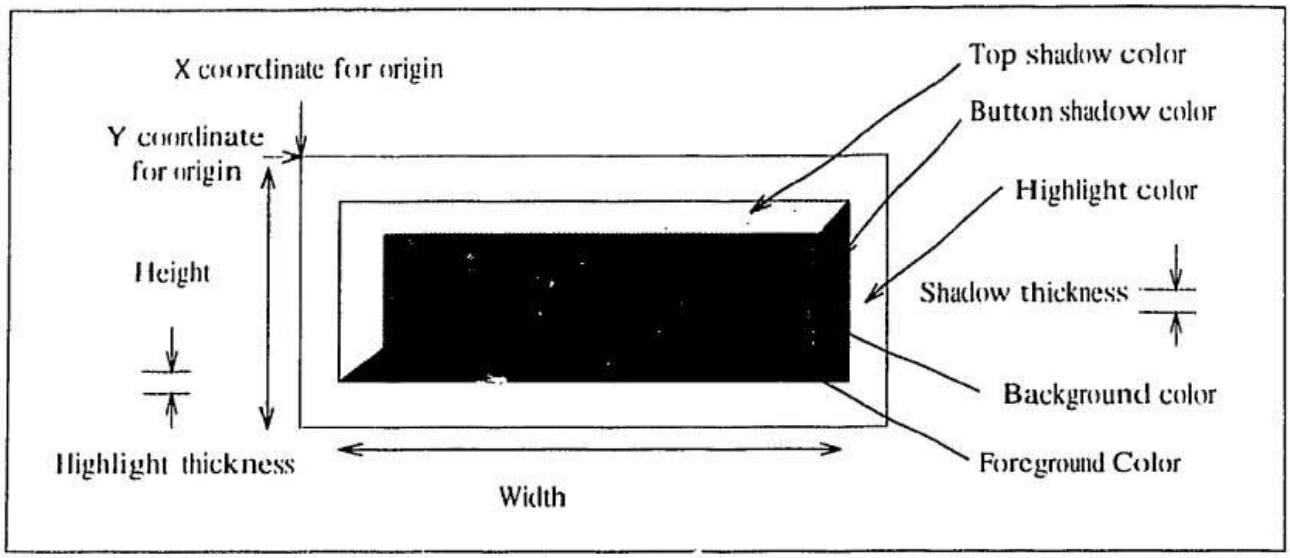

Fig.4.1. Resource example.

Each widget in an X/Athena program has attributes associated with it that define its look-and-feel. These attributes, called *resources*, are one of the main object-oriented features of a widget based-program. Fig.4.1 shows how these resources are related to the PushButton widget.

The widget class tree organizes all the available widgets into related classes. Widgets are grouped together into the same class when they have some attributes in common.

As shown in Fig.2.5, widgets have certain common attributes. These attributes are stored in the "Core" widget class, which is the root of the widget class tree. Whenever an instance of a particular widget is created, it is associated with a copy of all attributes that are related to the widget class. These attributes can be changed either before or after the widget is created; they can also be defined in a special file located externally to the program.

4.1.2 Resource File

The X Toolkit allows the resources of a widget to be specified directly in the source code at the time the program is compiled or at execution time from within a resource file. Using the resource file the user can override the default look-and-feel of some of the widgets in an X/Athena program.

A *resource file* is a text file that can be created with any text editor. Within the resource file, the resources are specified for one or more widgets within one or more X/Athena client application programs. Using a very simple syntax, the resource file specifies a resource name for a widget instance in a particular client along with a value for that resource.

The only requirement for the use of a resource file is that *we must know the instance names and instance hierarchy of the widgets in the client's widget instance tree.*

Using a resource file simplifies the source code of a program, and provides more flexibility in the specification of resources and their values for X/Athena client application programs.

4.1.3 Resource Specification Syntax

Each line within a resource file specifies a resource value for a particular instance of a widget, a group of widgets that have the same instance name, a group of widgets that belong to the same class, or a group of widgets that all have the same resource class.

Any object in a resource specification can be one of two types. With the object we

can associate either a specific client program or a class of similar, related programs. The subobjects in a resource specification show the path through the widget instance tree that must be followed to gain access to a particular widget. A subobject can be either the widget instance name (the second parameter to the widget creation function) or a class name (*instance names always start with a lowercase letter, and class names always start with an uppercase letter.*)

An object and subobjects in a resource specification can be separated from one another by either a period (.) or an asterisk (*). Using a period separator is called a *tight binding* and requires that we know the exact parent instance or class name on the left and the exact child instance or class name on the right. Using an asterisk separator is called a *loose binding* which allows us to indicate that any number of widgets may appear between the parent instance or class on the left and the child instance or class on the right.

The last item in a resource specification is the name of the resource. The resource name is followed by a colon, optional white space, and the value of the resource. Any line that starts with an exclamation point (!) in a resource file is a comment line and is ignored.

A set of *precedence rules* is established in X to resolve potential conflicts between resource specifications which attempt to set the same resource for the same widget. These rules determine which resource specification is taken into account in the case of conflicts. The following rules apply in ascending order:

1. The hierarchy of the instance and class names in a resource specification must match a client hierarchy *exactly* or the line is ignored.

Object(.|*)subobject(.|*)attribute:value

Fig.4.2. Resource specification syntax in resource files.

2. Tight binding takes precedence over loose bindings.

3. Instance names take precedence over class names.

4. Explicit instance or class name takes precedence over omitted instance or class names; for example, the specification "*scrollbar*background" takes precedence over the specification "*background".

5. Left components are more important than right components; for example, the specification "Xterm*background" takes precedence over "*scrollbar*background".

6. If two resource specifications have the same precedence, their physical ordering determines which one takes effect; the specification that appears later takes the precedence.

Resource specification syntax is shown in Fig.4.2.

4.1.4 Resource Database

When a client program starts executing, one of the first operations within the invocation XtInitialize() is to create a *resource database* (from different resource files). Although users usually place their resource descriptions in the file ".Xdefaults" in their home directory, XtInitialize() checks a number of other locations for resource

information. The following is a list of locations that are checked, in ascending order, for resource information:

1. /usr/lib/X11{LANG}app-defaults ⟨class⟩. ⟨class⟩ is the class name supplied as the second parameter to XtInitialize(). If the file does not exist, XtInitialize() looks for the file /usr/lib/X11/app-defaults⟨class⟩. The $LANG environment variable is intended for use by clients that have different resources for different languages; it specifies a subdirectory for a s . language.

2. XAPPLRESLANGPATH ⟨class⟩. ⟨class⟩ is the class name supplied as the second parameter to XtInitialize(). If this file does not exist, XtInitialize() looks for the file XAPPLRESDIR⟨class⟩.

3. RESOURCE_MANAGER property of the root window. This is a special data area that can be manipulated by the *xrdb* utility program. If this data area does not exit, XtInitialize() looks for the file $HOME/.Xdefaults.

4. $XENVIRONMENT. This environment variable contains a full or relative path or a simple file name of a resource file.

5. Command-line arguments. One or more *-xrm* options can be specified on the command line when starting any X/Athena client.

6. The widget's argument lists. The resources are specified in the argument lists of the widget creation and updating functions in the source code of the program.

4.1.5 The Resource File for FIT and SPICE-PAC

In summary, there are five main ways to set the values of widget resources:

1. In a user resource file.

2. In a class resource file, also called an application defaults file.

3. In the RESOURCE_MANAGER property of the root window. This property is created by the standard X Window program called *xrdb*.

4. In command-line parameters passed to the program.

5. Inside a program setting the resource values.

In particular, the file "Fit" contains a few resource definitions for the FIT program. The file "Spice" contains resource definitions for the SPICE-PAC program. Using these files, users can change the definitions to override the default resource file which will affect the layout of the user interface. A large part of these files specifies resources for the color and location of the pop up Display, pop up Help, and so on. These resource files can be located in the user's home directory ($HOME in UNIX parlance) or together with other application default files, in /usr/lib/X11/app-defaults. Fig.4.3 shows a fragment of the file "Fit".

4.2 Application Structure

One of the initial steps in developing an application is to build the application structure. X application structure is fairly well defined, and each client follows this structure to some degree.

Fig.4.4 outlines the structure of a typical application program. There are four header files which are usually used:

```
Fit*background:          pink
Fit*foregroound:         white

Fit*borderWidth:         2

Fit.width:               600
Fit.height:              400

Fit*Quit*font:           -*-courier-bold-r-normal--17-*100-100-*-*iso8859-1
Fit*Quit.shapeStyle:     ellipse
Fit*Quit.background:     indianred

%setting resource for display
Fit*paned. width:        600
Fit*paned.height:        400

Fit*ok.width:            50
Fit*ok.height:           20
Fit*ok.background:       salmon
```

Fig.4.3. A fragment of the resource file for FIT.

```
#include ⟨ X11/Intrinsic.h ⟩

#include ⟨ X11/StringDefs.h ⟩

#include ⟨ Xt/Xatom.h⟩

#include ⟨ Xt/Shell.h⟩
```

The *Intrinsic.h* header file contains the definitions of the functions XtCreateWidget(), the macros that are used by Xt based class names (XtC) and representation type names (XtR) used in Intrinsic. *Xatom.h* contains the predefined X atoms used for inter-client communication and selections. *Shell.h* contains those definitions for application shells which are the outer windows that all clients have.

The steps for writing the application are:

1. Initialize the toolkit (XtInitialize()/XtAppInitialize()).

2. Create the Widget (XtCreateWidget()/XtCreateManagedWidget()).

3. Realize the widgets (XtRealizeWidget()).

4. Wait on events (XtMainLoop()).

```
#include<stdio.h>
#include<X11/Intrinsic.h>
#include<X11/StringDefs.h>

#include"Fit.h"

Forward Declarations

Global Variables

Static definitions

main(argc, argv)

        Initialize Toolkit
            -XtInitialize
            -XtAppInitialize

        Build Interface
          -XtCreateWidget/XtCreateManagedWidget

        XtRealizeWidget

        XtManageChildren/XtManageChild

        XtMainLoop/XtAppMainLoop
```

Fig.4.4. Application structure.

Chapter 5

System Implementation

The implementation of the graphical user interface for FIT and SPICE-PAC consists of four parts: pop-up warning section; the file menu; the analysis section and the display section. All these parts are described in the sections that follow.

5.1 Pop–up Warnings

5.1.1 Widget Hierarchy for Pop–up Warnings

The first step needed to create a pop-up menu is a shell. The *overrideWidgetClass* is used for this purpose. Since pop-ups allow only one child as a direct descendant, so, in most cases, the typical approach is to use one of the container type widgets and then add the children to this container. Fig.5.1 shows this hierarchy of widgets.

The function *XtCreateWidget* is not used in this case. The reason for this is that each widget contains a part that identifies the pop-up children associated with the widget. *XtCreateWidget* does not fit into this structure. Therefore, the Intrinsics

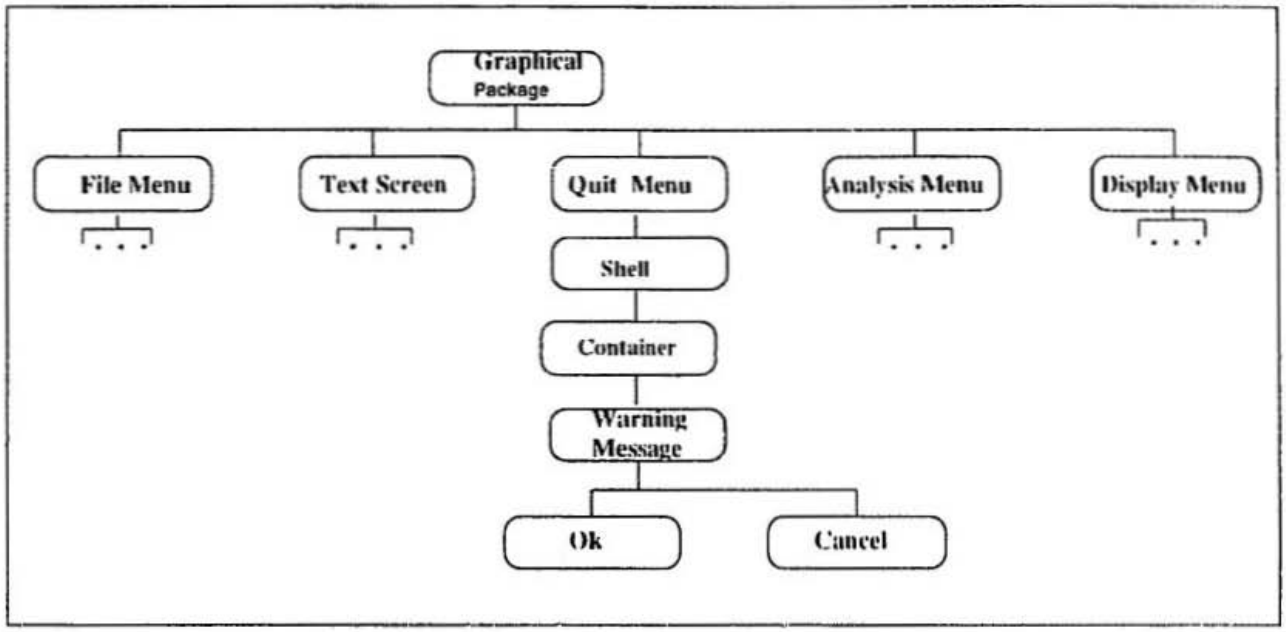

Fig.5.1. Hierarchy of widgets for a pop-up warning message.

provides *XtCreatePopupShell* for creating dialog shells.

The next step is to display a warning message. This message is composed of a container (to hold the panes), the message, and the panes. Each pane needs a label of a callback function that performs an associated action. The warning message in this example uses *formWidgetClass* as a container, *listWidgetClass* for the message, and *commandWidgetClass* for the panes.

5.1.2 Widget interactions for the Pop–up Warnings

Two callback functions, *cancel_quit_callback* and *chom_callback*. *cancel_quit_callback*, are used in the **Quit** section, for canceling the warning message; when the user clicks on the **Cancel** button, the pop-up warning message will disappear and the previous status is reinstated. *chom_callback* is for quitting the application; when the user clicks on the **OK** button, the application is terminated. This is shown in Fig.5.2.

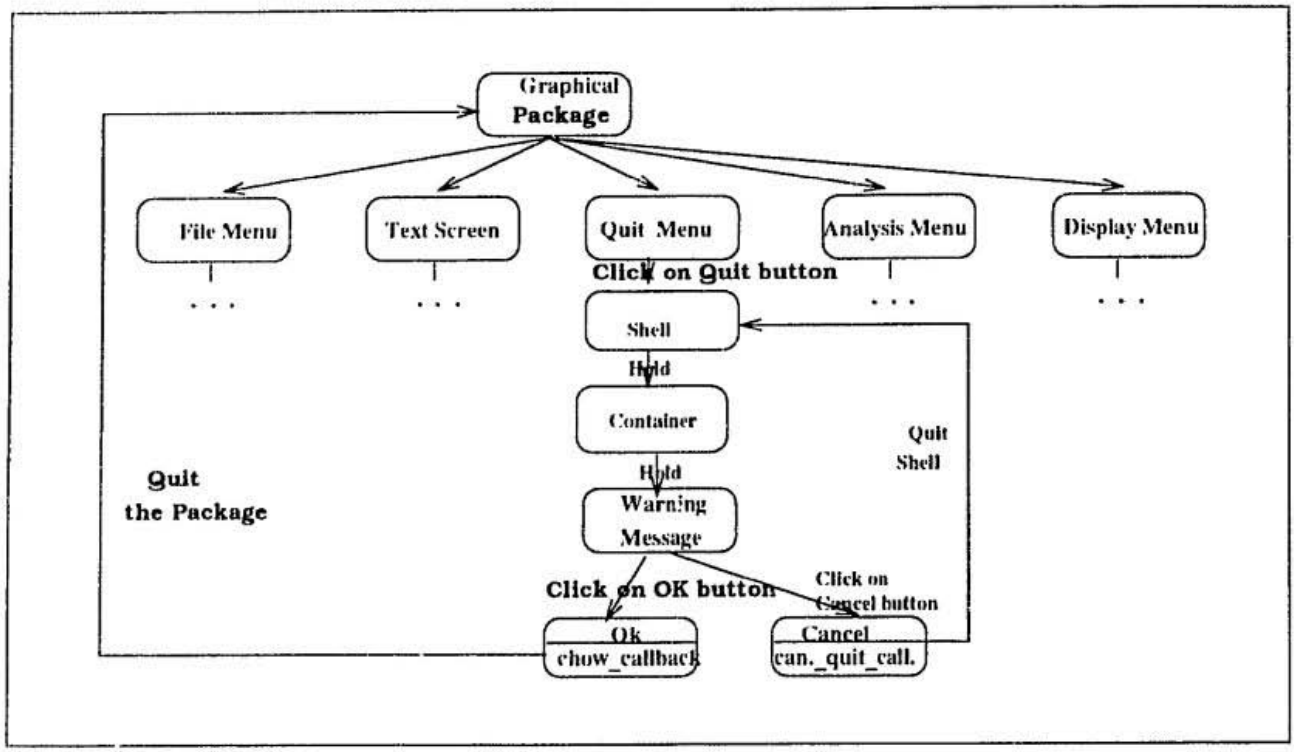

Fig.5.2. Widget interactions for a pop-up warning message.

After creating the widgets for the **Quit** section and their component interactions (callbacks), the look of **Quit** section with a warning message is shown on Fig.5.3.

5.2 Pop–up File Section

The file menu displays the files and subdirectories in a given directory. The user can view the contents of text files by clicking on the **Load** button and can save or print the file by clicking on the **Save** or **Print** button, respectively.

Two scrolled list are used to list a contents of a given directory: one presents the subdirectories and the other lists the files. When a user clicks on any name in the subdirectory list, the files in that subdirectory are listed in the file scrolled list (with the subdirectories in the subdirectory scrolled list); the label that shows current

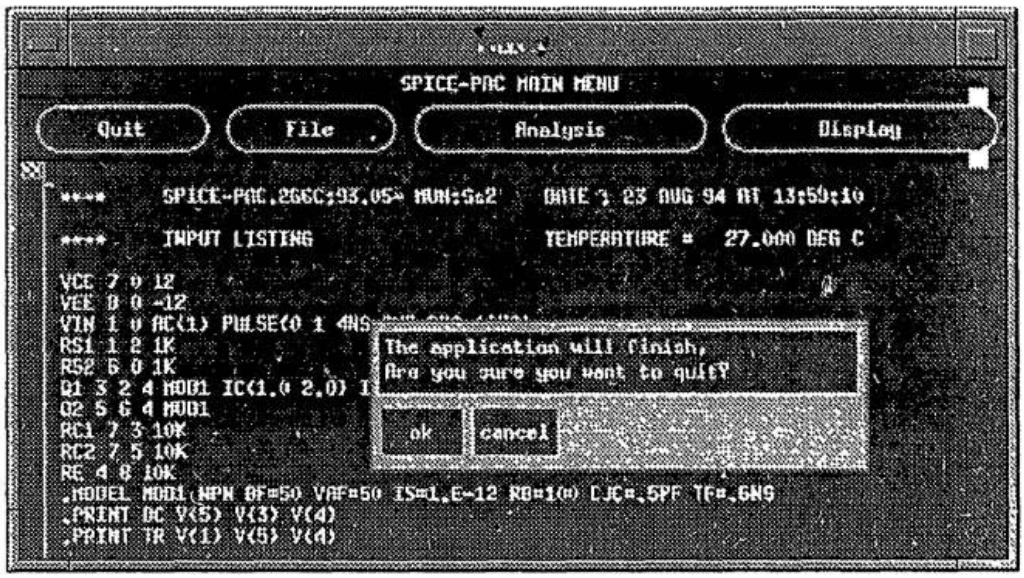

Fig.5.3. SPICE-PAC with a warning message.

directory changes to the new directory.

When a user clicks on a file name, the file name is shown in the current file label; the type of the file is also determined.

The **Home** button is provided to reset the current directory to the user's home directory.

5.2.1 File Menu

The function *Create_file* is used to build a file menu. It displays the files and subdirectories within a given directory and allows the users to view the contents of text files in that directory. Fig.5.4 shows its widget hierarchy and Fig.5.6 shows FTT with File menu popped-up.

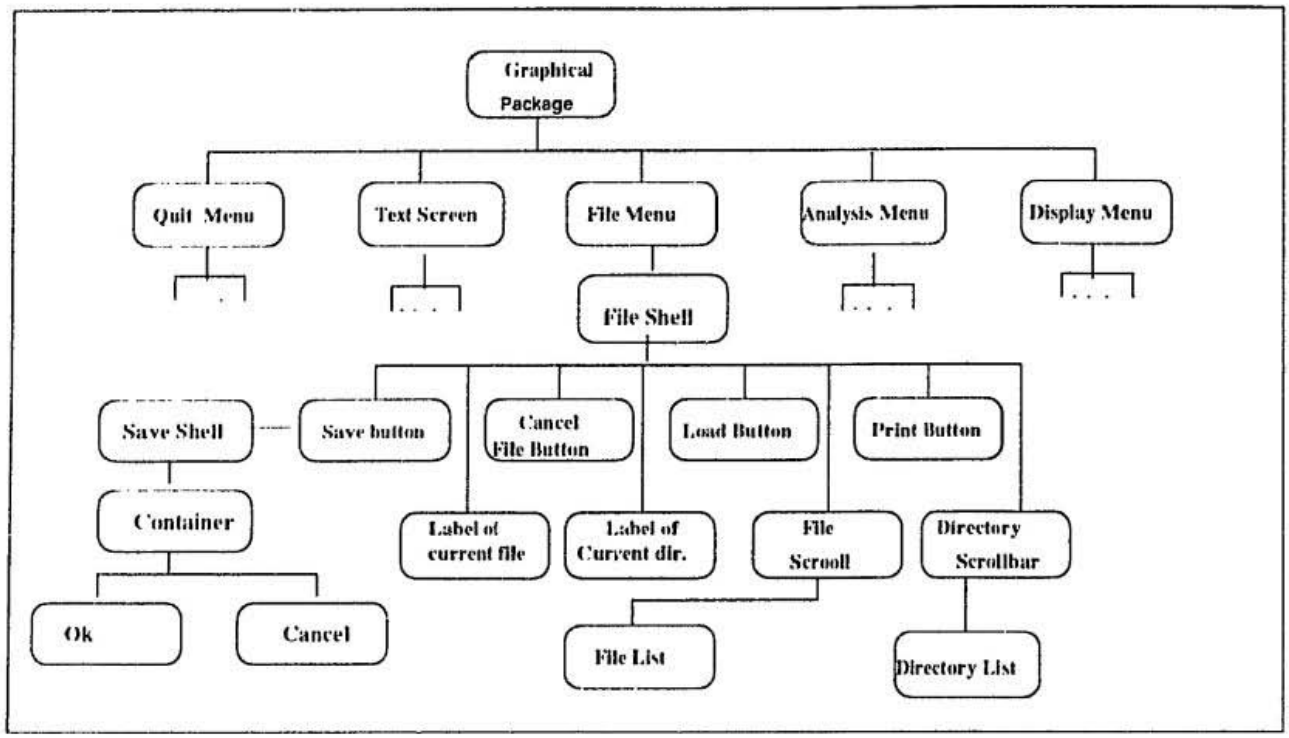

Fig.5.1. Widget hierarchy for the FILE menu.

5.2.2 Scrolled Lists

OSF/MOTIF provides a function *XmCreateScrollList* to create a scrollist. In the Athena widget set, a scrolled list can be made using *viewportWidgetClass*. A viewport is a container widget that allows its children to scrolL around. *viewportWidgetClass* is used twice in the file menu, for listing file names and for listing subdirectory names. Viewport is also used as a container to hold *textWidgetClass* to display the context of the text in the main menu.

5.2.3 Files and Directories

In addition to viewing files, lists of files and subdirectories in a given directory need to be created. The function *FillInFileNames* reads files in a given directory and adds

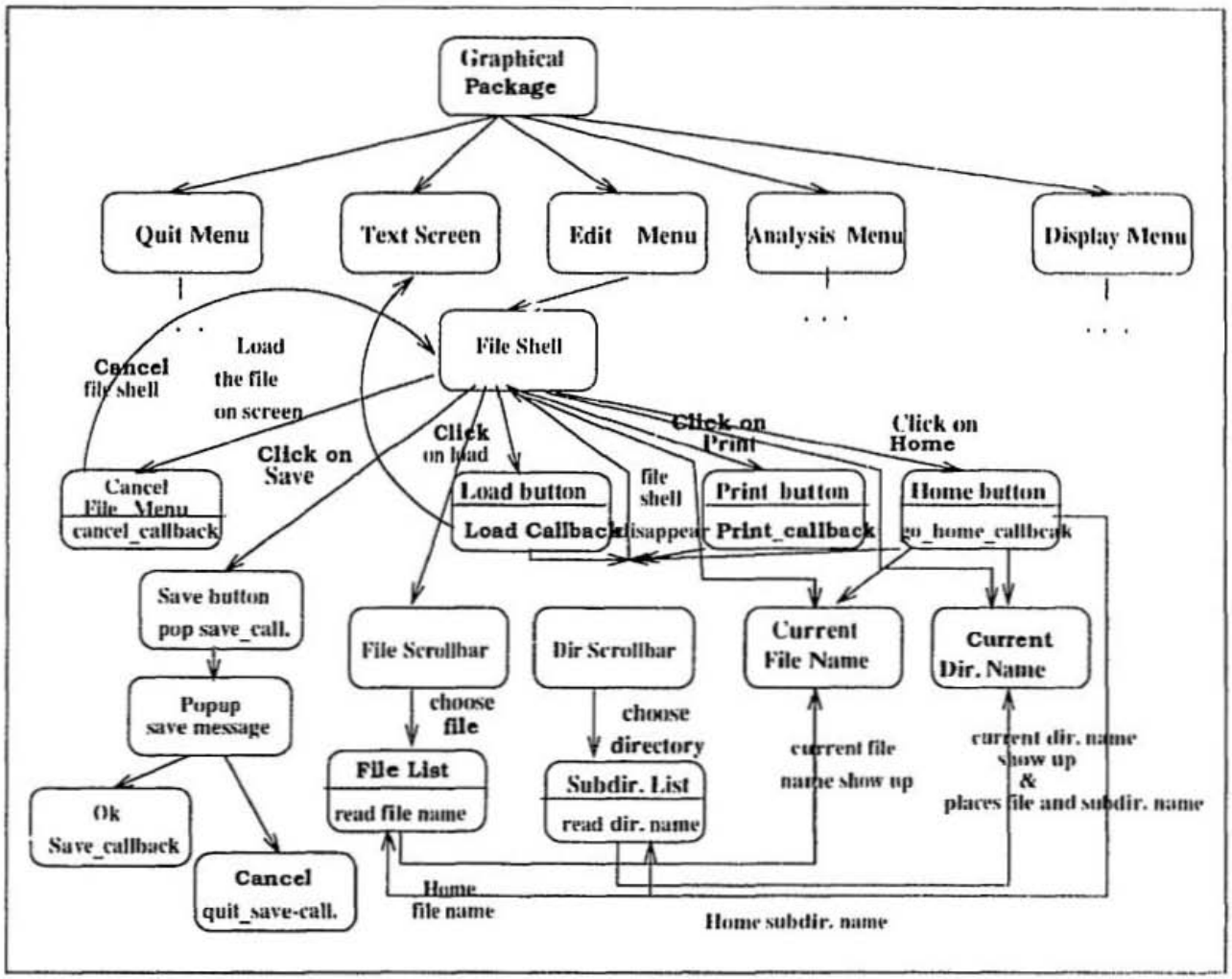

Fig.5.5. Widget interactions and callbacks for the FILE menu.

the file names and subdirectory names to these list widgets. The implementation of *FillInFileNames* is shown in Appendix A.

The function *CheckFileType* determines the type of a file as EXECUTABLE, DIRECTORY or NORMAL file, depending upon the type returned by the function *stat()*. The implementation of *CheckFileType* is shown in Appendix B.

5.2.4 Callbacks

There are six callback functions to support the FILE menu: *Quit_file*, *Save_callback*, *Load_callback*, *Print_callback*, *Selectfile_callback* and *Selectdir_callback* . Fig.5.5 shows widget interactions and callbacks for the File menu.

Quit_file: This callback function uses *XtPopdown*, similarly to other quit functions, to exit the fileshell.

Save_callback: This callback uses the function *XawAsciiSave* to find out if the text buffer has changed since the last time it was saved using *XawAsciiSave* or queried using *XawAsciiSourceChanged*. This function returns TRUE if the source has changed since the last time it was saved or was queried. An internal change flag is reset whenever the string is queried by *XtGetValues* or the buffer is saved by *XawAsciiSave*.

When the user clicks on the **Save** button, the *Save_file* callback function checks if the file has changed, a pop-up warning message shows up to let the user decide whether to save the file.

The source code of *Save_callback* is given in Appendix C.

Fig.5.7 shows the final look of FIT with file menu and the save message popped up.

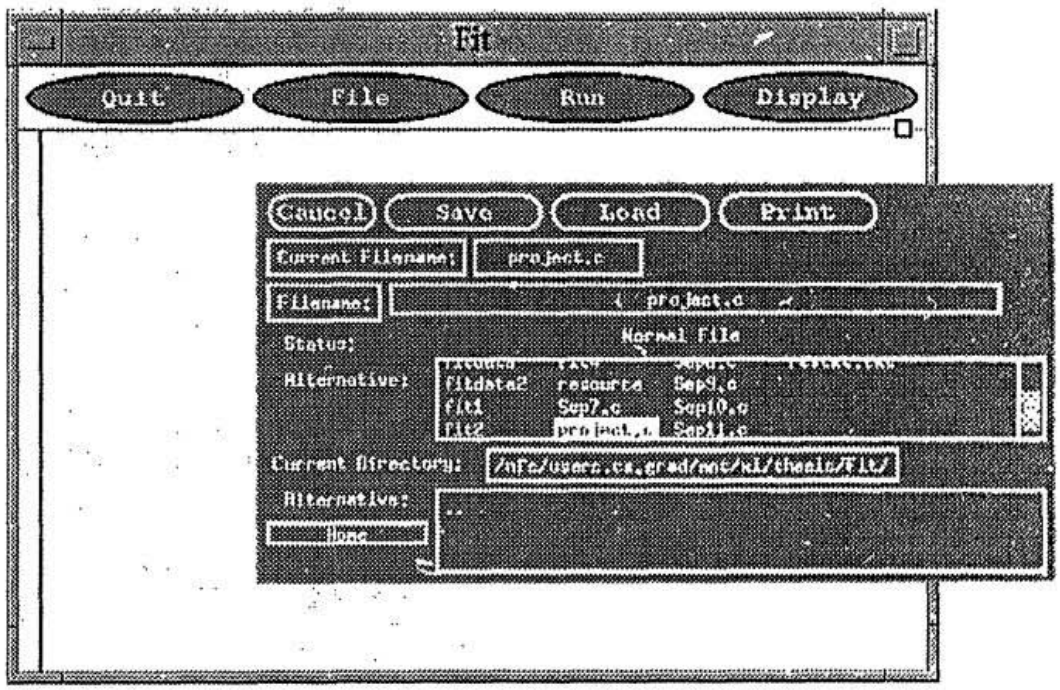

Fig.5.6. FIT with the FILE menu popped up.

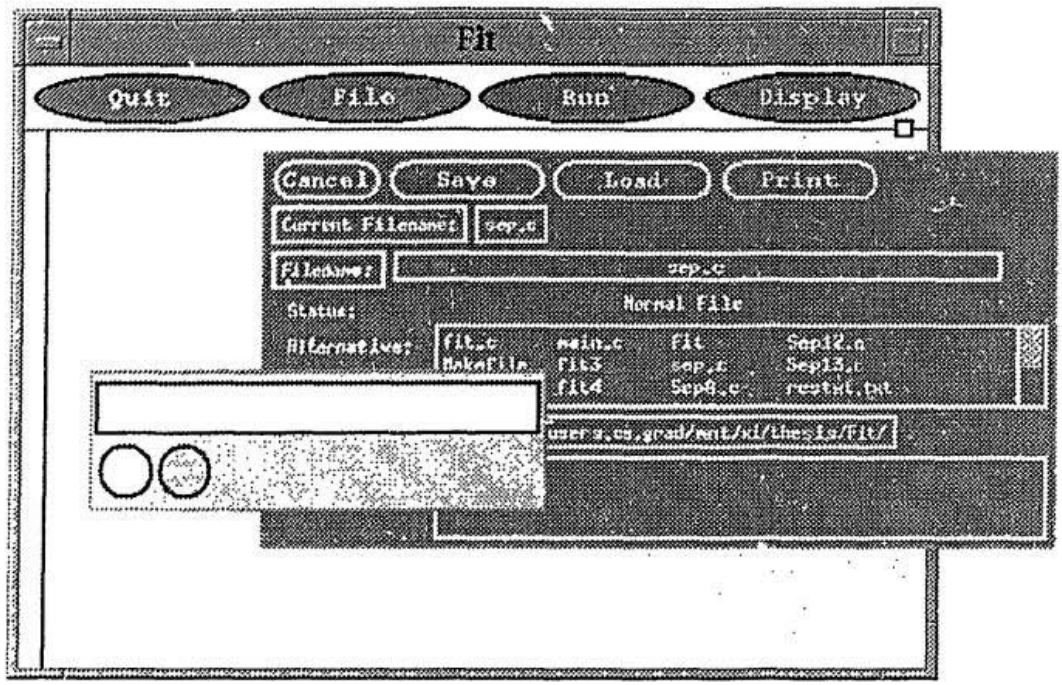

Fig.5.7. FIT with the FILE menu and a save warning message popped up.

Load_callback: When a user clicks on the **Load** button, the current file is indicated in the FileOutput area (as shown in Fig.5.8). In the function *Load_callback*, *XtSetArg* and *XtSetValues* are combined together to set the value of the FileOutput widget to indicate the current file name.

Implementation of *Load_callback* is shown in Appendix D.

Print_callback: When a user clicks on the **Print** button, the file indicated in the FileOutput area is printed out. This callback function uses the UNIX *system* call to perform printing. The file menu disappears after clicking on the **Print** button.

The implementation of *Print_callback* is shown in Appendix E.

Selectfile_callback: After selecting a file name from the File List, *Selectfile_callback* sets the current file name using *XtSetValues* and *XtSetArg*.

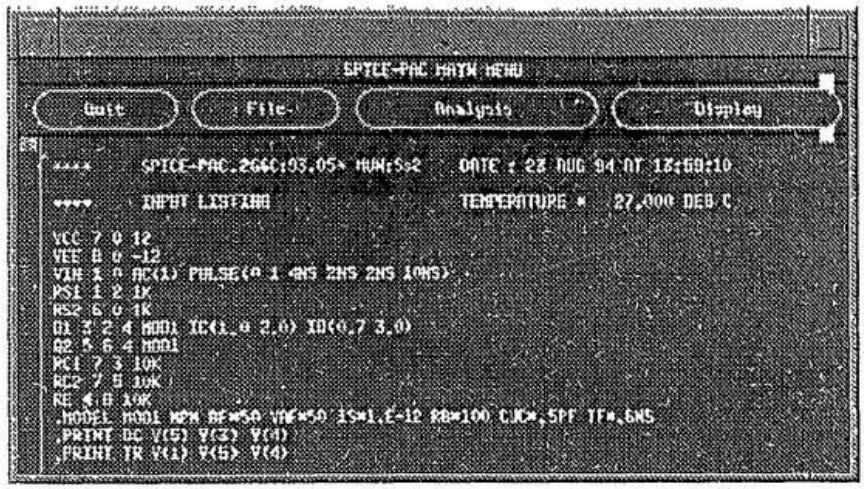

Fig.5.8. SPICE PAC with a file loaded.

Selectdir_callback: After selecting a directory name from the Directory List, the label in Current Directory as well as the contents of File List and Directory List change to reflect the new directory. This callback function uses *FillInFileNames* to fill in the two lists with file names and subdirectory names from the new directory (as shown in Appendix A).

5.3 Analysis Section

Separating the user interface from the rest of the application is a common sense approach leading to modular design. If the user interface is separated from the application program, the two parts can be modified independently and the user interface can be customized more easily, without affecting the application software.

Usually the graphical interface and software tools are linked by a GUI manager, as shown in Fig.5.9. The **Analysis** section plays the role of the GUI manager which drives and controls the application software. When a user selects the **Run** button,

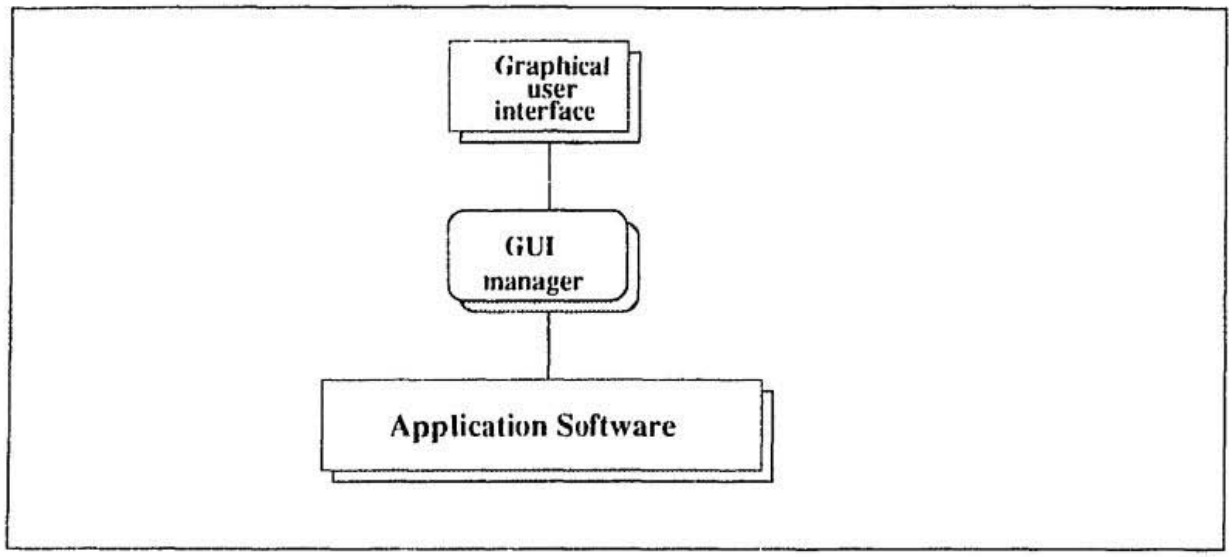

Fig.5.9. Interaction of GUI and the application software.

the tool (FIT/SPICE-PAC) is initiated. The user can then enter the commands for FIT/SPICE-PAC to execute the required analyses. FIT/SPICE data files are created automatically. Appendix N shows the implementation of the *Analysis_callback* function.

5.4 Display Section

The most important part of the graphical interface is its display section. The function *Popup_display* creates the layout of the display. It uses *formWidget* to act as the main workspace used for graphical presentation of (numerical) the results.

Graphical results of different analyses are displayed by corresponding commands, such as **AC Analysis**, **DC Analysis**, **Noise Analysis**, and so on.

In order to provide some flexibility of graphical presentation, the display section contains several functions for choosing line style, line width, line color, text color, and so on. A pop-up dialog widget to make appropriate selections is created using

DialogWidget.

The display section also provides some zooming capabilities. During viewing the graphical presentation of numerical results, users can select **zoom_in** or **zoom_out** options to make the picture smaller or larger, respectively, or they can use **X_zoom_out**, **X_zoom_in** or **Y_zoom_in**, **Y_zoom_out** to rescale the picture in one dimension.

Add_trace is another feature of the display section. If the user wants to view the results of only one output variable (rather than all of them), he can select the **Add_trace** button and type in the name of the output variable in a pop-up dialog box. If the name is incorrect for this analysis, a warning message appears on the top of the dialog box. If the name is correct, a pop-up window is created with a display of results of the selected output variable.

Compare_res feature is provided to compare the results obtained from different analyses for the same output variable. The user enters the name of the output variable in a dialog box popped up for this purpose.

5.4.1 Display Menu

The function *Popup_display* displays the results in a graphical form. It also provides the means to trace specific variables and compare the results of the same variable for different analyses. Fig.5.11 shows its widgets hierarchy and Fig.5.12 shows its widget interactions and callbacks.

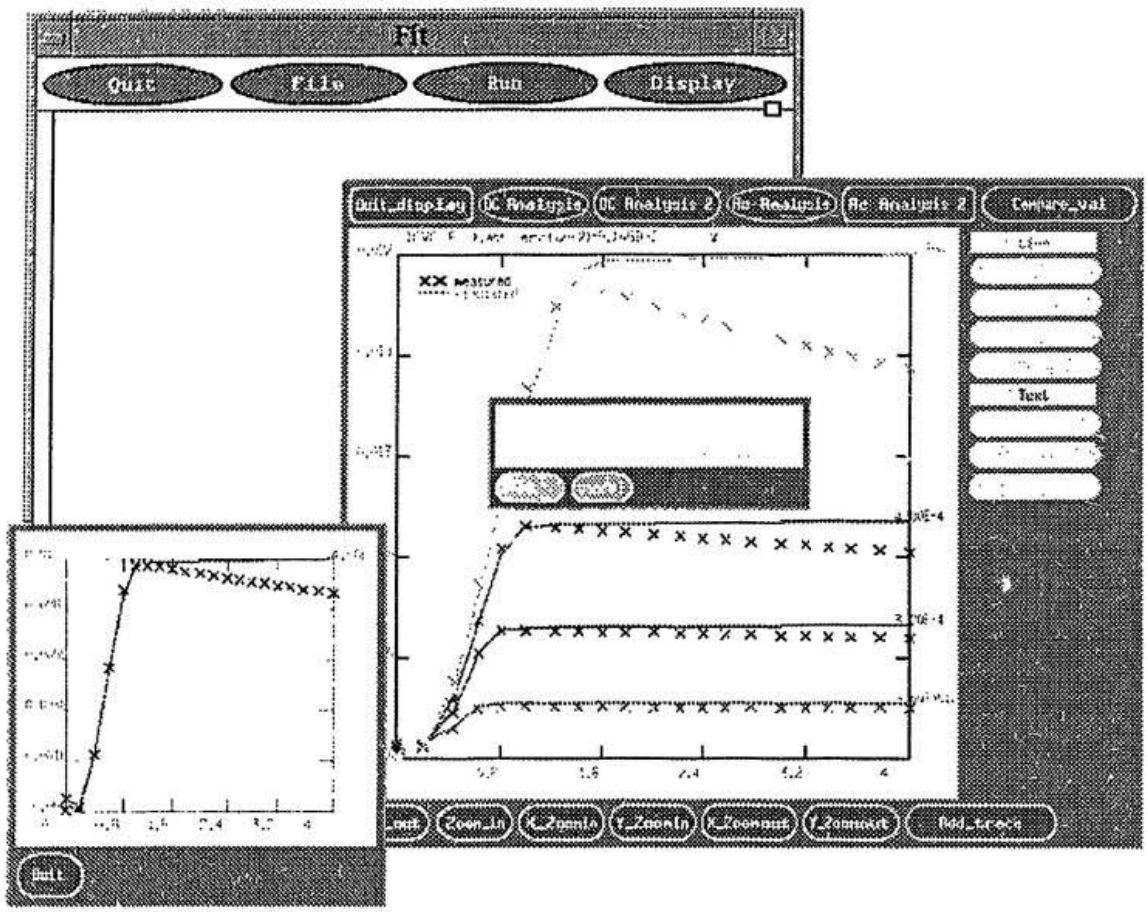

Fig.5.10. FIT with the DISPLAY menu popped up.

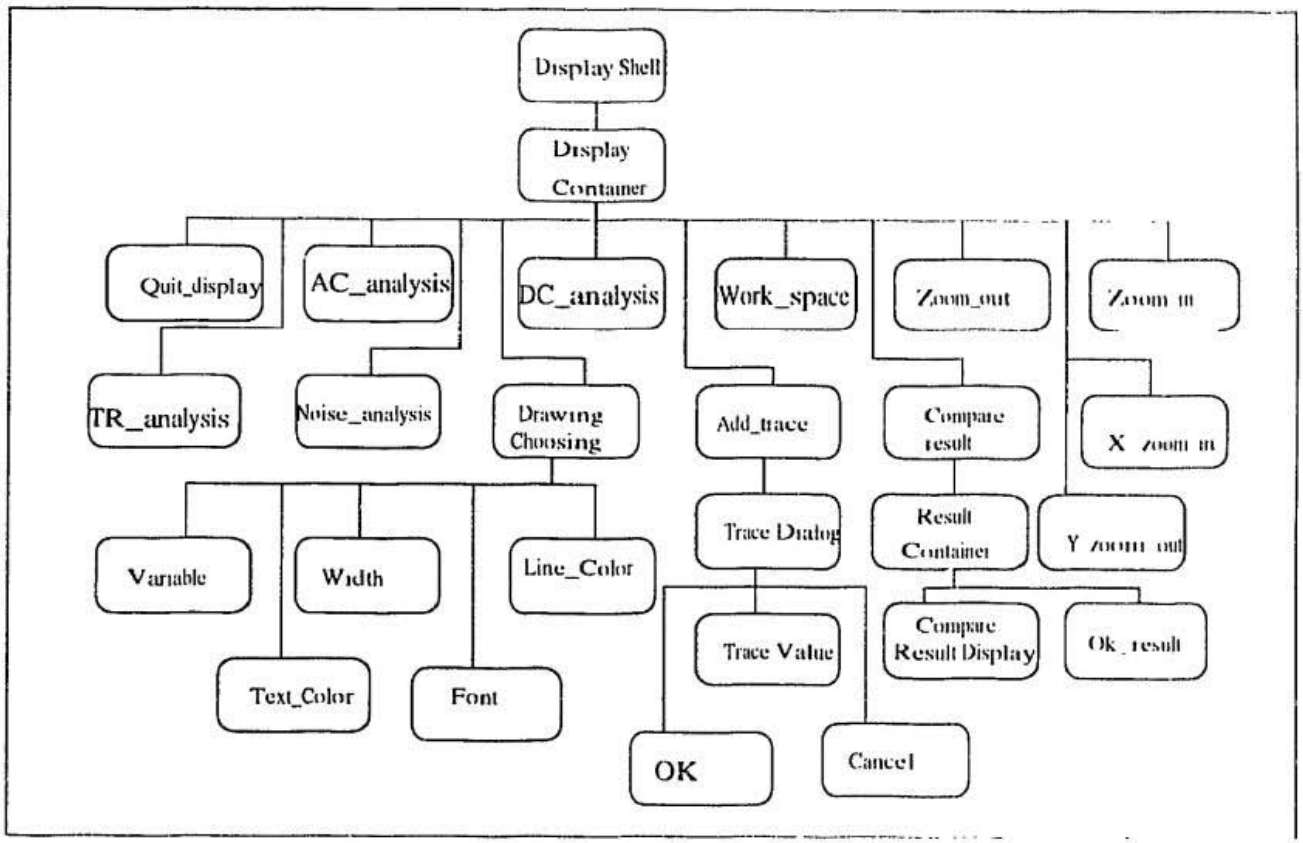

Fig.5.11. Widget hierarchy for the DISPLAY menu.

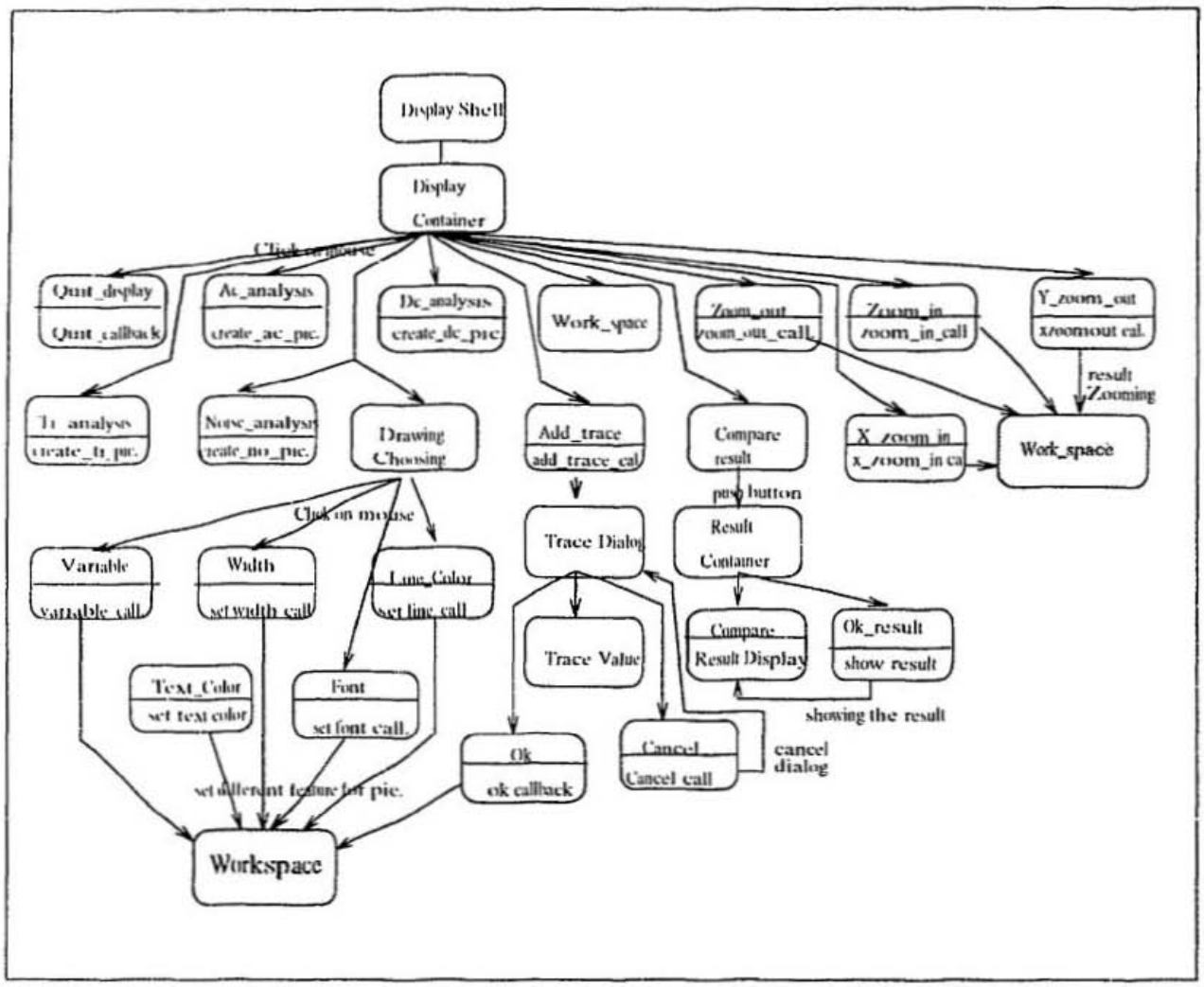

Fig.5.12. Widget interaction and callbacks for the DISPLAY menu.

5.4.2 Data Files

Data files are generated by the analysis part following a certain fixed format. Before displaying, these data files have to be analyzed. The functions *Read_fitdata* and *Read_fit3file* are used for analyzing input data files and reading them into corresponding arrays (such as the title array, the temperature array and arrays corresponding to the output variables). All those data arrays are then used for graphical presentation of results.

Implementation of *Read_fitdata* is given in Appendix F and *Read_fit3file* with its data structures in Appendix G.

5.4.3 Graphics Context

The X Window system provides a wide variety of functions to perform graphics operations. These *graphics primitives* allow the user to draw points, lines, rectangles, arcs, and so on.

Since all graphics operations require a GC (*Graphics Context* — an important graphics data structure), a GC must be created before a picture can be drawn. This GC is used to draw lines, display texts, etc. X stores the graphical information in the form of pixels that appear on the screen. This means that if another window is overlaid on top of the current one, the application program must do redrawing when the hidden area becomes exposed. A drawing exposure event-handling mechanism is used for this purpose. All the information which is presented on the screen is also stored in a pixmap. When an exposed event in the drawing area occurs, the exposed area is copied from the pixmap back to the screen. The details are shown in Appendix

H.

5.4.4 Creating Graphical Representations

Create Picture is the central part of the display section. After creating a picture, the graphical representation is displayed on the screen or saved in a bitmap. *Create Picture* is the function which reads the data and creates a graphical representation on the screen.

Create_dc_picture is one of the callback functions to create a picture for results of DC analysis. Similarly, *Create_ac_picture*, *Create_tr_picture*, *Create_no_picture* are provided for other analyses. The basic algorithm of creating a picture is the same, the differences are due to the data generated by different analyses. Implementation of *Create_dc_picture* is given in Appendix H.

5.4.5 Traces

Add_Trace is used when a user wants to display a single output variable for **DC**, **AC** or **TR** analysis. The name of the variable is entered in a dialog box, and if the name is correct, a new window is created for the graphical presentation of the selected results; if the name is incorrect, a warning message pops up.

The function *talking* creates the dialog box which allows the user to type in the variable name. The implementation of *talking* is shown in Appendix I.

The callback function *tracing* checks if the variable is correct for the current analysis. It creates a display window with a graphical result if the variable is correct, otherwise a warning message is displayed. The implement ... of *tracing* is shown in

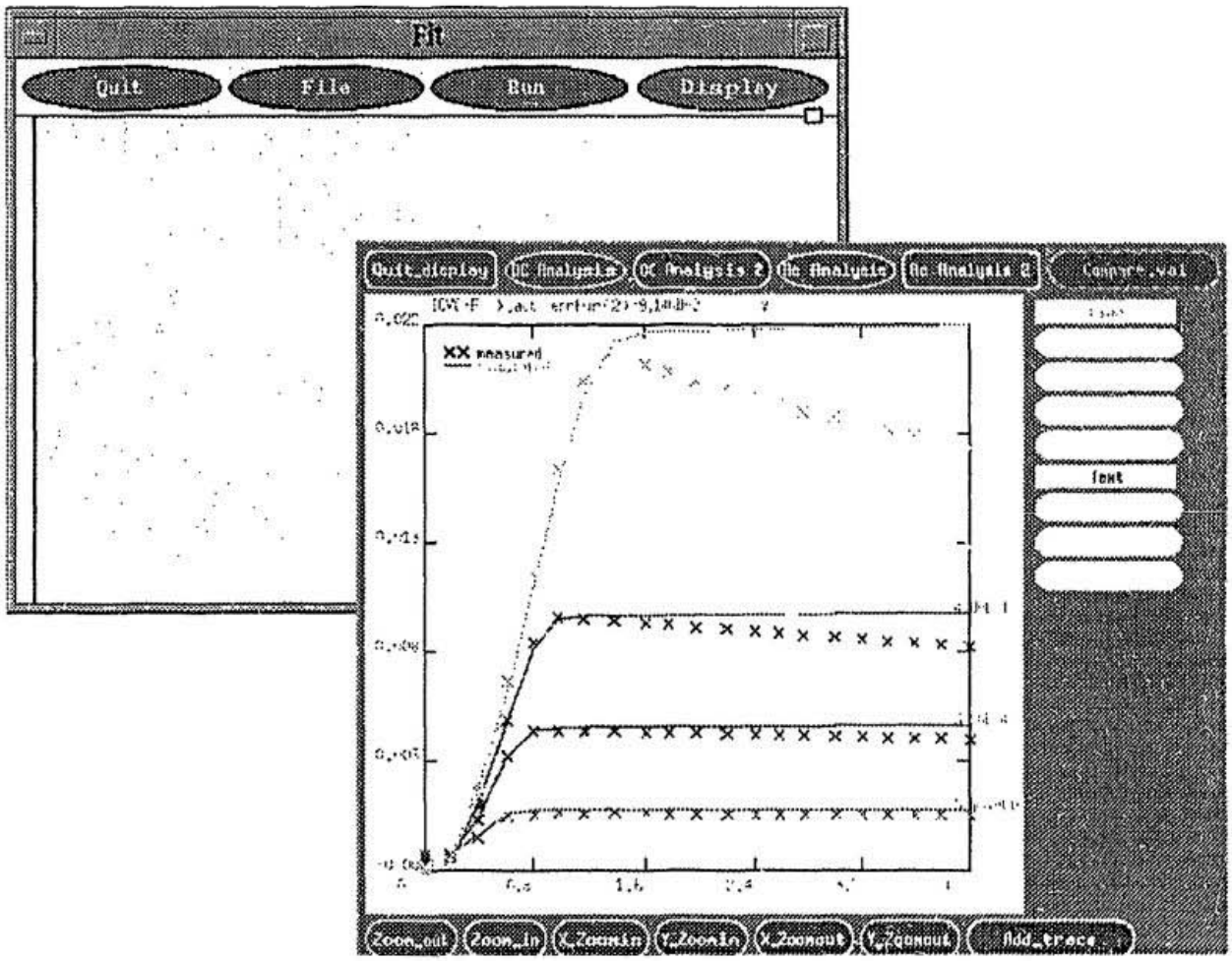

Fig.5.13. FIT with the DISPLAY menu popped up for DC analysis.

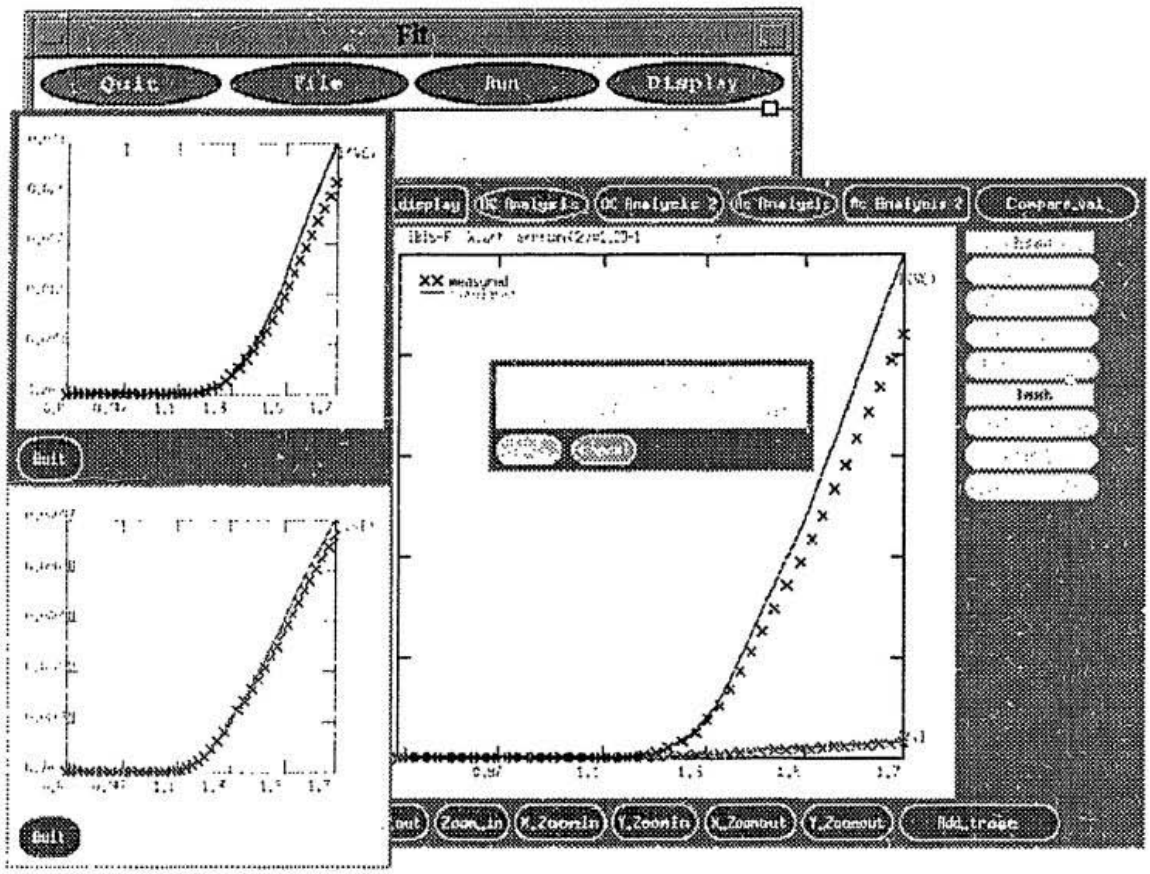

Fig.5.1-1. FIT With the DISPLAY menu and the TRACE popped up.

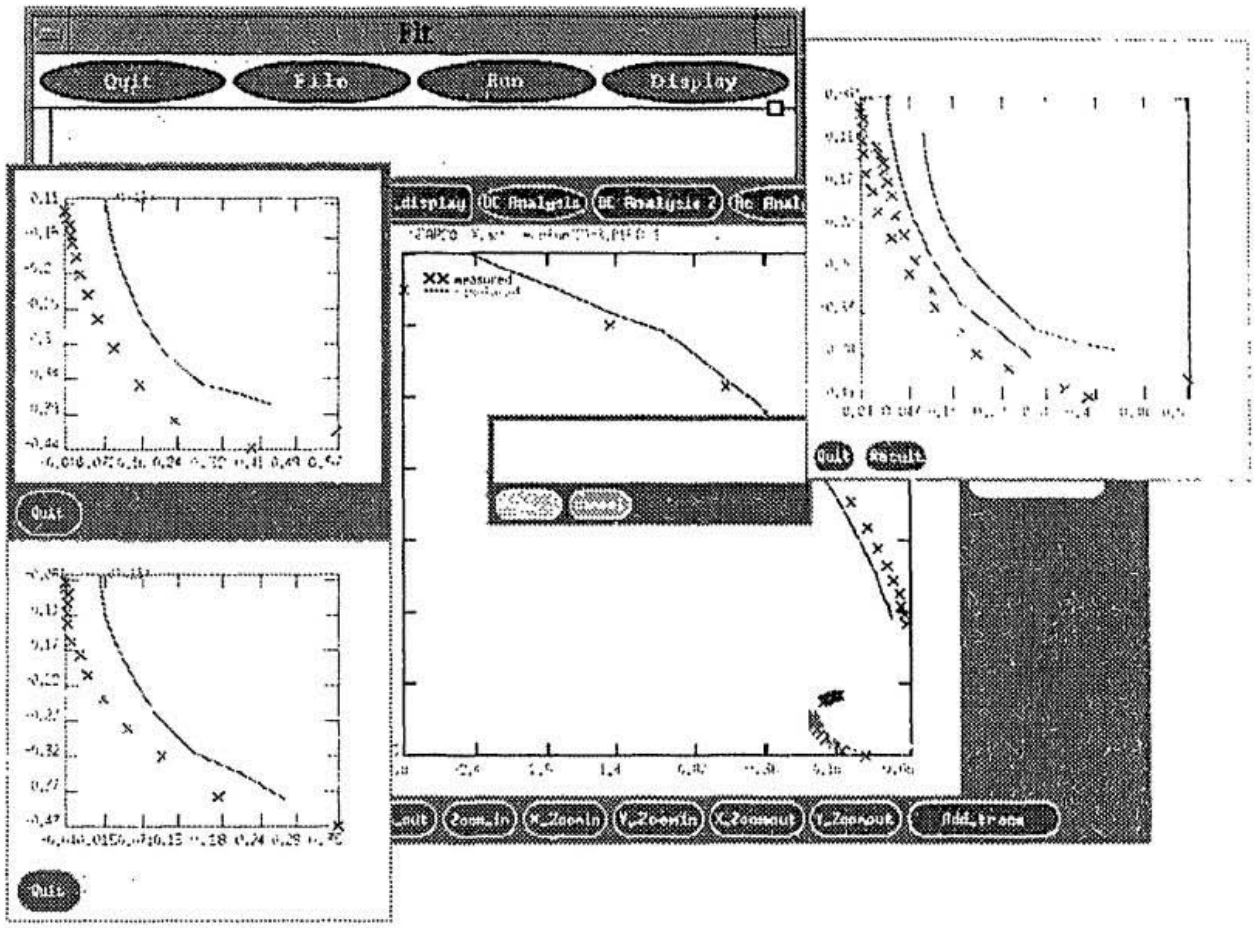

Fig.5.15. FIT with DISPLAY menu and a comparison of results popped up.

Appendix J.

The function *GetFit3Value* is used in *tracing* to check if the input variable is correct; it returns 0 if the variable is correct, otherwise it returns 1. The implementation of *GetFit3Value* is shown in Appendix K.

5.4.6 Comparisons of Results

The results of different analyses for the same output variable can be displayed in separate windows as shown in the left part of Fig.5.14. In order to compare these

results, they can also be displayed in the same window, as shown in the right part of Fig.5.15.

The function *talking_result* creates a pop up display window; it uses *formWidget* to create the main window for displaying graphical results, and uses *commandWidget* to create the **Quit** and **Result** buttons. The implementation of *talking_result* is shown in Appendix L.

The function *Result_comparing* is the callback function for **Result**. It displays the results of the same variable for different analyses. The source code of *Result_comparing* is given in Appendix M.

5.5 Compiling and Running the Program

If a user wants to recompile the package, he can use the *make* command; *Makefile* shown in Appendix O shows the necessary commands.

Before running the system, the user can prepare the default resource file ("Fit" for the FIT program and "Spice" for the SPICE-PAC program) or override the default resource file to change the layout of graphical user interface.

In order to run the program, the user can use the commands "fit" or "spice" to execute the FIT or SPICE-PAC graphical user interface, respectively.

Chapter 6

Conclusions

The main objective of the project was to design and implement a package for graphical presentation and manipulation of numerical results for SPICE-PAC and FFT type applications.

Several conclusions related to this project are as follows:

- The interface can be developed without detailed knowledge of the application software. Separating the user interface from the rest of the application is a common-sense approach to modular design; the two separated parts can be modified independently, without affecting each other.

- More capabilities such as zooming on sections indicated by a mouse should be provided to improve the flexibility of viewing capabilities.

- The graphical interface and SPICE-PAC/FFT should interact directly without using the intermediate files. Data sharing is the way to connect GUI and software applications in this case. Direct invocations of relevant procedures of the

application program would be a more efficient but also a more complicated solution.

- Similar interfaces can be developed for other CAD tools; several "standard" interface features can be identified (e.g., **File** section or **Display** section).

- The interface could be generated from "high-level" specifications provided that a specification language as well as its translator were available. Design of such a specification language and the development of its translator could be the goal of another interesting project in this area.

The methodology developed during this project can be applied to many similar applications, reducing the development time and simplifying the use of the programs.

Bibliography

[1] B.J. Keller. *A Practical Guide to X Window Programming*. The CRC Press, 1990.

[2] C.D. Peterson, *Athena Widget Set — C Language Interface*, MIT X Consortium.

[3] D.A. Young, *Window Systems Programming and Applications With Xt*, Prentice Hall, 1989.

[4] E.F. Johnson, *Power Programming ... Motif*, MIS Press, 1991.

[5] A. Nye, *The Xlib Programming Manual*, O'Reilly and Associates, 1988.

[6] OSF, *OSF/Motif Programming's Guide*, Prentice Hall, 1990.

[7] W.A. Parrette, *Motif Programming in the X Window System Environment*, McGraw-Hill, Inc. 1992.

[8] D. Heller and P.M. Ferguson, *Motif Programming Manual*, O'Reilly and Associates, 1994.

[9] S.L. Fowler and V.R. Stanwick, *The GUI Style Guide*, AP Professional, 1995.

[10] D. Flanagan, *Motif Tools*, O'Reilly and Associates, 1994.

[11] I. Sommerville. *Software Engineering*. Addison-Wesley. 1992.

[12] W.M. Zuberek. *SPICE-PAC version 2G6c — An Overview*. Technical Report #8903, Department of Computer Science, Memorial University of Newfoundland, St. John's, Canada, A1C 5S7, 1989.

[13] W.M. Zuberek, A. Konczykowska. "FIT-2, a simulation-based parameter extraction program"; Technical Report #9111; Department of Computer Science, Memorial University of Newfoundland, St.John's, Canada A1C 5S7, 1991.

Appendix A : FillInFileNames

Appendix A shows the implementation of the function *FillInFileNames* which fills in two list widgets with file names and subdirectory names from a given path.

```
function FillInFileNames(file_widget, dir_widget, current_path, max_files)
Widget file_widget;              where file names go
Widget dir_widget;               where directory names go
char current_path[ ];            name of directory
int max_files;                   max we allow in a list widget
begin
        file_count = 0;
        dir_count = 0;
        total_count = 0;
        dirp = opendir(current_path);
        if directory == NULL then
            return file_count;
        end if;
        dp = readdir directory.
        if strcmp(current_path."/") == 0 then
            dp = readdir(dirp);
        end if;
        while directory != NULL loop
            total_count = total_count + 1;
            file_type = CheckFileType(path, dp.d_name);
            if file_type == 1 then
                dir_count = dir_count + 1;
                Add file name to dir_string_list;
                i = i + 1;
            else
                file_count = file_count + 1;
                Add file name to file_string_list;
                j = j + 1 ;
            end if;
        end loop;
        Set path to currentdir_list;
          Set new dir list string to dir list
        XawListChange(alternative1_list, filelist_str, file_count, 0, TRUE);
          set new file list string to file list
        XawListChange(alternative2_list, homedir_str, dir_count, 0, TRUE);
        closedir(dirp);
end  FillInFileNames;
```

Appendix B : CheckFileType

Appendix B shows the implementation of the function *CheckFileType* which determines the type of a file as DIRECTORY, EXECUTABLE or NORMAL file.

```
function CheckFileType(current_path, filename)
char current_path[];            the name of current directory
char filename[];                the name of file
begin
        char full_name[300];
        int file_type[300];
        strcpy(current_path, full_name);
        length = strlen(current_path);
        if path[length-1] != '/' then
            strcat(full_name, "/");
        end if;
        strcat(full_name, filename);
          get information on the file
        file_type = NORMAL;
        if stat_buffer.st_mode & S_IFDIR == TRUE then
            file_type = DIRECTORY;
        elseif  stat_buffer.st_mode & S_IEXEC == TRUE  then
            file_type = EXECUTABLE;
        end if;
        return file_type;
end CheckFileType;
```

Appendix C : Save_callback

Appendix C shows the implementation of the function *Save_callback* which saves the current file.

```
static void Save_callback(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
begin
        static String save_string[ ] = {"The file already exists.",
                                        "Do you want to overwrite it?"};
        ...
        XtSetArg(args[0], XtNtextSource, &name);
        XtGetValues(fileOutput, args, 1);
        flag = XawAsciiSourceChanged(name);
        if flag = TRUE then
                Popup warning message : Are you sure to change it?
            XtPopup(saveshell, XtGrabExclusive);
        end if;
end Save_callback;
```

Appendix D : Load_callback

Appendix D shows the source code of the function *Load_callback* which loads the current file in the FileOutput area.

```
static void Load_callback(widget, closure, call_data)
Widget widget;
XtPointer closure ;
XawListReturnStruct call_data;
begin
        Arg args[1];
        XtSetArg(args[0], XtNstring, fullnames);
        XtSetValues(fileOutput, args, 1);
        XtPopdown(fileshell);
end Load_callback;
```

Appendix E : Print_callback

Appendix E shows the source code of the function *Print_callback* which prints the current file.

```
static void Print_callback(widget, closure, call_data)
Widget widget;
XtPointer closure ;
XawListReturnStruct *call_data;
begin
        strcpy(buf, "ljpr");
        strcat(buf, " ");
        strcat(buf fullnames);
        system(buf);
        XtPopdown(fileshell);
end Print_callback;
```

Appendix F : Read_fitdata

Appendix F shows the implementation of the function *Read_fitdata* which analyzes the input data file for the FIT program.

```
static void Read_fitdata(output_file)
FILE output_file;
begin
        int res, res1, res2, res3, res4;
        char buf[81];
        if (fit = fopen(fildata, "r")) == NULL then
            printf("can't open filedata "); end if;
        if (fit1 = fopen(datafile1, "w")) == NULL then
            printf("can't open datafile1"); end if;
        ...
        while not feof(fit) loop
            fgets(buf, 81, fit);   get line form fit and write it into buffer
              compare buffer with datatitle1, write into res1
            res1 = strncmp(buf, "number 1 of 4", 15);
              compare buffer with datatitle2, write into res2
            res2 = strncmp(buf, "number 2 of 4", 9);
              compare buffer with datatitle3, write into res3
            res3 = strncmp(buf, "number 3 of 4", 9);
              compare buffer with datatitle4, write into res4
            res4 = strncmp(buf, "number 4 of 4", 9);
            if res1 == 0 then
                for i in 1..30 loop
                    fgets(buf, 81, fit);
                    res = strncmp(buf, "number ", 4);
                      compare buffer with end line of filedata
                    if end == 0 then break; end if;
                    if res != 0 then
                        fputs(buf, datafile1);  put buf into fit1 data file
                    else
                        break;
                    end if;
                    ...
                end loop;
                ...
            end if;
            fclose(filedata);
            fclose(datafile1);
        end while;
end Read_fitdata;
```

Appendix G : Read_fit3file

This appendix shows the structures *Fit3_File*, *Fit3_Value* and the function *Read_fit3file* which opens the file *fit3file* and reads the data into corresponding arrays.

Fit3_File

```
struct Fit3_File
{
        char title[81];
        char temp[81];
        char name[81];
        char res[81];
        struct Fit3_Value fit3_value[100];
        int num;
} fit3_file[10];
```

Fit3_Value

```
struct Fit3_Value
{         char na[81];
        float va[100];
} fit3_value;
```

Read_fit3file

```
function Read_fit3file(fit1file)
FILE *fit1file;
begin
        char name[81], name1[81], name2[81]
        char buf[81];
        float data[200];
        if (fit3 = fopen("fit3", "r")) == NULL then
                printf("can't open fit3 ");
        else
                fgets(buf, 81, fit3);
                  put the buf into structure fit3_file[k].title
                strcpy(fit3_file[k].title, buf);
                fgets(buf, 81, fit3);
                  put the buf into structure fit3_file[k].temp
                strcpy(fit3_file[k].temp, buf);
                fgets(buf, 81, fit3);
```

```
...
  scan strings and put them into fit3_value.name
fscanf(fit3,"%s ...%s", fit3_file[k].fit3_valu e[0].na,
            ..., fit3_file[k].fit3_value[8].na);
loop
      scan the string from fit3
    fscanf(fit3,"%s",name);
      since the original data have no space between negative numbers,
      the functions fscanf and strncpy are used to separate the numbers
    len = strlen(name);
      if length is greater than 19, there should be 3 data
    j = 0;
    if len > 19 then
        data[j] = strncpy(name, 0, 8);
        j = j+1;
        data[j] = strncpy(name, 9, 10);
        j = j+1;
        data[j] = strncpy(name, 19, 10);
        j = j+1;
    else if len < 9 then
        data[j] = strncpy(name, 0, 8);
        j = j+1;
        data[j] = strncpy(name, 9, 10);
        j = j+1;
    else
        data[j] = strncpy(name, 0, 8);
        j = j+1;
    end if;
    max = j;
    k = 1;
    fit3_file[k].num = max/9;
    j = 0;
      put data into fit3_value
    for i in 1..fit3_file[k].num loop
        fit3_file[k].fit3_value[0].va[i] = data[j];
        j = j+1;
        ...
        fit3_file[k].fit3_value[8].va[i] = data[j];
        j = j+1;
    end loop;
    ...
  end loop;
 end if;
end Read_fit3file;
```

Appendix H : Create_dc_picture

Appendix H shows the implementation of the function *Create_dc_picture* which creates a representation of results for DC' analyses.

```
function Create_dc_picture(origin_x, origin_y, max_x, max_y)
Position origin_x, origin_y, max_x, max_y
begin
        float range_x, range_y
        float real_x, real_y
        Position start_x, start_y
        Position end_x, end_y
        ...
        float max_real_x, min_real_x
        float temp_x, temp_y
          initialize
        k = 0;
        min_real_x = max_real_x = fit1_file[k].fit1_value[0].va[0];
        min_real_y = max_real_y = fit1_file[k].fit1_ value[1].va[0];
          get min real x value, max real x value in file 1
        for i in 1..fit1_file[0].num loop
              if(fit1_file[k].fit1_value[0].va[i] ≤ min_real_x) then
                    min_real_x = fit1_file[0].fit1_value[0].va[i];
              end if;
              if(fit1_file[k].fit1_value[0].va[i] ≥ max_real_x) then
                    max_real_x = fit1_file[k].fit1_value[0].va[i]; }
              end if;
        end loop;
          get min real x value, max real x value in file 2
          get min real y value, max real y value in file 1
          get min real y value, max real y value in file 2
        range_x = max_real_x - min_real_x;
        range_y = max_real_y - min_real_y;
        ...
        coordinate_x = min_real_x;
        coordinate_y = min_real_y;
        rate_x = (max_x - origin_x)/range_x;
        rate_y = (origin_y - max_y)/range_y;
        temp_x = origin_x - min_real_x *rate_x;
        temp_y = origin_y + min_real_y * rate_y;
        start_x = origin_x;
        start_y = origin_y;
        end_x = max_x;
        end_y = start_y;
```

```
  clear screen and bitmap
XFillRectangle(XtDisplay(work_space), Picture,
        gc2, 0, 0, 500, 500);
XFillRectangle(XtDisplay(work_space), XtWindow(work_space),
        gc2, 0, 0, 500, 500);
  set foreground for file 1 title, Text_color[0] has default color
XSetForeground(XtDisplay(work_space), gc2, Text_colo r[0]);
  draw title at the top of window and picture bitmap
XDrawString(XtDisplay(work_space), XtWindow(work_space ), gc2,
        start_x, max_y-10, fit1_file[k].title, strlen(fit1_fil e[k].title));
XDrawString(XtDisplay(work_space), Picture, gc2, start_x,
        max_y-10, fit1_file[k].title, strlen(fit1_file[k].title) );
  draw temperature at the top of window and picture
...
  set line color and line style for frame of the coordinate
XSetForeground(XtDisplay(work_space), gc2, Line_color[ 0]);
XSetLineAttributes(XtDisplay(work_space), gc2,
        Line_width[0], Line_style[0], NULL, NULL);
XDrawRectangle(XtDisplay(work_space), XtWindow(work_spa ce),
        gc2, origin_x, max_y, max_x - origin_x, origin_y - max_y);
XSetForeground(XtDisplay(work_space), gc2, Text_color[ 3]);
  draw cross for lable of measured data
DrawCross(origin_x+20, max_y+20);
DrawCross(origin_x+30, max_y+20);
XDrawString(XtDisplay(work_space), XtWindow(work_spa ce), gc2,
        origin_x+40, max_y+25, "measured", strlen("measured"));
  draw line for lable of simulated data in middle of result picture
XDrawLine(XtDisplay(work_space), XtWindow(work_space),
        gc2, origin_x+16, max_y+30, origin_x+36, max_y+30);
XDrawString(XtDisplay(work_space), XtWindow(work_space) ,
        gc2, origin_x+ 40, max_y+35, "simulated", strlen("simulated"));
  draw grad for the result picture
for i in 1 .. fit1_file[k].num-1 loop
    XSetForeground(XtDisplay(work_space), gc2, Line_color[0]);
    XDrawLine(XtDisplay(work_space), XtWindow(work_space, gc2,
            start_x, , start_y, start_x+10, end_y);
    . .
    XDrawLine(XtDisplay(work_space), Picture, gc2, start_x,
            start_y, end_x-10, end_y);
      give label of coordinate
    XSetForeground(XtDisplay(work_space), gc2, Text_color[2]);
            sprintf(scratch, " %.2g", coordinate_x)
    XDrawString(XtDisplay(work_space), XtWindow(work_space),
            gc2, start_x-20, start_y+15, scratch, strlen(scratch));
      set new coordinate data
```

```
            start_x = start_x+distance_x;
            start_y = start_y;
            end_x = start_x;
            end_y = end_y;
            coordinate_x = coordinate_x + cor_dist_x;
        end loop;
        ...
          set line attribute for first simulated data
        XSetLineAttributes(XtDisplay(work_space), gc2,
                    Line_width[1], Line_style[1], NULL, NULL);
          draw first measure data using cross figure
        for i in 1 .. fit1_file[k].num-1 loop
            XSetForeground(XtDisplay(work_space), gc2, Line_color[1]);
            x = temp_x + rate_x*fit1_file[k].fit1_value[0].va[i];
            y = temp_y - rate_y*fit1_file[k].fit1_value[1].va[i];
            DrawCross(x, y);
        end loop;
          draw second measure data using cross figure
        ...
          draw first simulated data
        for i in 1 .. fit1_file[k].num-1 loop
            XSetForeground(XtDisplay(work_space), gc2, Line_color[1]);
              calculate the x, y value in the window
            x = temp_x + rate_x*fit1_file[k].fit1_value[0].va[i];
            y = temp_y - rate_y*fit1_file[k].fit1_value[1].va[i];
            XDrawPoint(XtDisplay(work_space), XtWindow(work_space),
                        gc2, x, y);
            x1 = temp_x + rate_x*fit1_file[k].fit1_value[0].va[i+1];
            y1 = temp_y - rate_y*fit1_file[k].fit1_value[1].va[i+1];
            XDrawLine(XtDisplay(work_space), XtWindow(work_space),
                        gc2, x, y, x1, y1);
        end loop;
          draw the second simulated data array
          draw the second simulated data value
        XDrawString(XtDisplay(work_space), XtWindow(work_space), gc2
        x+5, fit1_file[k].fit1_value[2].na, strlen(fit1_file[k].fit1_value[2].na));
end Create_dc_picture;
```

Appendix I : talking

Appendix I shows the source code of *talking*, the dialog handling function.

```
function talking(button, client_data, call_data)
Widget button;
XtPointer client_data, call_data;
begin
        n = 0;
        trace_dialog = XtCreatePopupShell("TraceDialog",
                transientShellWidgetClass, button, args, n);
        n = 0;
        container = XtCreateManagedWidget("Container", formWidgetClass,
                trace_dialog, args, n);
        n = 0;
        talk = XtCreateManagedWidget("dialog", dialogWidgetClass,
                container, args, n);
        n = 0;
        XtSetArg(args[n], XtNfromVert, talk);
        n++;
        ok = XtCreateManagedWidget("ok", commandWidgetClass,
                container, args,n);
        XtAddCallback(ok, XtNcallback, tracing, (XtPointer) talk);
        n = 0;
        XtSetArg(args[n], XtNfromVert, talk);
        n++;
        XtSetArg(args[n], XtNfromHoriz, ok);
        n++;
        cancel = XtCreateManagedWidget("cancel", commandWidgetClass,
                container, args, n);
        XtAddCallback(cancel, XtNcallback, cancel_trace_dialog, NULL);
        XtPopup(trace_dialog, XtGrabNone);
end talking;
```

Appendix J : tracing

Appendix J shows the implementation of the function *tracing* which checks if the output variable is correct for the current analysis. It creates a display window with graphical result if the variable is correct, otherwise it creates a warning message.

```
static void tracing(widget,closure,callData)
Widget widget;
XtPointer closure, callData;
begin
        char buf[81], shell_name[81], container_name[81];
        char quit_name[81], space_name[81];
        int flag;
        Widget talk = (Widget) closure;
        String vname = XawDialogGetValueString(talk);
        ...
          check the value form different analyst mode
        case (Display_Id) is
            when 0 do
                flag = GetFit1Value(vname);
            when 1 do
                flag = GetFit2Value(vname);
            when 2 do
                flag =GetFit3Value(vname);
            when 3 do
                flag = GetFit4Value(vname);
            default:
                flag = GetFit3Value(vname);
        end case;
          if it is a correct value, popup window showing the result
        if flag == 0 then
            Trace_Var = Trace_Var + 1;
            i = Trace_Var;
            Trace_Id = i;
            XtSetArg(args[0], XtNsensitive, TRUE);
            XtSetValues(Compare_result, args, 1);
              get trace value and display Id
            sprintf(shell_name, "TraceShell%d", i);
            sprintf(container_name, "TraceContainer%d", i);
            sprintf(space_name,"Trace_space%d", i);
            sprintf(quit_name, "Quit_trace%d", i);
            sprintf(quit_callback_name, "quit_tracing%d", i);
              create new display window which includes quit button
            n = 0;
```

```
                traceshell[i] = XtCreatePoupShell(shell_name,
                        transientShellWidgetClass, widget, args, n);
                n = 0;
                traceContainer[i] = XtCreateManagedWidget(container_name,
                        formWidgetClass, traceshell[i], args, n);
                trace_space[i] = XtCreateManagedWidget(space_name,
                        formWidgetClass, traceContainer[i], args, n);
                  if visibility changes, then redraw the picture
                XtAddEventHandler(trace_space[i], VisibilityChangeMask, FALSE,
                        trace_exposing, NULL);
                  create quit button, under the display window
                XtSetArg(args[n], XtNfromVert, trace_space[i]);
                n++;
                quit_trace[i] = XtCreateManagedWidget(quit_name,
                        commandWidgetClass,traceContainer[i], args, n);
                XtAddCallback(quit_trace[i], XtNcallback, quit_tracing, traceshell[i]),
                XtPopup(traceshell[i], XtGrabNone);
        else
                - warning message shows up on top of the dialog
                printf("Can't get the value");
                sprintf(str,"Can't get the value Try again.", vname);
                XtSetArg(args[0], XtNlabel, str);
                XSetArg(args[1], XtNvalue, "");
                XtSetValues(talk, args, TWO);
        end if;
end tracing;
```

Appendix K : GetFit3Value

Appendix K shows the implementation of the function *GetFit3Value* which checks the input variable and returns 0 if the variable is correct, otherwise returns 1.

```
function GetFit3Value(value_name)
char value_name[50];
int k, res, res1, res2, res3, res4;
int i;
int judge;
begin
        judge = -1;
        k = 0;
        res = strcmp(value_name, fit3_file[k].fit3_value[2].na);
        res1 = strcmp(value_name, fit3_file[k].fit3_value[4].na);
        res2 = strcmp(value_name, fit3_file[k].fit3_value[6].na);
        res3 = strcmp(value_name, fit3_file[k].fit3_value[8].na);
        if res == 0 then
            Fit3_Value_No = 2,
            judge = 0;
        end if;
        if res1 == 0 then
            Fit3_Value_No = 4;
            judge = 0;
        end if;
        if res2 == 0 then
            Fit3_Value_No = 6;
            judge = 0;
        end if;
        if res3 == 0 then
            Fit3_Value_No = 8;
            judge = 0;
        end if,
        return judge;
end GetFit3Value;
```

Appendix L : talking_result

Appendix L shows the source code of the function *talking_result* which handles the dialog for displaying results.

```
static void talking_result(widget,closure,callData)
Widget widget;
XtPointer closure, callData;
begin
        i = 1;
        Result_Id = i;
        sprintf(shell_name, "ResultShell%d", i);
        sprintf(container_name, "ResultContainer%d", i);
        sprintf(space_name,Result_space%d", i);
        sprintf(quit_name, "Quit_result%d", i);
        n = 0;
        resultshell[i] = XtCreatePopupShell(shell_name,
                transientShellWidgetClass, widget, args, n);
        n = 0;
        resultContainer[i] = XtCreateManagedWidget(container_name,
                formWidgetClass, resultshell[i], args, n);
        n = 0;
        label = XtCreateManagedWidget("resultlabel", labelWidgetClass,
                resultContainer[i], args, n);
        n = 0;
        XtSetArg(args[n], XtNfromVert, label);
        result_space[i] = XtCreateManagedWidget(space_name,formWidgetClass,
                resultContainer[i], args, n);
        XtSetArg(args[0], XtNfromVert, result_space[i]);
        n = 1;
        quit_result[i] = XtCreateManagedWidget(quit_name,
                commandWidgetClass, resultContainer[i], args, n);
        XtAddCallback(quit_result[i], XtNcallback, quit_callback, resultshell[i]);
        XtSetArg(args[0], XtNfromVert, result_space[i]);
        XtSetArg(args[1], XtNfromHoriz, quit_result[i]);
        n = 2;
        result = XtCreateManagedWidget("Result",commandWidgetClass,
                resultContainer[i], args, n);
        XtAddCallback(result,XtNcallback,result_comparing, NULL);
        XtPopup(resultshell[i], XtGrabNone);
end talking_result;
```

Appendix M : Result_comparing

Appendix M shows the source code of the function *Result_comparing* which displays the results of the same output variable for different analyses.

```
static void Result_comparing(widget,closure,callData)
Widget widget;
XtPointer closure;
XEvent *callData;
begin
        i = 1;
        gc2 = XtGetGC(result_space[i], 0, 0);
        trace_display = XtDisplay(result_space[i]);
        trace_window = XtWindow(result_space[i]);
        Compare_ac_value(40, 280, 300, 40);
end Result_comparing;
```

Appendix N : Analysis_callback

Appendix N shows the source code for the function *Analysis_callback* which links the graphical interface with an application program.

```
static void Analysis_callback(w, client_data, call_data)
Widget w;
XtPointer client_data, call_data;
begin
        char run[30];
        strcpy(run, "sppac");
        system(run);
end Analysis_callback;
```

Appendix O : Makefile

This appendix shows the makefile for FIT.

```
CC= cc -O
LIBS = -L/usr/X11R5/lib -lXaw -lXext -lXmu -lXt -lX11 -lm
fit:            fit.c
                $(CC) -o Fit fit.c $(LIBS)
```