1994

Distributed event monitor user interface tool

Keyang Huang
Lehigh University

AUTHOR:

Huang, Keyang

TITLE:

Distributed Event Monitor User Interface Tool

DATE: October 9, 1994

Distributed Event Monitor User Interface Tool

by

Keyang Huang

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Computer Science

in

Department of Electrical Engineering and Computer Science

Lehigh University

July 6, 1994

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Computer Science.

8-25-94
Date

Thesis Advisor

Chairperson of Department

Acknowledgement

This Distributed Event Monitor User Interface Tool development took me a couple months to learn, write and compile. More thanks go to my advisor Dr. Spezialetti, for her instructions, helps, discussions and paper writing in this project. I learned more from her about distributed computation and other things that exceed what I ever expected. Added thanks to those who worked together as team with me on this project include Scott Bernberg, Ali Yildirim, Ali Erkan, and Max Brogi. We discuss and learn from each other, we share the great ideas. Especially Scott Bernberg supports me the I/O interface requirements and event script definitions. Also many thanks to Brain Hearn, Sr. system programmer of computer center and Binod Taterway, Sr. user consultant of computer center for their help in troubleshooting of my program debugging. I learned a lot of dbx in Motif, X window, and X Toolkit Intrinsics from them.

Table of Contents

Acknowledgements iii

Table of Contents iv

List of Figures vii

Abstract 1

1. Introduction 2

2. Event-Driven Environment and Object-Oriented Nature of the Program 5

3. Style Guide of the Program 9

4. Three Basic Concepts: Resources, Callbacks, and Managers 10

5. Shell Widget, Application Context and Class, Command-line Arguments, and Fallback Resources 14

6. Functionality, Menus, and Menubars 16

7. Creating of Main Window, Build Menu, Menubars, Pulldown Menu, and Pullright Menu 17

1) Create Main Window 17

2) Steps to Create Menu and Menubars 18

3) Techniques to Build General Menu 19

4) Write a Generalized Function 22

8. Compound String, C String and Their Conversion 26

9. Text Widget and Text Editor 29

10. File Menu Creation and All Dialog Boxes 30

11. Edit and Search Menu 34

12. General Approach and Techniques for Generalizing Event Definitions 37

1) Form Dialog Box and Shell 37

2) Form Widget and Resizing 38

3) RowColumn Widget and Push Buttons Managers 39

4) Radio Boxes and Options 41

5) List Widget, Event Adding and Deleting 42

13. Event Definition Menus 44

1) General Structure of the Event Definition Form 44

2) General Steps for Input Event Definition 45

14. Command Widget and Run, Compile, Parser Menus 47

15. Output Menu 48

16. User's Manual 50

1) What Is Needed to Run This Tool 50

2) Running This Tool 50

3) Quitting This Tool 50

4) Defining Event Script File 50

a) Primitive Event Definition 51

b) Highlevel Event Definition 53

c) Protocol Event Definition 60

d) Goal Event Definition 63

e) Local Event Definition 65

f) Assign Event Definition 67

g) Place Event Definition 69

5) Checking User Defined Event and Event Script File 72

6) Modifying User Defined Event and Event Script File 74

17. References 78

18. Vita 79

List of Figures

Figure 1. The Distributed Event Monitor User Interface Tool (DEMUIT) 5

Figure 2. The Hierarchy of the Motif User-Interface Library Model 8

Figure 3. File Selection Dialog For Open File 33

Figure 4. The Output of Opening File menu.c (DEMUIT source file) 34

Figure 5. The List Widget Popup Form 43

Figure 6. The Primitive Event Definition Form 51

Figure 7. The Completed "primitive E1 is access x main" Definition 53

Figure 8. The Highlevel Event Definition Form 54

Figure 9. The Completed "highlevel E7 is E1 or E2" Definition 56

Figure 10. The Completed "highlevel E4 is E6 notbtwn E9 E5" Definition 58

Figure 11. The Completed "highlevel E1 is E2 and E3 then E4" definition 60

Figure 12. The Protocol Event Definition Form 61

Figure 13. The Completed "protocol all is lc" Definition 62

Figure 14. The Goal Event Definition Form 63

Figure 15. The Completed "goal E7" Definition 64

Figure 16. The Local Event Definition From 65

Figure 17. The Completed "local E7" Definition 66

Figure 18. The Assign Event Definition 67

Figure 19. The Completed "assign all to em pl118.cc.lehigh.edu 1" Definition 69

Figure 20. The Place Event Definition Form 70

Figure 21. The Completed "place all at myprog pl118.cc.lehigh.edu 1" Definition 71

Figure 22. The Event Checking Form 72

Abstract

The Distributed Event Monitor User Interface Tool (DEMUIT) was developed to aid in developing and specifying both events and monitoring configurations for the "Event Recognition Testbed (EVEREST)" project. DEMUIT was written in C programming language and Motif graphical user interface (GUI). This paper first describes the principles and requirements of designing DEMUIT. The methodology, the concepts, the functionalities, the design process and approaches, the technical skills are then discussed in details. DEMUIT is an event-driven environment. The main menu consists of Edit, Search, Event, Monitor, Display, Run, Output, and Help menubars. Each menubar has pulldown menus associated with it. The user can specify seven events from a popup event definition form: primitive event, highlevel event, protocol event, assign event, local event, goal event, and place event. The user's manual is also attached.

1. Introduction

In distributed computing systems there are two main characteristics: no single, shared memory and no single, global time. These characteristics make monitoring and analyzing the system's behavior a complex and potentially costly task. The Event Recognition Testbed (EVEREST) [1] was designed to study the overhead and applicatbility of various strategies for monitoring distributed computations for the occurrences of distributed events. The system allows users to specify events, which are activities of interest regarding the execution of a distributed computation. An event may test the behaviors of one or more process in the distributed computation. In addition, the user can specify the allocation of event monitoring and evaluation responsibilities to various monitoring modules. In this way, different monitoring configurations can be tested and compared. As the computation executes, EVEREST captures information pertaining to the defined events, organizes and analyzes the data, and alerts the user when an occurrence of any of the specified events is detected. As part of the EVEREST system, a user interface was developed to aid developing and specifying both the events and monitoring configurations.

The principles to design this user interface are to satisfy as many functionalities as possible, to keep the interface as simple as possible, while not restricting the functionality to accommodate simplicity, to support many ways for user to define, modify, and edit the defined events file and to help the user to be as productive as possible.

The C programming language and Motif graphical user interface (GUI) were used to develop this user interface tool. Motif is an event driven programming environment. Its important property is called "what you see is what you get". It provides a complete set of widgets for such common user interface objects (appearance) as: the menus, the labels, the push buttons, the dialog boxes, the scrollbars, text entry, and the display areas (*the look*). The user interacts with these objects by typing at the keyboard, and by clicking, selecting, and dragging various graphic elements of the objects with mouse (*the feel*). The Motif window manager, mwm, which does all of the widget management internally, helps to enforce this *look-and-feel* style guide. To develop the user interface using Motif is to decide which widgets will be used to create the user interface that is desired. Motif functions position the widgets on the screen in the right location and at the right size. At the same time, Motif uses functions known as callback routines that notify the program when a user adjusts one of the widgets. Thus the primary responsibility of the application in program is to insure that the application responds correctly to the user's actions.

The Distributed Event Monitor User Interface Tool (DEMUIT) is designed to meet these requirements, and is built upon the user's perspectives. The programming methodology, the styles, the concepts, the task, the functionality, the design process, the technical skills, and usages are also presented. The DEMUIT was developed on UNIX workstation RISC 6000 using the C programming language, Motif graphical user interface (GUI) which uses X as the window system and the *X Toolkit Intrinsics (Xt)* as the platform. All Motif programs (run on Motif 1.2) need to link in at least three libraries: *Motif Xm library*, *intrinsics' Xt library*, and X version 11 release 5 (X11 R5) which consists of the X server, X protocol, and Xlib. **Xt** provides a higher-level set of functions for creating and setting resources on the widgets that make programming easier. **Widget** is the user interface object that the user sees in a picture-on-screen form, and the programmer sees in a set of the resources and callbacks form. **Xm** provides the widgets plus an array of utility and convenience functions for creating groups of widgets that are used collectively as a single type of user-interface element. **X server** is the process executing in your workstation and managing the graphics output and the inputs from the keyboard and mouse. The X graphical window system provides the user with multiple windows run by multiple applications. **X protocol** defines the meaning of the data exchanged between the client and the server. **Xlib** is the library of C routines that programmers use to access the server. The combination of Motif and X allow programmer to create graphical interface applications capable of running on practically any vendor's workstation[2,4,6,7].

Although Motif presents a de facto standard interface which is in use across many platforms, the user should be aware of some trade-offs: There is considerable overhead incurred per program, Xt-based programs, including Motif, allocate more than 50K of RAM at application start time, and tend to toward one megabyte minimum executable size. Motif programs tend to be slow in the beginning of the execution[5].

The DEMUIT has a menu bar with pulldown menu. The menu bar contains **text editor**, **event scripts definition**, **display**, **run**, **output** and **help** menus as shown below in Figure 1.

The text editor includes **File**, **Edit** and **Search**. The **event** scripts definition menu consists of seven events: **primitive** event, **highlevel** event, **protocol** event, **assign** event, **local** event, **goal** event, and **place** event. The **Display** menu can show time, clock, and the order of events in a distributed system graphically. The **Run** menu has **compile**, **run**, and **parser** mode.
The **Output** menu will show all events user defined. The **Help** menu provides help information dialog. The user can also push the help button to get the on-line help on user specified event definition input popup form. By using this tool user can complete most of the processes such as:

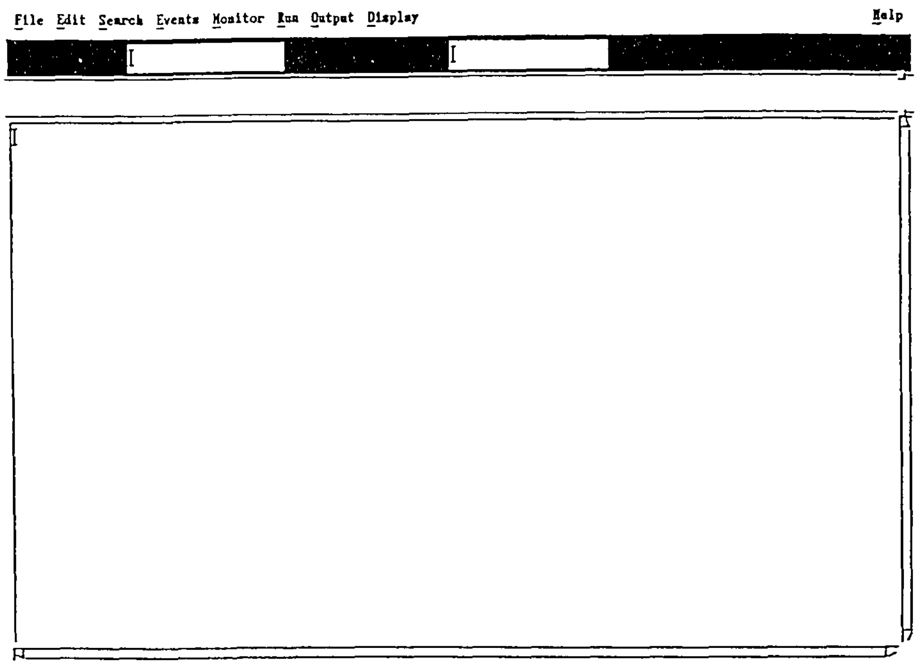


Figure 1. The Distributed Event Monitor User Interface Tool (DEMUIT)

define and **modify** all the events user wants to define, **edit** final file, **parse** the final file, and **run** the executable file. The result will display on the screen. The details of design and implementation, approaches and techniques are described in the following sections.

2. Event-Driven Environment and Object-Oriented Nature of the Programs

User interfaces design has evolved through three stages. The first stage produced the **command-driven** user interface, which presents the user with a generally cryptic prompt such as:"%". The user must know the set of commands that the interface recognizes and must enter those commands at the prompt. From the programmer's standpoint, this interface is the simplest because the code for this interface can be very straightforward and compact. From the user's standpoint, command-driven interfaces leave something to be desired, especially if the user is new to the system.

Interfaces of the second stage are **menu-driven** user interfaces. Structurally, menu-driven and command driven interfaces are similar. The main difference is that menu-driven interfaces have much more elaborate prompts. The user enters the appropriate choice according the prompt request, and the new menu appears. The code for this interface is fairly simple. It is less compact because all of the menus have to be displayed, but it does not pose a serious programming challenge.

Interfaces of the third stage are **event-driven**, which are more complex from the programmer's perspective. An event-driven environment consists of some type of application program interface (API), which provides a library of functions that create user interface objects such as menus, windows, buttons, scroll bars, and the like[4]. The user can manipulate these objects with the keyboard or mouse. Each time the user presses a key or clicks the mouse, the API picks up the action and delivers it to the program as an event. Usually, events are held in an event queue so that the program does not have to handle them in real time. Similarly, each time a user clicks the mouse, the click's location is packaged in a event record, which is also placed in the event queue. The pseudo-code for event-driven program might look like this:

```
draw the graphical objects onto the screen;
repeat
 wait for an event to appear in the event queue;
```

```
  parse the event;
  call the appropriate function to handle it;
until done;
```

The piece of looping code that receives and handles events like this is called an **event loop**. Event-driven programs have two main sources of complexity. First, many objects appear simultaneously on the screen, and many of them have separate parts. When the code detects a mouse event, it has to determine which object, and which part of that object, it effects. The second source of complexity is the many internal events the window system itself can generate. When part of one window is exposed by the movement of another window, the program receives an expose event. When a window is resized, the code gets a resize event. Focus change can generate focus events, and so on. Many lines of code are needed to parse out and handle all of the events. Much of this code is unique to each application, because each displays its own assortment of user interface objects.

The beauty of Motif is that it handles most of the low-level details for the programmer. Motif is a part of a UNIX library hierarchy which has four layers as shown below in Figure 2. At the bottom is UNIX and its standard libraries such as *stdio.h* and *math.h*. On top of the UNIX sits the X Window System and its library, accessed through *Xlib.h*. On top of X sits the X toolkit, accessed through *Intrinsics.h*. And finally, on top of the X Toolkit is Motif, accessed through *Xm.h*. The UNIX layer provides normal operating system support. The X layer provides basic windowing and event-handling capabilities[5,6]. The X Toolkit layer provides support for the creation and use of widget sets and Motif provides the widgets needed to create user interfaces easily.

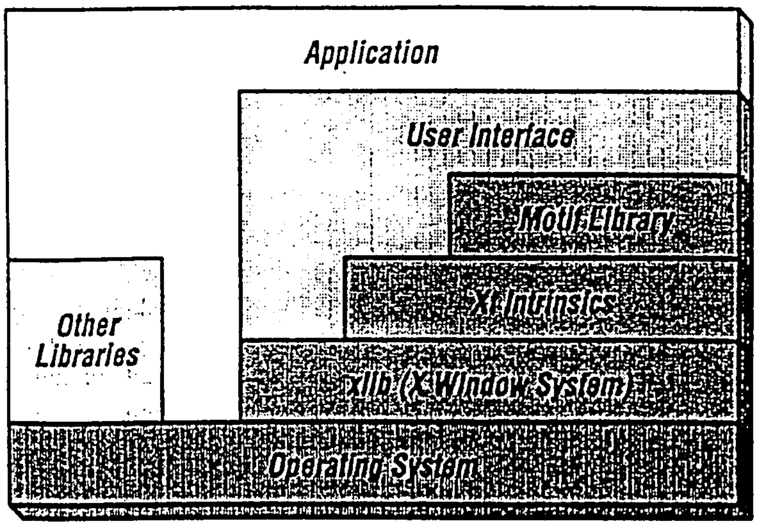


Figure 2. The Hierarchy of the Motif User-interface Library Model

The X Window System provides a basic event-driven programming environment. It runs on a workstation or on an X terminal. It controls the mouse, the keyboard, and the screen; packages events, places them in an event queue, and allows programs to draw graphical items on the screen. The X library provides a programming interface to the X Window System. This library is accessed by including its header files in the application code and then calling the appropriate routines to access the event queue, issue drawing commands, and so on. Xt is a general-purpose tool, it allows the programmer to design widget sets, but it does not enforce a particular **look and feel**. By design, widget sets appear very objective-oriented to the programmer. The Motif widget sets looks much like an object-oriented programming environment, but because the programming is all done in C rather than in an object-oriented

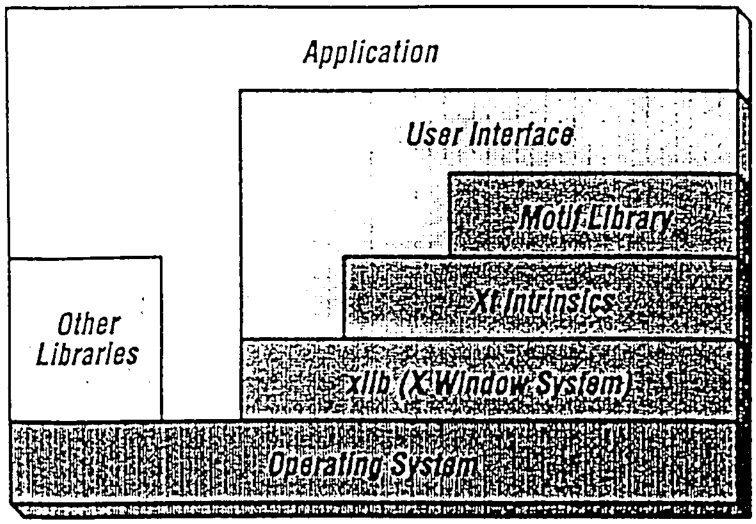


Figure 2. The Hierarchy of the Motif User-interface Library Model

The X Window System provides a basic event-driven programming environment. It runs on a workstation or on an X terminal. It controls the mouse, the keyboard, and the screen; packages events, places them in an event queue, and allows programs to draw graphical items on the screen. The X library provides a programming interface to the X Window System. This library is accessed by including its header files in the application code and then calling the appropriate routines to access the event queue, issue drawing commands, and so on. Xt is a general-purpose tool, it allows the programmer to design widget sets, but it does not enforce a particular **look and feel**. By design, widget sets appear very objective-oriented to the programmer. The Motif widget sets looks much like an object-oriented programming environment, but because the programming is all done in C rather than in an object-oriented language like C++, it is not completely object-oriented. In Motif programming, each user interface object is controlled by a set of variables called resources. By changing the resources, the programmer can control the appearance and behavior of the widget. By reading the resources, the programmer can find out about the widget's state. The widget can also send out messages, known as callbacks, when it wants to communicate with application code. Motif uses inheritance to build widgets on top of other widgets or out of groups of widgets. All Motif widgets use inheritance internally.

There are two significant advantages to handling user interface objects in this way. First, someone else has already coded the widget's appearance and behavior. Second, the widget handles all of the low-level event management. It animates itself and then, using callbacks, tells application code about it in a very controlled and simple way. Motif makes creating a graphical user interface easier by providing such functionality to the user.

3. Style Guide of the Program

The Open Software Foundation (OSF)/Motif style guide specifies the way an application should interact with the user[2,3]. The primary goal is to promote consistency. All applications with similar menus and dialog boxes should act in a similar manner. The components should look familiar and iteration should be familiar. The components should be organized in a familiar manner. The second goal is to allow the user to perform tasks by manipulating graphical representations of objects displayed on the screen such as pushing a button to start some action or dragging a slider to scroll the display. Flexibility is another key concern. The user should be able to perform a task many ways. This means that application should make use of the X resource manager to access these user-specified resources. The application should provide as much help information for the user as possible. The Applications are also expected to request confirmation from the user whenever an action may irreversibly destroy something.

To avoid common design pitfalls - it is helpful to design and build applications iteractively, that is build it, try it, see what does not work, change it, and so on.

4. Three Basic Concepts: Resources, Callbacks, and Managers

The heart of the Motif programming is to understand these three concepts: resources, callbacks, and managers.

1) Resources

The programmer designs a Motif application by selecting a set of widgets to compose the user interface. Every widget, in turn, has a set of associated resources that control its appearance and behavior. Resources are much like normal variables, except that they must be accessed in a special way. For example, a scrolled text widget created in the menu bar has resources that determine such features as 134 columns by 38 rows text display area, the edit mode is multi_line_edit, and the scrolling policy is automatic. The rest of the resources retain their default values. These resources can be read or set to new values. The programmer changes the widgets in the user interface by adjusting their resource values. The modification of widget resources is the key to controlling the behavior and appearance of each widget. To use a widget effectively, the programmer should be familiar with all of its resources, as well as the techniques for setting them. The resource lists provides this information for each widget.

There are two ways to set resources[2,4]. The first way is through the use of a structure called an argument list *Arg al[]*. This list is an array that contains pairs of items; the first item in pair is the resource's name, and the second is the resource's new value. The *ac* variable keeps track of the number of valid items in the argument list. Then using the *XtSetValue* function to set the values of those resources in the widget. This function accepts three parameters: the widget to set, the argument list, and a count. The typical codes look like:

```
Widget widget;

Arg al[10];

int ac;

XtSetArg(al[ac], resource's name, resource's new value); ac++;

......

XtSetValues(widget, al, ac);
```

The Xt layer supports a second way to pass an argument list to a existing widget, using *XtVaSetValues* function. In this case, the argument list is passed directly. the Xt layer also supports an *XtVaGetValues* function, which can be used to retrieve values from a widget. Both techniques were utilized in the implementation of DEMUIT.

2) Callbacks

All Motif widgets have callbacks, which can be used to trigger specific actions in response to user events. The idea behind callbacks is straightforward. If a user manipulates a widget on-screen, the program of the change must be notified. For example, a menu bar contains push-button widgets and when the user clicks the push button, it is expected that some specific action will result. Motif handles a mouse-click event in its main event loop (which is established by the call to the *XtAppMainLoop* function) and routes the event to the push-button widget. The push-button widget handles the event appropriately by making the button flash, but it also needs a way to communicate this event to the program. A *callback* function is a normal C function that performs an appropriate action. The address of the callback function is passed to the widget with the function *XtAddCallback*, and is thereby registered as a callback function for that widget. Whenever the widget detects a mouse event, it calls that function and the action occurs. The typical codes look like:

Widget widget;

String callback_name; /* The name of the callback */

XtCallbackProc callback; /* The function to call when the callback is triggered */

XtPointer client_data; /* Programmer specified data sent to the callback function */

XtAddCallback(widget, XmNactivateCallback, callback, client_data);

This code is used to add a callback function to the widget. The XtAddCallback tell the "widget" to call the function named "callback" whenever its activate callback is triggered.

3) Managers

Manager widgets are the backbone of the application. Without them, widgets have no way of controlling their sizes, layout, and input focus model[2,3,4]. In this user interface tool, we need to display a number of widgets simultaneously. For example, when program was executed, the menu bar displays multiple widgets such as: *MainWindowWidget, PaneWindowWidget, RowColumnWidget, TextWidget, ScrolledText* simultaneously. Here manager widgets handle the placement of multiple widgets in a single window.

Unlike primitive widgets such as *PushButtons, Scrolledbars, and Labels*, whose usefulness depends on their visual appearance and behavior, manage widgets provide no visual feedback and have few callback routings that react to user input.

Manager widgets have two purposes:

a) To manage the sizes and positions of the widgets they contain.

b) To provide special support for gadgets. A *gadget* is virtually the same as a *widget* from both the programmer's and the user's point of view. The main different between a gadget and a widget is that a gadget does **not** possess its own window, therefore it takes less time and memory to create, manage, and update on the screen.

In all other respects, manager widgets are like other widgets in that they have windows, can receive events, and can be manipulated directly through Motif or X Toolkit Intrinsics functions. This means that the users can draw directly into their windows, look for events, or specify resources for them. When it comes to supporting gadgets, all manager widgets are create equal.

There are many manager widgets classes, each of which is tuned especially for different kinds of widget layout. The DEMUIT used a variety of manager widgets class including *MainWindow, RowColumn, Form widget, PaneWindow, ScrolledWindow, and DrawingArea.*

Manager widgets manage other widgets. The relationship between managers and the widgets they manage is commonly referred to as the *parent-child* model. The manager acts as the parent and the other widgets are the children. A manager widget negotiates its child's size and position either by changing its size to accommodate the new child, or by changing the child's size to its own size. Since manager widgets can be children of other managers, this model provides for the widget-tree hierarchy, a framework for how widgets are laid out visually on the screen and how resources are specified in the resource database.

A manager may be created and destroyed like other widgets. The main difference between managers and other widgets is when they are declared to be managed in the creation process. The creation process of an entire widget tree is top down, but the management process is bottom up. For best results, *XtVaCreateManagedWidget()* should be used to create *primitive widgets*. *XtVaCreateWidget()* is used to create *manager widgets* and *XtManageChild()* is used to manage them later. If the programmer is adding another manager as a child, the principle applies; the programmer should also create them as an unmanaged widget until all their children are added as well. The idea is to descend as deeply into the widget tree and create as many children as possible before managing the manager parents as the programmer ascends back up. Once all the children have been added, *XtManageChild()* can then be called for the managers so that they only have to negotiate with their parents once, thus saving time, improving performance, and producing better results. This is very important principle. The entire motivation factor behind this principle is to optimize the method by which managers negotiate size and positions of their children.

5. Shell Widget, Application Context and Class, Command-line Arguments, and Fallback Resources

In DEMUIT, the following code was used to create the shell widget and is extremely important since it creates the only top-level window for this application. All the other widgets created are the children of this toplevel shell.

```
Widget toplevel = XtVaAppInitialize( XtAppContext * context,
                        String application_class,
                        XrmOptionDesRec option[],
                        Cardinal num_options,
                        Cardinal *argc,
        String *agv,
        String *fallback_resources,
        ArgList *args,
        Cardinal num_args);
......
XtRealizeWidget(toplevel);
XtAppMainLoop(context);
```

The first argument to *XtVaAppInitialize()* is the address of an *application context*, a structure in which Xt will manage some data internal to Xt that is associated with the application. The second argument to *XtVaAppInitialize()* is a normal C string that specifies the class name for the application. A class name is used in the resource database to specify values that will apply to all instances: a widget, or a resource. By convention, the class name is the same as the name of the application itself, except that the first letter is capitalized. The *options* and *num_options* parameters specify an array of strings which (custom command-line option) can be used to parse command line options that can set resource values. The other way to achieve this is to explicitly set all of the widget resource value directly using the *XtSetArg* and *XtSetValues* functions. The *&argc* and *argv* parameters specify the standard command line options. The fallback_resource parameter points to an array of strings that contain fallback resources. The shell widget will handle all of the application's interaction with the *Motif window manage, mwm*, and act as the "parent" of all other widgets in the application.

~~The widget that is returned by *XtVaAppInitialiaze()* is a *Shell widget.*~~ The shell widget initializes all of X and the X Toolkit and sets up the main application window. It parses out standard X command line options and fallback resources. Importantly, the programmer must pass Motif, Xt, and X functions exactly what they expect, or the program will create segmentation faults that can be very hard to track down, or addressing errors[4].

XtRealizeWidget() realizes the toplevel widget. When the toplevel shell is realized, the window frame that holds this application is created, along with the application's title, resizing borders, and so on. All of toplevel's child widgets are realized as well, and they too appear on the screen. In general, the toplevel is the only widget must be realized with an actual call, because the call to *XtRealizeWidget()* recursively realize all of its children.

XtAppMainLoop() causes the event loop to begin processing events. The event loop removes events from the X event queue and passes them to the appropriate widget for processing.

6. Functionality, Menus, and Menubars

The design process begins by deciding on what functionality the program needs. In DEMUIT, the user has to enter the event script definitions, and the program has to display the seven different event script forms for user input. Once all the event definitions have been completed, the user may modify the event definition file. The program also provides text editor cababilities, such as File, Edit, Search functions to do so. The user may use this editor to edit C program. This program also provides Run menu for the user to compile the program, parse the event script file, and run the executed file. In order to see the results of the event monitor evaluation, the program has an Output menu which can double check the user's defined events file, show the evaluation results, and some error messages if any. The display menu can be used to show the time, clock, and the order of events in this distributed system in color. Help menus provide some degree of help information. On line help can also be accessed from the event script definition push button. Once the programmer has determined the functionality, the programmer need to combine different Motif widgets to create the best interface.

Because the large number of the program options and complex of the functionality, the program could be a large one. A non-menu approach might require the programmer to display too many push buttons, and these push buttons would take up quite a bit of space. Menus economize space when the programmer has a large number of program options and commands. They organize different options in groups. The menu bar displays the name of the each group at the top of the application window. Clicking a name in the menu bar pulls down a customized *RowColumn widget* - a pulldown menu pane containing options associated with the menu name. The menu bar takes up very little space in the application, but it gives users access to a large number of program options organized by category.

Menus are somewhat more complicated to use, but once implemented they provide easy and intuitive ways to get commands and information from the user. This provides more advanced application interfaces. As shown previously in Figure 1, Figure 1 displayed the menu and menubars of the DEMUIT.

7. Creating of MainWindow, Build Menu, Menubars, Pulldown Menus, and Pullright Menus

1) Create Main Window

The *MainWindow* widget acts as the standard layout manager for the main application. The *XmMainWindow* is used as a toplevel window for this application. It contains a menubar, ScrollText, pane window widgets and label gadgets. The function *XtCreateManagedWidget()* is used to create an instance of the *MainWindow* widget, as shown in the following code fragment:

```
    #include <Xm/MainW.h>
    ......
    Widget toplevel, main_w, menubar;
    main_w = XtCreateManagedWidget("main_w",
                     xmMainWindowWidgetClass, toplevel,
                     resource-value list,
                     NULL);
```

The *MainWindow* widget class is defined in <Xm/MainW.h> which must be included whenever *MainWindow* widget will be created. The "resource-value list" provides fine control over the three-dimensional appearance of widgets.

2) Steps to Create Menu and Menubars

To create a menu and menubars is a fair complex operation. Several steps must be followed to create the menu.

First, the *XmCreateMenubar()* is used to create the menubar. For each title that appears in the menubar, a cascade button (pullrigh menu) along with the pulldown menu pane is then created with the menubar as its parent. The order in which the cascade buttons created determines the order in which they appear in the menubar.

Next a pulldown menu pane is created using *XmCreatePulldownMneu()* for each cascade button. The cascade buttons have a resource named **subitems**. Set this resource to the pulldown menu pane that will be used for cascade button. When a user clicks the cascade button, it manages the widget in the **subitems** resource, makes the menu pane visible. When the pulldown menu pane is created, do **not** manage the pane.

Finally, the options for individual menu panes is created using push buttons, with the pulldown menu pane as their parent. The order in which the programmer creates the push button determines the order in which they will appear in the pane. These buttons **should** be managed. When user clicks one of the push buttons, and its callback function causes the desired action to occur.

The above steps are repeated as needed for each title. The MenuBar was managed using *XtManageChild()*.

Menus are basically simple objects, and they contain many repetitive elements. While the simple menu creation routines are handy for basic prototyping and other simple application constructs, a great deal of redundant code will have to be generated if they are used for creating a large number of menus. The simple menu creation routines make it difficult to build a looping construct or a function to automate the process. To develop larger-scale application like this user interface tool needs use of abstraction and generalization techniques that are flexible enough to fill the menu functionality requirement.

3) Techniques to Build General Menu

In order to generate arbitrarily large MenuBars, titles, and pulldown menus using a substantially smaller code-set, the elements of a menu item must be identified. These elements include[2]:

. Labels

~~The labels are set for the menu items.~~

. Mnemonics

The mnemonics help user traverse menus or select actual menu items without having to use the mouse. For example, the user can use the key sequence Meta-F to open or close the file menu without using the mouse.

. Accelerators

The accelerator provides the user with the ability to activate menu in pulldown menu without having to display the menu at all. For example, if the Quit menu displayed the accelerator text Ctrl+Q to indicate that the user could press the Ctrl-Q keyboard sequence to activate that menu item and quit the application.

. Accelerator text

The accelerator text is the string (like convention recommended, such as "Ctrl+Q") that is displayed on the right side of the menu item.

. Callback routine

The callback routine is the routine called by any menu item. The callback functions for menu items are declared early. In *MenuItem file_menu[]*, the callback routine is *file_cb* for all items in File menu.

. Callback data

The callback data is the client data for callback routine. In *MenuItem file_menu[]*, the client data XtPointer 0 for callback routine *file_cb* is for the pulldown menu item "New".

Using this information a data structure *MenuItem* can be constructed that comprises all the important aspects of a menu item:

```
typedef struct _menu_item {
        char        *label;
        WidgetClass *class;
        char        mnemonic;
        char        *accelerator;
        char        *accel_text;
        void        (*callback)();
        Xtpointer   callback_data;
        struct _menu_item *subitems;
    } MenuItem;
```

To create a pulldown menu, all needed to do is to initialize an array of MenuItems and pass it to a routine that iterates through the array using appropriate information. For example, the following code initializes an array to contain all the elements for a File menu:

```
    MenuItem file_menu[] = {
        { "New", &xmPushButtonWidgetClass, 'N', NULL, NULL,
            file_cb, (XtPointer)0, (MenuItem *)NULL },
        { "Open", &xmPushButtonWidgetClass, 'O', NULL, NULL,
            file_cb, (XtPointer)1, (MenuItem *)NULL },
        { "Save", &xmPushButtonWidgetClass, 'S', NULL, NULL,
            file_cb, (XtPointer)2, (MenuItem *)NULL },
        { "Save As", &xmPushButtonWidgetClass, 'A', NULL, NULL,
            file_cb, (XtPointer)3, (MenuItem *)NULL },
        { "Close", &xmPushButtonWidgetClass, 'C', NULL, NULL,
            file_cb, (XtPointer)4, (MenuItem *)NULL },
        { "Quit", &xmPushButtonWidgetClass, 'Q', NULL, NULL,
                file_cb, (XtPointer)5, (MenuItem *)NULL },
        NULL,
    };
```

Two important design and implementation points are worth mentioning. First, it is advisable to use **widget class** push button instead of **gadget class** push button if the user wants the push button could pop up new window, in particular, the interface utilizes since the gadget class push button does not possess its own window, but the widget class push buttons have their window to pop up the dialog windows when user clicks them. The another important point the use of a **cascade button** instead of **push button** to implement pullright menu.

Each element in the *MenuItem* data structure is filled with default values for each menu item. Resource values that are not meaningful or hard-coded are initialized to NULL. The only field that cannot be NULL is the widget class. This design makes modification and maintenance very simple. One particular point of interest is the way the *WidgetClass* field is initialized. It is declared as a **pointer** to a widget class rather than just a widget class. Because of that, the field with the address of the widget class variable was initialized. The use of *&xmPushButtonWidgetClass* is one such example. The *xmPushButtonWidgetClass* pointer has no value until the program is actually running.

4) Writing a Generalized Function

The *MenuItem* data structure can now be utilized by writing a routine that pulls it all together to make menu. The *BuildPulldownMenu()* function, shown below, loops though each element in an array of pre-initialized *MenuItems* and creates menus from that information.

```
Widget
BuildPulldownMenu(parent, menu_title, menu_mnemonic, items)
        Widget parent;
        char *menu_title, menu_mnemonic;
        MenuItem *items;
{
        Widget PullDown, cascade, widget;
        int i;
        XmString str;
        PullDown = XmCreatePulldownMenu(parent, "_pulldown", NULL, 0);
        str = XmStringCreateSimple(menu_title);
        cascade = XtVaCreateManagedWidget(menu_title,
                        xmCascadeButtonWidgetClass, parent,
          XmNsubMenuId, PullDown,
                        XmNmnemonic, menu_mnemonic,
                        NULL);
        XmStringFree(str);
        for (i = 0; items[i].label != NULL; i++) {
                if (items[i].subitems)
                        widget = BuildPulldownMenu(PullDown,
                                items[i].label, items[i].mnemonic, items[i].subitems);
                else {
```

```
                widget = XtVaCreateManagedWidget(items[i].label,
                                items[i].class, PullDown,
                                NULL);
            }
            if (items[i].mnemonic)
                XtVaSetValues(widget, XmNmnemonic, items[i].mnemonic, NULL);
            if (items[i].accelerator {
                str = XmStringCreateSimple(items[i].accel_text);
                XtVaSetValues(widget,
                        XmNaccelerator, items[i].accelerator,
                        XmNacceleratorText, str,
                        NULL);
                XmStringFree(str);
            }
            if (items[i].callback)
                XtAddCallback(widget, XmNactiveCallback,
                        items[i].callback, items[i].callback_data);
        }
        return cascade;
    }
```

The function takes five parameters: the **parent** is a handle to a *MenuBar* widget that must have been created, the **menu_title** indicates what the title of the menu will be, the

menu_mnemonic should be set according to its title, the bold field at the end of the structure is a pointer to another array of MenuItems. If this pointer is not NULL, then the menu item points to another group of menu items that represent a cascading menu (pullright menu), and the last parameter is a array of **MenuItems**.

The three bold lines included in the above function fit in rather nicely with the rest of the routine. Because the function creates and returns a *CascadeButton*, the return value may be used as the menu item in the menu currently being built. But the menu must exist before it can be attached to a *CascadeButton*. Recursion handles that problem by creating the deepest submenus first and returning to the top later. This ensures that all the necessary submenus are build before their *CascadeButtons* require them.

By combining the above three techniques together: construct a abstract data structure *_menu_item*, initialize an array of *MenuItems*, and pass it to a generalized *BuildPulldownMenu()* function, an arbitrarily large *MenuBars* can be generated using small code-set. *BuildPulldownMenu()* function must be called from another function that passes the appropriate data structures and other parameters. The following shows the code for the *CreateMenuBar()* routine. This simple function creates a *MenuBar* widget calls *BuildPulldownMenu()*, manages the *MenuBar*, and then returns it to the calling function.

```
Widget menubar, widget, BuildPulldownMenu();

......

menubar = XmCreateMenuBar(main_w, "menubar", NULL, 0);

BuildPulldownMenu(menubar, "File", 'F', file_menu);

BuildPulldownMenu(menubar, "Edit", 'E', edit_menu);
BuildPulldownMenu(menubar, "Search", 'S', search_menu);
BuildPulldownMenu(menubar, "Events", 'E', events_menus);
BuildPulldownMenu(menubar, "Monitor", 'M', monitor_menus);
BuildPulldownMenu(menubar, "Run", 'R', run_menu);
BuildPulldownMenu(menubar, "Output", 'O', output_menu);
BuildPulldownMenu(menubar, "Display", 'D', display_menu);
widget = BuildPulldownMenu(menubar, "Help", 'H', h_menus);
XtVaSetValues(menubar, XmNmenuHelpWidget, NULL);
......
XtManageChild(menubar);
```

The program is composed of the function *BuildPulldownMenu()* and the menu and submenu declarations. The above example shows how to adjust the algorithms, data structures, and some detail design and implementation to fit the needs of the application by using optimized code.

8. Compound String, C String and Their Conversion

The string types and their conversion process is a very important design issue. A *compound string* is a means of encoding text so that it can be displayed in many different languages or fonts without changing anything in the program. The C programming language defines a string as a null-terminated array of characters. In order to use C string with Motif widgets, the C string must be converted to the *XmString* format. Motif provides functions to convert a C string to *XmString*, or *XmString* to a C string[2,3,4].

Motif supports its own string type, *XmString*. This type offers more functionality than a standard null_terminated C string. *XmString* is the data type for *compound string*. Compound strings include one or more components, each of which contains character set, string direction, and text. When a compound string is displayed, the character set and direction are used to determine how to display the text.

The conversion process is very important. Almost all Motif widgets require a compound string when specifying text. To display messages in the **label widgets, PushButtons, list widgets**, a C string must be converted to XmString. For example, the *labelString* resource is *XmString*, when use a call to XtSetArg as shown below:

```
XtSetArg(al[ac], XmNselectionLabelString, XmStringCreateLtoR(
                              "Enter the name of the new file", char_set));
```

It is very easy to forget the conversion and innocently try to pass a normal C string in the *XtSetArg* call. The code will compile correctly, but will result in core dump in the run time because the resource type for the *labelString* is *XmString*.

The most basic form of C string to compound string conversion is done using the function *XmStringCreateLtoR* and *XmStringCreateSimple* convert a C string to an *XmString*. These two functions are used throughout the entire interface program:

```
XmString

str = XmStringCreateSimple(text)

            char *text;

XmStringFree(str);
```

The *text* parameter is a common C string; the value returned is of type *XmString*. *XmStringCreateSimple()* creates compound strings and allocates memory to store the strings created. Widgets whose resources take compound strings as values always allocate their space and store copies of the compound string value given to them, so the copy of the string must be freed to prevent memory leaks[3] after having set it in the widget resource. A compound string is freed using *XmStringFree()*.

The call to *XmStringCreateLtoR* accepts two parameters: the C string to be converted, and the character set for the conversion. The character set is represented by the variable **char_set**, which is set to the value *XmSTRING_DEFAULT_CHARSET*. *XmStringCreateLtoR* also creates default string direction of left-to-right.

Unlike the other widgets, the **text widget**, does not use compound strings but C strings. The *XmTextSetString* function displays the C string on the text widget. The *XmTextGetString* gets the C string from the text widget. However, a conversion problem arises: to get strings from list widget and put them on the text widget, they must be converted from *Xmstring* to C string, or vice versa.

The conversion process from compound string to C string can be simple or complicated, depending on the complexity of the compound string to be converted. If the compound string has one character set associated with it and it has a left-to-right orientation, the process is quite simple. This is usually the case. To make the conversion, the following function can be used:

```
Boolean
XmStringGetLtoR(string, charset, text)
        XmString            String;
        XmStringCharSet     charset;
        char                **text;
```

Xtfree(text);

XmStringGetLtoR() takes a compound string and a character set and converts it back into a C character string. If successful, the function returns **True** and the **text** parameter will point to a newly allocated pointer to a string. Therefore, this pointer must be freed when you are through with it. The correct use of the conversions is important to prevent one of the common errors - core dump in run time.

9. Text Widget and Text Editor

The text widget is the most complicated of the Motif widgets, but it is also the most interesting and the most useful. It provides the following mechanisms for program control[2]:

. Resources that access the widget's text.

. Callback routines that enable "interposition" on events that add new text, delete text, and change input position or input focus.

. Keyboard management methods that control input, output, character positioning, and word-breaks or line-wrapping.

. Convenience routines that enable quick and simple access to the clipboard.

A mutliline edit scrolled text widget called text_w was utilized in DEMUIT. The code used to created this text widget and its resources is the following:

Widget pane, text_w;

Arg args[10];

XtSetArg(args[0], XmNrows, 38);

XtSetArg(args[1], XmNcolumns, 134);

XtSetArg(args[2], XmNeditable, False);

```
XtSetArg(args[3], XmNeditMode, XmMulti_LINE_EDIT);

XtSetArg(args[4], XmNscrollingPolicy, XmAUTOMATIC);

text_w = XmCreateScrolledText(pane, "text_w", args, 5);

XtManageChild(text_w);
```

When the program is run, the text widget is created on the main menu about 38 rows and 134 columns big. Characters can be typed and displayed on the widget. The user can press the ENTER key to begin a new line. The user can use the mouse or arrow keys to move the cursor around the text. Characters can be inserted at any location in the text and can be deleted at any location using BACKSPACE key. The user can scroll text back and forth by using scrolled bars in horizontal or vertical direction. This text widget provides full-featured text editing capabilities that can be used anywhere.

The convenience functions can be used to set or get the text widget's text. To get a text widget's text, *XmTextGetString()* is used, and to set a text widget's text, *XmTextSetString()* is used. Keeping in mind that the text widget only accepts C strings not XmStrings. The text widget provides a window in which the user can enter and edit text from keyboard. This tool makes full use of the text widget functionality in order to give the user the best interact with the application menu. The following is the full-feature text editor that was built from all information addressed so far.

10. File Menu Creation and All Dialog Boxes

The project requires defining the new event script file using or without using event definition form, modifying the event script file and monitored programs at any time. The DEMUIT designed the full-feature text editor in order to do so.

The text editor **File** menu contains New, **Open**, **Save**, **Save As**, **Clear**, and **Quit** pulldown menus. The user can use **New** option to creates a new program file or event script definition file without using event definition form. The user can also use the **Open** option to manage a file selection dialog box that allows the user to select a file to open and load for modification. The **Save** option saves file as the original file name. The **Save As** option saves a file as the name the user desired. The **Clear** option clears the text area. The **Quit** option exits the DEMUIT. The **File** menu provides the user with another way to modify the event script file even after user has input all events defined. The user can open the event file, check it, modify it, save it, or save it as new file.

In **File** menu, when the user selects different options, the DEMUIT needs to pop up different dialog boxes to satisfy the user's requirement, such as *file selection dialog*, *new_dialog*, *open_dialog*, *save_dialog*, *save_as_dialog*, *readonly_dialog*, *overwite_dialog*, *error_dialog*, *close_dialog*, and *quit_dialog*.

Dialog boxes are composed of a number of separated children bounded into a single widget. Let us take the *file selection dialog* box as example. The *file selection dialog* box is very powerful and provides a number of resources that contain such data as the current directory, the list of files, the filter string, and three buttons: *ok, filter,* and *cancel*. It lets user select a list of files available in the current directory. It also gives the user an intuitive way of traversing the directory structure. The children of a typical *file selection box* are shown in Figure 3. There are two techniques for manipulating the children that make up the dialog box. The first technique uses several resources that exist in the resource list for *file selection box* widget. These resources allow direct manipulation. For example, the labels on the three buttons have resources in the file selection box widget's resource list named *XmNokLabelString, XmNcancelLabelString,* and *XmNfilterLabelString*. Changing these resources modifies the labels on the three buttons. The second technique involves extracting the child's widget variable from the file selection box widget itself and then manipulating the child widget in the normal manner. In the file selection box widget, the extraction is done using a convenience function called *XmSelectionBoxGetChild*, which accepts as parameters the parent widget and a constant to identify the child.

The code for creating a file selection dialog box is shown as following:

```
Arg al[10];
int ac;
Widget open_dialog;
ac = 0;
XtSetArg(al[ac], XmNmustmatch, True); ac++;
XtSetArg(al[ac], XmNautoUnmanage, False); ac++;
XtSetArg(al[ac], XmNdialogTitle, XmStringCreateLtoR(
                                "DEM Editor: Open", char_set)); ac++;
open_dialog = XmCreateFileSelectionDialog(toplevel, "open_dialog", al, ac);
XtAddCallback(open_dialog, XmNokcallback, openCB, OK);
XtAddCallback(open_dialog, XmNcancelcallback, openCB, CANCEL);
XtUnmanageChild(XmSelectionBoxGetChild(open_dialog,
                                XmDIALOG_HELP_BUTTON));
```

In DEMUIT, *create_all_dialog()* function generates all dialog boxes for the file menu. The code for creating these dialog boxes is nearly identical to the code for creating the file selection dialog box, except for the widget name, title, and label of the push buttons.

Consider the **Open** option as example. When the user selects the **Open** button from the **File** menu in the menubar, the program pops up the *FileSelectionDialog* (as shown in Figure 3). The user can pick up the filename (e.g. menu.c), then press the **Ok** button in the file selection

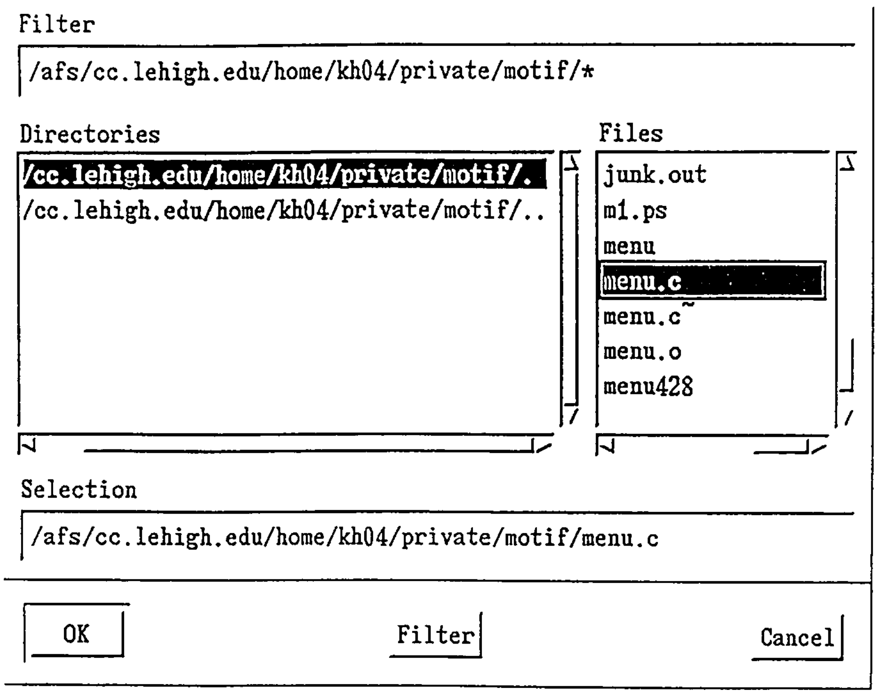
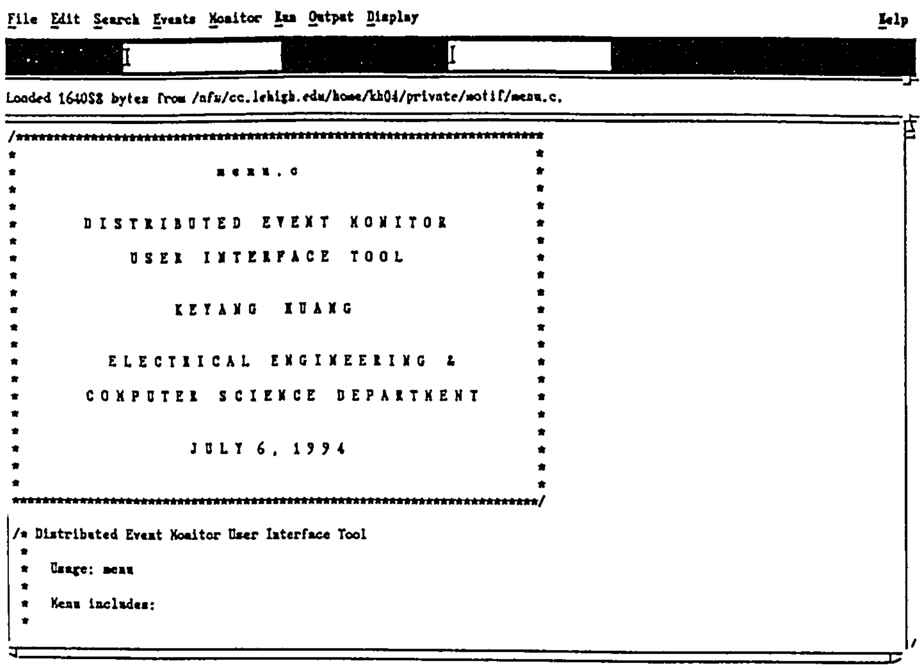

Filter
/afs/cc.lehigh.edu/home/kh04/private/motif/*
Directories
/cc.lehigh.edu/home/kh04/private/motif/.
/cc.lehigh.edu/home/kh04/private/motif/..
Files
junk.out
m1.ps
menu
menu.c
menu.c~
menu.o
menu428
Selection
/afs/cc.lehigh.edu/home/kh04/private/motif/menu.c
OK
Filter
Cancel

Figure 3. File Selection Dialog for Open File

text widget or press ENTER. The **Ok** callback function *read_file()* gets the value of the filename specified and checks its type. If the file chosen is not a regular file (e.g., if it is a directory) or if it cannot be opened, an error is reported and the function returns. When the file checks out, its contents are loaded into the text widget *text_w*. Rather than loading the file by reading each line using function like *fgets()*, the interface just allocates enough memory to contain the entire file and reads it all in with one call to *fread()*. The text is then loaded into the text widget using *XmTextSetString()*. The scrollbars are automatically updated and the text is positioned so that the beginning of the file is displayed first. The output of the open and load file menu.c is shown below in Figure 4. The user can utilize the information provided in the popup dialog boxes to make use of the other options in the **File** menu.



```
Loaded 164058 bytes from /afs/cc.lehigh.edu/home/kh04/private/motif/menu.c.

/****************************************************************************
*                                                                           *
*                          m e n u . c                                      *
*                                                                           *
*                                                                           *
*            D I S T R I B U T E D   E V E N T   M O N I T O R              *
*                                                                           *
*                 U S E R   I N T E R F A C E   T O O L                     *
*                                                                           *
*                                                                           *
*                       K E Y A N G   H U A N G                             *
*                                                                           *
*                                                                           *
*          E L E C T R I C A L   E N G I N E E R I N G   &                  *
*                                                                           *
*          C O M P U T E R   S C I E N C E   D E P A R T M E N T            *
*                                                                           *
*                                                                           *
*                       J U L Y   6 ,   1 9 9 4                             *
*                                                                           *
*                                                                           *
****************************************************************************/
/* Distributed Event Monitor User Interface Tool
 *
 *   Usage: menu
 *
 *   Menu includes:
 *
```

Figure 4. The Output of Opening File menu.c (DEMUIT source file)

11. Edit and Search Menu

To fulfill the project requirement of text edition, the DEMUIT provides Edit and Search menu which support the user with general text editor tool just like most word prossing software. The most common example of the data transfer model is **cut** and **paste**, a method by which the user can move or copy text between clipboard and primary selection as well as between the windows. When the user **selects** text, the selection should be placed on the clipboard. This is known as a **copy** operation. Retrieving text from the clipboard and placing it in the another window is known as a **paste** operation. In some cases, after the data is pasted from the clipboard, the original window will delete the data it copied. This is classified as a **cut** operation. The **Edit** menu contains **Cut**, **Copy**, **Paste**, and **Clear** functionalities. The cut option copies the primary selection to the clipboard and then deletes the primary selection. The copy option copies the primary selection to the clipboard. The paste option inserts the clipboard selection at the destination cursor. The clear option clears the primary selection. The text widget has convenience routines that support communication with the clipboard. All these clipboard functionalities work by default within the text widget. The *cut_paste()* function makes use of four text widget functions:

```
XmTextCut(widget, cut_time)
        Widget widget;
        Time cut_time;
XmTextCopy(widget, cut_time)
        Widget widget;
        Time cut_time;
XmTextPaste(widget)
        Widget widget;
XmTextClearSelection(widget, clear_time)
        Widget widget;
        Time clear_time;
```

The **Search** menu provides the easy way to find text strings, replace them with new strings. It contains **Find Next, Show All, Replace Text**, and **Clear** options. As an example, consider the text **search** usage. First, the file contents are loaded into the text widget *text_w*. Then a string pattern can be typed in the "Search pattern" text widget *search_w*. Pressing ENTER activates the search. The main text is searched starting at the position immediately following the current cursor position. If the pattern isn't found by the time the end of the text's string is reached, the searching resumes at the beginning of the text widget and continues until either the pattern is found or the original position is reached. If the pattern is found, the insertion point is moved to that location, the matched string is highlight; otherwise, an error message is printed at the another text widget called text_output and the insertion cursor does not move. In this program function *XmTextSetCursorPosition()* is used to set the cursor position, and function *XmTextSetInsertionPosition()* or *XmTextGetCursorPosition()* is used to get the current cursor position.

If the user wants a string pattern in the "Replace pattern" text widget *replace_w* to replace the Search pattern string, the program can perform the replacement automatically. To accomplish this, the function *XmTextReplace()* is used:

```
XmTextReplace(text_w, frompos, topos, value)
        Widget          text_w;
        XmTextPosition  frompos, topos;
        char            *value;
```

This function replaces the text widget starting at "frompos" up to but not including the position "topos" with the text in value. If value is NULL or an empty string, the text between the two positions is simply deleted. To remove the entire text of widget, call *XmTextSetString()* with a NULL string as the text value. This is done so often in reinitializing the user's input text widgets of the event definition.

12. General Approach and Techniques for Generalizing Event Definitions

Handling the user's input event script definitions is most important, complicate and difficult part of the DEMUIT. This involved the creating of the customized dialog boxes to define seven different event scripts. The requirements are: all the input event definition form should look and feel similar; when the user clicks the menu button, the DEMUIT should pop up a dialog box widget, and this widget should be a manager widget. It should be allowed to attach any other widgets. It should handle its own resizing as well as the resizing of the widgets it holds automatically.

1) Form Dialog Box and Shell

In order to create a **popup** customized dialog box, first, the programmer creates a *form dialog widget*, and **attach** the user interface widgets needed to this dialog box. The *FormDialog* creates a *dialog shell widget* and a *form widget.* The *form dialog widget* is the manager widget which is hooked into a dialog shell widget. The dialog shell allows the manager to act like dialog boxes when they appear on screen. The code used to create form dialog is shown below. As shown, the code should **not** manage the dialog until the programmer want it to appear on screen.:

```
Widget toplevel, form_dialog;

XmStringCharset char_set = XmSTRING_DEFAULT_CHARSET;

Arg al[10];

int ac;

ac = 0;
```

```
XtSetArg(al[ac], XmNdialogTitle, XmStringCreateLtoR
        ("User Define Event:", char_set)); ac++;
XtSetArg(al[ac], XmNautoUnmanage, False); ac++;
form_dialog = XmCreateFormDialog(toplevel, "form_dialog", al, ac);
```

Once the container dialog was created, it creates and manages all the user interface widgets that will make up the dialog. They should be created as children of the manager dialog, so that the dialog box will appear with all of the children in the correct places when the dialog is managed. The children's callbacks, resources, and so on, are all completely standard.

2) Form Widget and Resizing

Both *Bulletin board* and *Form widget* are manager widgets. But *Bulletin board* does not handle resizing very well. It only changes bulletin board size, the widgets it holds remain fixed when the user resizes windows. To solve this problem, *Form widgets* are the best choice because they automatically resize and reposition the widgets they hold when a user resizes the form.

The code used to create *Form widget* is shown below:

```
Widget rowcol_w, form;
form = XtVaCreateWidget("form",
                xmFormWidgetClass, rowcol_w,
                XmNfractionBase, 10,
                NULL);
```

When you attach other widgets to a form, these attached widgets change shape and size along with the form widget. There are four types of the attachments used in the program:

. Attachment to the form's edges:

XtSetArg(al[ac], XmNleftAttachment, XmATTACH_FORM); ac++;

. Attachment to another widget:

XtSetArg(al[ac], XmNtopAttachment, XmATTACH_WIDGET); ac++;

XtSetArg(al[ac], XmNtopWidget, sep); ac++;

. Attachment to a position on the form:

XtSetArg(al[ac], XmNbottomAttechment, XmATTACH_POSITION); ac++;

XtSetArg(al[ac], XmNbottomPosition, 30); ac++;

. Attachment to nothing:

XtSetArg(al[ac], XmNbottomAttachment, XmATTACH_NONE); ac++

Note that when attaching to another widget or attaching to a position, the code must set a pair of resources for the attachment to work. However all of the child widgets must be attached appropriately. It is easy to create bugs when attaching objects to a form widget, especially if the form has many children[4,7].

3) RowColumn Widgets and Push Buttons Managers

RowColumn widget is the most widely used and robust of all the manager widget. The widget lays out its children in the rows and columns. This is one of the features of the *RowColumn widget*. In DEMUIT, seven common used push buttons: **Modify**, **Done**, **Confirm**, **Save**, **Clear**, **Cancel** and **Help**, are set into identically sized boxes, and are chained together side by side. If all these buttons attached and placed to a *form widget*, all of this attaching and placing can become bothersome. The layout performed by *RowColumn widget* on its children is determined by the setting of resources. The children can be packed tightly in row and columns and they can be placed in boxes of the same size.

RowColumn widget is a *general-purpose composite widget* that manages the layout of its children. *RowColumn widget* may contain widgets of any type and requires no special knowledge about how those children function. This is exactly what was needed in event definition design. *Label gadgets* is created to define the event scripts; *label widgets* is created to show the usage information; *text field widgets* is created to input the user's data, *text widgets* is created to display the result, *radio boxes* is created to chose the options. *RowColumn widget* can be used whenever needed to manage sets of widgets as a group, such as *text* and *text field widgets* group, *label gadgets* and *label widgets* group, *radio boxes* group, *separator gadgets* group, *push button gadgets* group.

The typical code for creating *RowColumn widget* for the push buttons in primitive event is shown as following:

```
Widget prim_form, rowcol_x, psep;
rowcol_x = XtVaCreateWidget("rowcol_x",
                xmRowColumnWidgetClass, prim_form,
                XmNtopAttachment, XmATTACH_WIDGET,
                XmNtopWidget, psep;
                XmNorientation, XmHORIZONTAL,
                XmNpacking, XmPACK_COLUMN,
                NULL);
```

A *RowColumn* widget provides a number of resources. Orientation determines whether or not the RowColumn widget favors filling rows or columns as the container is resized. HORIZONTAL manages all seven push buttons in one row. Packing controls how widgets align.

PACK_COLUMN places seven push buttons in same-size boxes based on the largest child widget.

4) Radio Boxes and Options

Radio boxes let the user make one-of-many choices among a number of options. For example, in highlevel event definition eight operator names are defined: *or, sfl, and, then, conc, count, not, and notbtwn. Radio boxes* arrangement is the best choice for implementation because the user can make one-of-eight choice. Also in protocol event definition, three protocol types needed to be specified: *lc (logic clock)*, *vc (vector clock)*, and *sr (simultaneous region)*.

A *radio box* is implemented using a combination of *ToggleButton widgets or gadgets* and a *RowColumn manager widget*. Only one toggle button can be selected at any one time. This functionality is enforced by the *RowColumn* when the resource *XmNradioBehavior* is set to **True**. *Radio box* is created with a convenience function, as shown below for highlevel event operators:

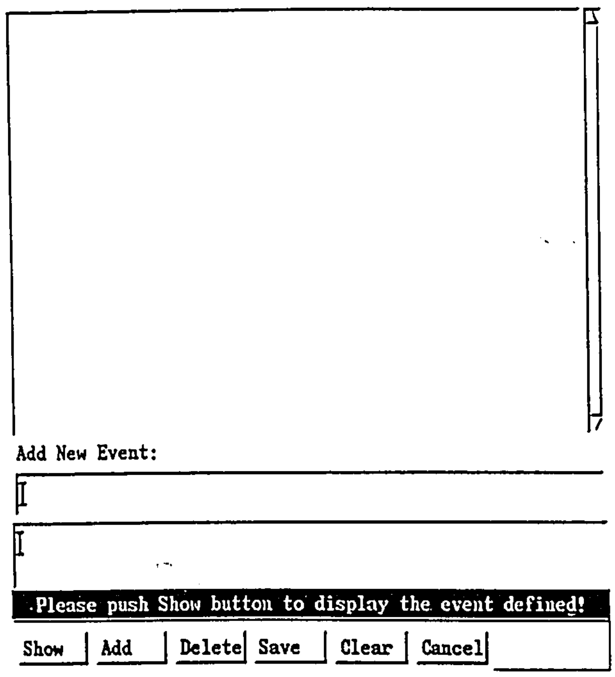
```
char *operator_labels[] = {"or", "sfl", "and", "then", "conc",
                                   "count", "not", "notbtwm"};
int j;
Widget radio_box, hrowcol_w, w;
......
radio_box = XtVaCreateWidget("radio_box",
                xmRowColumnWidgetClass, hrowcol_w,
                XmNpacking, XmPACK_COLUMN,
                XmNnumColumns, 4,
                XmNradioBehavior, True,
                NULL);
    for (j = 0; j < XtNumber(operator_lables); j++) {
        w = XtVaCreateManagedWidget(operator_label[j],
                xmToggleButtonGadgetClass, radio_box, NULL);
        XtAddCallback(w, XmNvalueChangedCallback, op1_toggled, j);
    XtmanageChild(radio_box);
```

When toggle buttons have a *radio box* as their parent, only one of the toggles can be set "ON" at any one time. When user clicks a toggle, for example "*and*", the "*and*" toggle turns on and all the other seven toggles turn off.

Using this general approach and techniques and carefully combining those different widgets discussed in this section together, the general event definition popup dialog form was created.

5) List Widget, Immediate Event Checking

The *list widget* was used to create another popup form for checking event definitions as shown below in Figure 5. This form pops up after the user defined the first event. The user can use this form to check the event just defined, add it or delete it before saving them to "*final.out*" file. This form is called "**List of Defined Events**". The top of the form is the *scrolled list widget* for displaying all the events defined. The middle of the form is the *label widget* called "Add New Event" and the *text field widget* to show the event just defined. The next widget attached to the bottom of the *text field widget* is the information display *text widget* and *label widget* for warning message. The button of the form is five *push button gadgets* managed by a *RowColumn* *widget*. These five buttons are: Show, Add, Delete, Save, Clear, and Cancel.



.Please push Show button to display the event defined!

Figure 5. The List Widget Popup Form

. Push **Show** button, the new defined event will display on the middle of the form.

. Push **Add** button, the new defined event will display on the top of the form; but disappears from the middle of the form.

. User can use browse to select event in the top list widget, the selected item is highlight. Push **Delete** button, the selected item is deleted.

. After checking all user defined events, The user can push the **Save** button to save all events into "final.out" file. Here XmStrings need to be converted to C strings, then events defined in the list widget can be save in C file. Finally the popup form is unmanaged.

. **Clear** button is used to clear the list widget area.

. **Cancel** button is used to unmanage this *List of Defined Event* form.

The *list widget* manages a list of *XmString* items on screen. It has adding, deleting, selecting capabilities. The selection policy was defined as **Browse** selection, i.e, the user can select one item at a time and can drag the cursor to change the selection.

13. Event Definition Menu

Event definition menu contains definition of primitive event, highlevel event, protocol event, goal event, local event, assign event, and place event. When the user clicks the menubar Event option, the program creates these seven events pulldown menus.

1) General Structure of the Event Definition Form

the *form dialog* was used to pop up the event definition form. The *scrolled window widget* then manages the *RowColumn widget*. The *RowColumn widget* manages the *form widget*, and the *form widget* manages all it children: event *labels gadgets* (on the left side, for example: **primitive** event **name, type, variable, funcname**) and *text field widgets* for user input (on the right side). The general *"for loop"* was used to define all labels. The *XmTRAVERSE_NEXT* was used to advance the cursor to the next *text field widget* for some events, such as radio boxes and toggle button gadgets for selecting options. For example, the protocol event types has three selections: lc, vc, and sr; the highlevel event has seven operators: or, sfl, and, then, conc, count, not, and notbtwn. All these selections are using *"for loop"* and managed by the *RowColumn widget* for automatically resizing. The *text widget* is used for information display attached just below the *RowColumn widget*. Below to the information *text widget* is the warning message *label widget*. The bottom of the form is another *RowColumn widget* which manages seven *push button gadgets*: **Modify**, **Done**, **Confirm**, **Save**, **Clear**, **Cancel**, **Help**. The functionality of those seven push buttons are shown below:

. **Modify** button allows user to modify the user's input information **before** the user pushes the **Done** button. the user can move the cursor to any text widget to reenter the input, then hit the RETURN key, the newly entered inputs are then recorded in data structure.

. **Done** button is used to record user input to a buffer and data structure.

. **Confirm** button is used to display user's event definition on information *text widget*.

. **Save** button is used to save the event definition to a file. When the first event is defined, the program pops up a file named *dialog form*, and asks the user to input the file name. After the first event was defined, all the following defined events are saved in the same file. After the event definition phase is finished, the event definition form is unmanaged.

. **Clear** button is used to clear the information text widget.

. **Cancel** button can be used to cancel the current event selection if the user hit the wrong pulldown menu.

. **Help** button can be used to get on line help for event script usage. The functionality is the same as select menubar **Help**, then select pulldown menu and pullright menu for this event.

2) General Steps for Input Event Definition

The general steps to create the events definition as shown below:

. Click menubar **Event**, the pulldown menus are displayed on the screen.

. Select any one of the event pulldown menus, click this selected menu.

. The *Dialog form* for selected event definition popups. For example, if the user selected

primitive event, then the primitive event definition dialog form pops up.

. For text widget input, move the cursor to the first user input text widget, click it to activate that widget.

. The user enters characters into the text widget from keyboard, according to the label specification on the left side of the text widget.

. The user **must** hit ENTER key after entering in all characters, the cursor then moves to the next line of the text widget automatically. This step is necessary, otherwise the input data will not be recorded.

. Repeat the above two steps until all text widgets are defined.

. For *radio box* selection, the user must move the cursor to the box desired and then click on it. For example, in highlevel event definition, if the user wants to choose the "**and**" operator, the user must move the cursor to the radio box located on the left of the "**and**" operator, and click on this box. The selection is then recorded.

. The user can push the **Help** button to get on line help of the event script usage at any time.

. If the user wants to modify some input, the cursor should be moved to the text widget the user wants to change, the desired changes are then typed and entered via ENTER key. For modifying the radio box selection, the cursor was moved to the box the user wants to select and the box selected by clicking.

Keep in mind, all modifications must be done **before** the **Done** Push button is hit.

. Check all inputs are Ok, then hit **Done** push button.

. Push **Confirm** button to display on the user's defined event on the information widget.

. Push **Save** button to save the event definition in the file.

. The program will pop up a file save dialog to ask the user to input the file name to save if this is the first event definition. All the following event definitions will be saved into this file then this event dialog form will be unmanaged.

. The program will pop up another **List of Defined Events** form for the user to double check the event just defined. This form will last until the **Save** button is hit on this form. The user could use this form to delete some event definitions.

. Push **Show** button on this form, the defined event will display on the text widget.

. Push **Add** button, this defined event will be added to the file named "final.out". The new added event shows on the top list widget.

. Repeat all above steps until all the definitions were entered.

. Push **Save** button, all events defined will be saved into "final.out". This popup form is unmanaged.

. Select menubar **Output**, click **Open** pulldown menu, the defined events display on the text widget of the main menu.

14. Command Widget and Run, Compile, Parser Menu

The Run menubar contains **Run**, **Compile**, and **Parser** menus. The user can use this menubar to compile the C file, run the executable file, and parse special defined file. The design strategy is to pop up a *form dialog widget* to manage a *command widget*. A *command widget* allows the user to enter commands and have them saved in a "history" list widget for later reference. The results of the command execution are display on the scrolled text area. Another way to achieve this is to issue the system call "aixterm" to pop up an aix window. On this window the user can compile a program, run executable file, parse the specified event file, and display all the information.

The *form dialog widget* is actually like a simple menubar. It contains *ScrolledText* area and *command widget* which prompts for system call. *Command widget* is a convenient interface for an application that has a command-driven interface such as compiling, running, parsing. The code used to create the *command widget* as shown bellow:

```
Widget command_w, cmd_w, menu_bar;
command_w = XtVaCreateWidget("command_w",
                    xmCommandWidgetClass, cmd_w,
                    XmNpromptString, file,
                    XmNtopAttachment, XmATTACH_WIDGET,
                    XmNtopWidget, menu_bar;
                    XmNleftAttachment, XmATTACH_FORM,
                    XmNrightAttachment, XmATTACH_FORM,
                    NULL);
XtManageChild(command_w);
```

When any polldown menu on Run menubar is pushed, a dialog pops up and asks the user to select a file to compile, to run or to parse. The user should select the correct file name. For example, to compile a C program, the extension of the file name should be ".c"; for run menu the file should be executable. After selecting a file name, the command dialog form pops up. The user can then enter the command and see the results on this form.

15. Output Menu

The Output menubar contains **Open**, **Clear**, and **Exit** menus. This menubar provides the user another way to look what has been done on the event definitions.

. The user can use this **Open** menu to check all events defined in the "final.out". When the user pushes the "open" menu, the contents of the file named "final.out" will display on the main menu text widget area.

. Clear menu can be used to clear the text widget area. This must be done before another file can be displayed on this text widget.

. Exit menu used to quit main menu and exit.

16. User's Manual

Welcome to Distributed Event Monitor User Interface Tool (DEMUIT). This user's manual shows, step by step, how to run and quit this tool, how to define seven events, how to check, modify, delete them, and how to modify the defined event script file using text editor supported by this tool.

1) What Is Needed to Run This Tool

This tool runs under Motif Graphical User Interface (GUI) 1.2 developed by Open Software Foundation (OSF) and X version 11 release 5 (X11 R 5) on any AIX RISC 6000 workstation.

After logging on any RISK 6000 workstation on campus, start X window by entering "startx" or "xinit". When running remote X clients, it is necessary to set the DISPLAY environment variable.

2) Running This Tool

At the aix prompt, type **menu** and press ENTER.

The main menu window will open with nine menubars from left to right:

File, Edit, Search, Event, Monitor, Run, Output, Display, and Help.

3) Quitting This Tool

The DEMUIT can be quit from the File menu.

. Click File menu to open the pulldown menu.

. Click Quit push button to quit this tool.

4) Defining Event Script File

Seven different event scripts, such as primitive event, ~~highlevel event, protocol event,~~

goal event, local event, assign event, and place event can be defined. There is no limit in the number of each event type which can be defined.

The general *event script form* consists scripts labels on the left, and empty text boxes on the right. The *information display box* will show the event scripts you defined. The *warning message box* shows what steps should be taken next. Seven push buttons are on the button of the form, they are: Modify, Done, Confirm, Save, Clear, Cancel, Help.

A step by step sample for each example is shown below:

a) Primitive Event Definition

The primitive event definition form is shown in Figure 6.

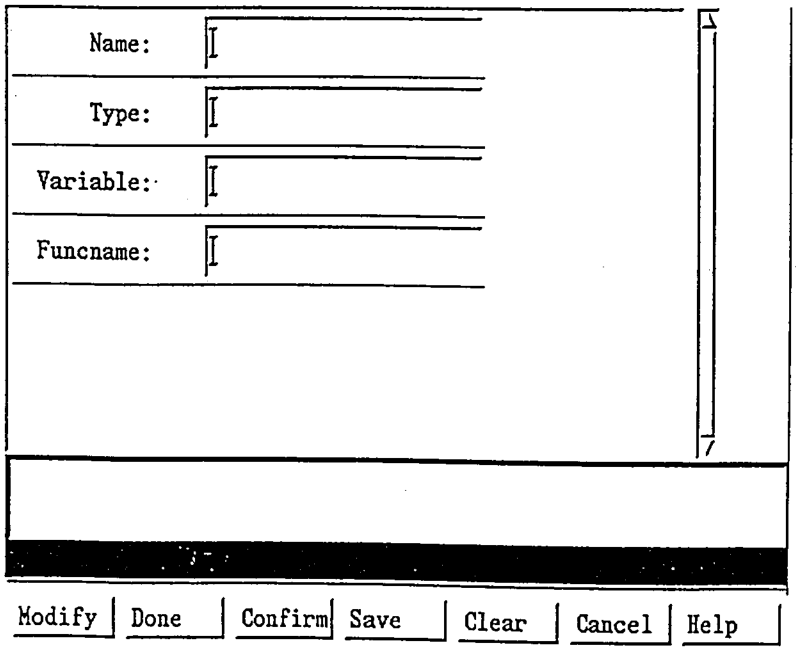


Figure 6. The Primitive Event Definition Form

The primitive event definition is defined as:

primitive <name> is <type> |<variable>| |<funcname>|

Example: **primitive E1 is access x main.**

To input this script:

. Push the pulldown menu Primitive. The primitive event definition form is open.

. Move the cursor to the first empty box, an *insertion point*(flashing vertical bar) appears at the far left side of the box. The text typed starts at the insertion point.

. Enter primitive event *Name* **E1** into this box, press ENTER (must).

. The cursor will move to the second empty box automatically.

. The information display box shows "primitive".

. Enter primitive event *Type* **access** into second box, press ENTER.

. The cursor moves to the third empty box automatically.

. Enter primitive event *Variable* **x** into third box, press ENTER.

. The cursor moves to the fourth empty box automatically.

. Enter primitive event *Funcname* **main** into fourth box, press ENTER.

. Push **Done** button.

. Push **Confirm** button. Information display box shows "E1 is access x main" next to the "primitive".

. Push **Save** button. This tool will open a save_event_dialog, ask you to "Enter the new file name to save the event definition", if this is the first event defined. This dialog only happens once at first event definition.

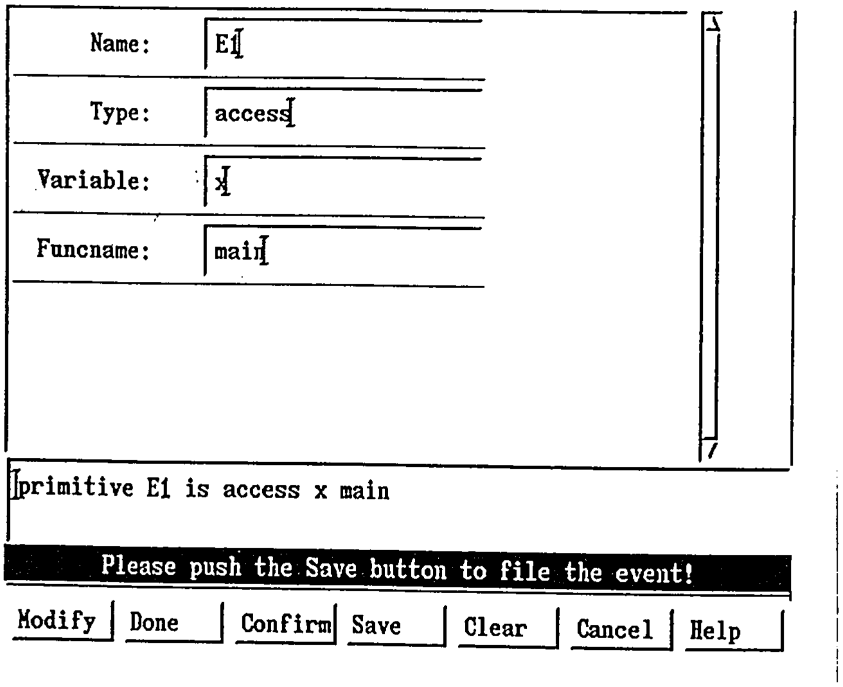
. Enter the file name you desired. Press ENTER or push the Ok button on dialog form.

. The event checking form pops up for checking the user defined event. For details see section 5) Checking Defined Event below.

. The primitive event definition form is unmanaged. Another event can be selected or the primitive event can be selected again.

The "primitive E1 is access x main" is save into the file named by the user.

The completed this event definition is shown as in Figure 7.



Name:	E1
Type:	access
Variable:	x
Funcname:	main

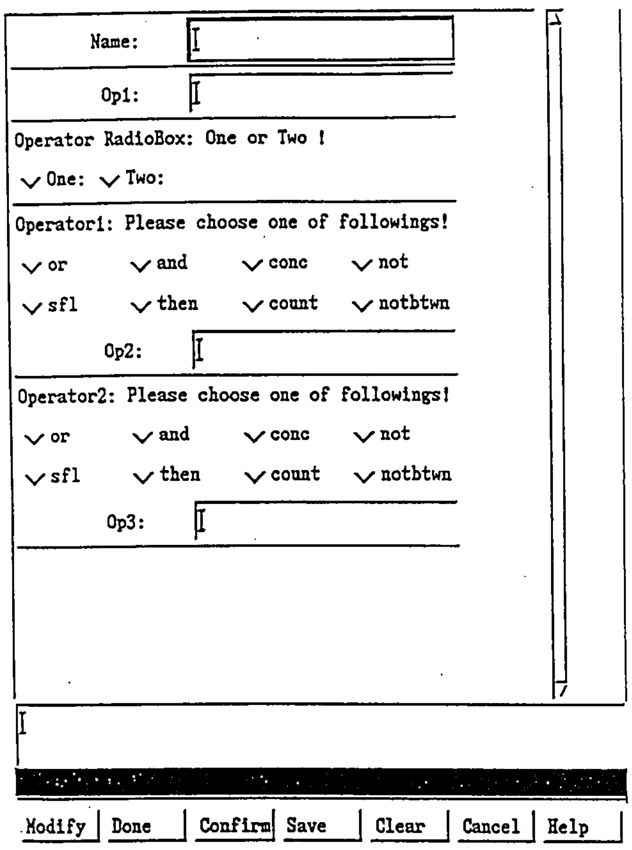
```
primitive E1 is access x main
```

Please push the Save button to file the event!

Figure 7. The Completed "primitive E1 is access x main" Definition

b) Highlevel Event Definition

The highlevel event definition form is shown below in Figure 8.


Name:

Op1:

Operator RadioBox: One or Two !

One: Two:

Operator1: Please choose one of followings!

or	and	conc	not
sfl	then	count	notbtwn

Op2:

Operator2: Please choose one of followings!

or	and	conc	not
sfl	then	count	notbtwn

Op3:

Figure 8. The Highlevel Event Definition Form

The highlevel event is defined as:

highlevel <name> is <op1> <operator> <op2> |<operator>| |<op3>|

There are three different cases:

i) highlevel <name> is <op1> <operator> <op2>

Where operator is ont include *notbtwn*.

ii) highlevel <name> is <op1> *notbtwn* <op2> <op3>

iii) highlevel <name> is <op1> <operator> <op2> <operator> <op3>

Example 1: **highlevel E7 is E1 or E2.**

To input this script:

. Push the pulldown menu Highlevel. The highlevel event definition form will open.

. Move the cursor to the first empty box, an *insertion point* appears at the far left side of this box. The text typed starts at the insertion point.

. Enter highlevel event *Name* **E7** into this box, press ENTER.

. The cursor moves to the second empty box automatically.

. The information display box shows "highlevel".

. Enter the highlevel event *op1* **E1** into second box, press ENTER.

. Move the cursor to the operator radio box: "One or two!" Press the radio box One.

. Move the cursor to the operator1 radio box: "Please choose one of followings!" Press the radio box **or**.

. Move the cursor to the Op2 empty box, click on it. An *insertion point* appears at the far left side of this box.

. Enter the highlevel event *op2* **E2** into the third box, press ENTER.

. Do not push radio box on operator2.

. Do not enter anything into op3 empty box.

. Push **Done** button.

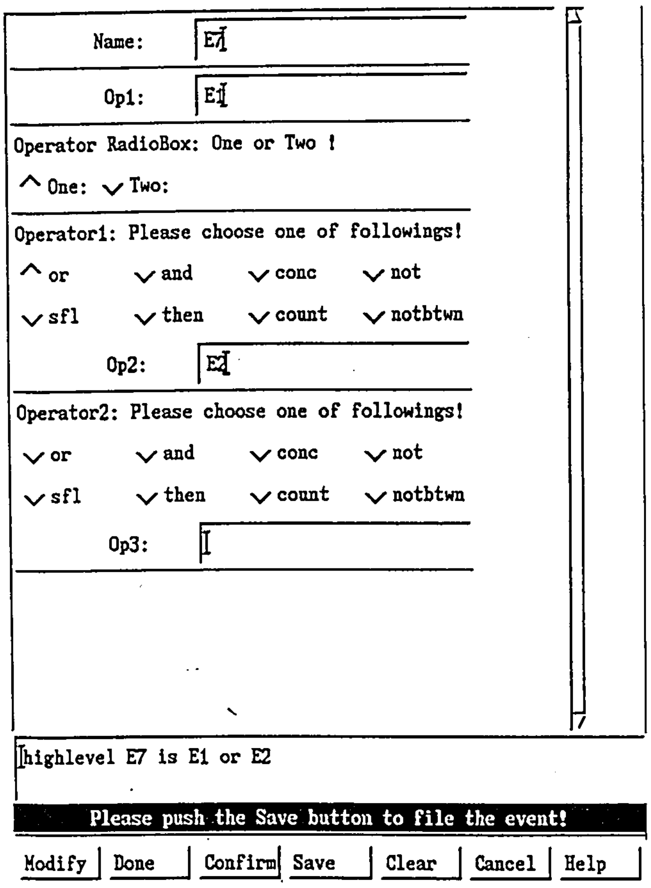
. Push **Confirm** button. Information display box shows "E7 is E1 or E2" next to the "highlevel".

. Push **Save** button. "highlevel E7 is E1 or E2" is save into the same file defined for the first event.

. Use the event checking form to check the defined event. Details see section 5) below.

. The highlevel event definition form is unmanaged. Another event can be selected or the highlevel event can be selected again.

The completed this event definition is shown in Figure 9.



Name: E7

Op1: E1

Operator RadioBox: One or Two !

^ One: ∨ Two:

Operator1: Please choose one of followings!

^ or ∨ and ∨ conc ∨ not

∨ sfl ∨ then ∨ count ∨ notbtwn

Op2: E2

Operator2: Please choose one of followings!

∨ or ∨ and ∨ conc ∨ not

∨ sfl ∨ then ∨ count ∨ notbtwn

Op3:

highlevel E7 is E1 or E2

Please push the Save button to file the event!

Figure 9. The Completed "highlevel E7 is E1 or E2" Definition

Example 2: **highlevel E4 is E6 notbtwn E9 E5**.

To input this script:

. Push the pulldown menu Highlevel again. The highlevel event definition form will open.

. Move the cursor to the first empty box, an *insertion point* appears at the far left side

of this box. The text typed starts at the insertion point.

. Enter the highlevel event *Name* **E4** into this box, press ENTER (must).

. The cursor moves to the second empty box automatically.

. The information display box shows "highlevel".

. Enter the highlevel event *op1* **E6** into the second box, press ENTER.

. Move the cursor to the operator radio box: "One or two!" Press the radio box **One**.

. Move the cursor to the operator1 radio box: "Please choose one of followings!" Press the radio box **notbtwn**.

. Move the cursor to the Op2 empty box, click on it. An *insertion point* appears at the far left side of this box.

. Enter the highlevel event *op2* **E9** into the third box, press ENTER.

. Do not choose operator2 radio box.

. Enter the highlevel event *op3* **E5** into the fourth empty box. Press ENTER.

. Push **Done** button.

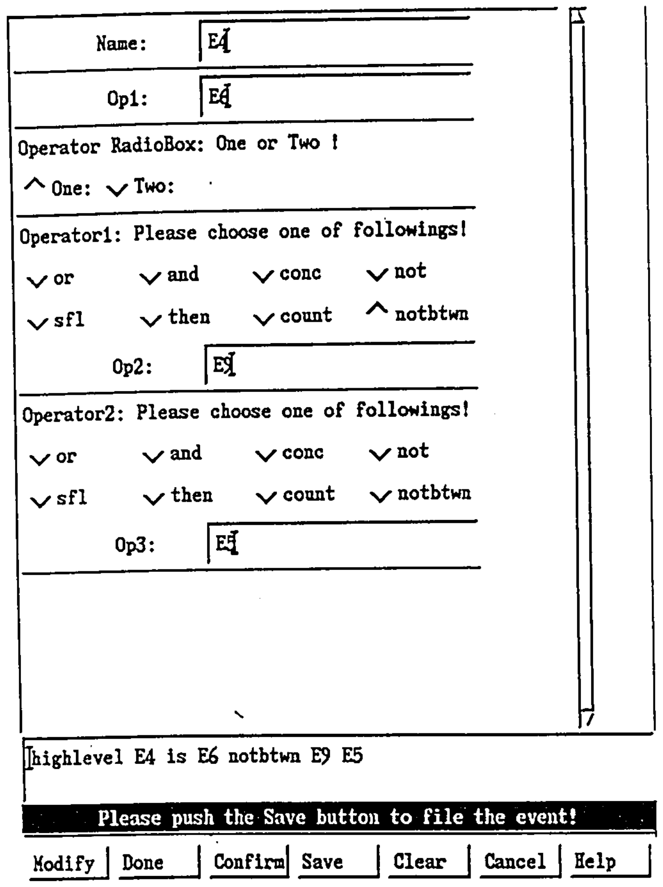
. Push **Confirm** button. The information display box shows "E4 is E6 notbtwn E9 E5" next to the "highlevel".

. Push **Save** button. "highlevel E4 is E6 notbtwn E9 E5" is save into the same file defined for the first event.

. The highlevel event definition form is unmanaged.

. Use the event checking form to check the defined event. Details see section 5) below.

The completed this event definition is shown in Figure 10.



Name: E4

Op1: E6

Operator RadioBox: One or Two !

^ One: ✓ Two:

Operator1: Please choose one of followings!

✓ or ✓ and ✓ conc ✓ not

✓ sfl ✓ then ✓ count ^ notbtwn

Op2: E9

Operator2: Please choose one of followings!

✓ or ✓ and ✓ conc ✓ not

✓ sfl ✓ then ✓ count ✓ notbtwn

Op3: E5

highlevel E4 is E6 notbtwn E9 E5

Please push the Save button to file the event!

Figure 10. The Completed "highlevel E4 is E6 notbtwn E9 E5" Definition

Example 3: **highlevel E1 is E2 and E3 then E4.**

To input this script:

. Push the pulldown menu Highlevel again. The highlevel event definition form will open.

. Move the cursor to the first empty box, an *insertion point* appears at the far left side

of this box. The text typed starts at the insertion point.

. Enter the highlevel event *Name* **E1** into this box, press ENTER (must).

. The cursor moves to the second empty box automatically.

. The information display box shows "highlevel".

. Enter the highlevel event *op1* **E2** into the second box, press ENTER.

. Move the cursor to the operator radio box: "One or two!" Press the radio box **Two**.

. Move the cursor to the operator1 radio box: "Please choose one of followings!" Press the radio box **and**.

. Move the cursor to the Op2 empty box, click on it. An *insertion point* appears at the far left side of this box.

. Enter the highlevel event *op2* **E3** into the third box, press ENTER.

. Move the cursor to the operator2 radio box: "Please choose one of followings!" Press the radio box **then**.

. Enter the highlevel event *op3* **E4** into the fourth empty box. Press ENTER.

. Push **Done** button.

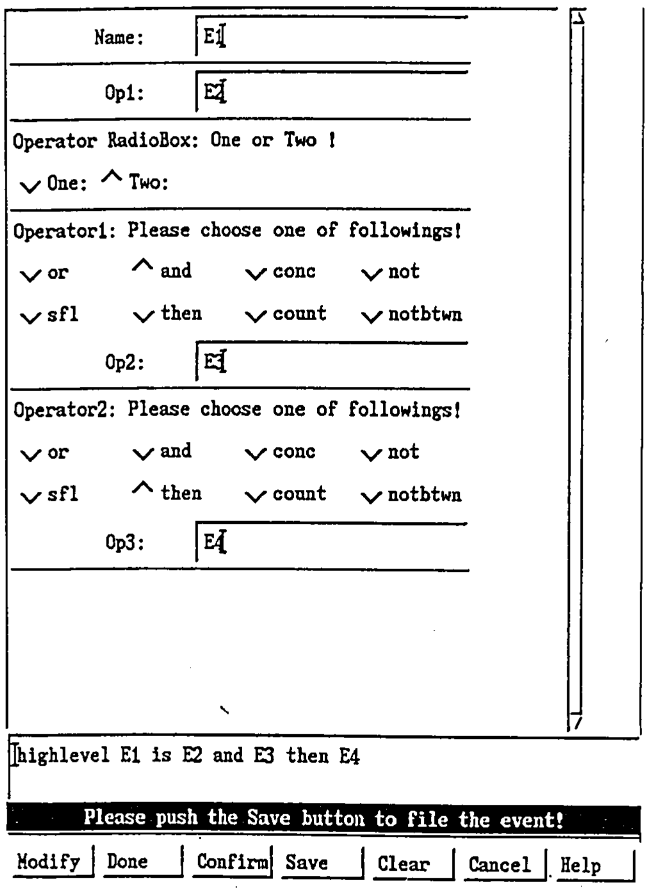
. Push **Confirm** button. the information display box shows "E1 is E2 and E3 then E4" next to the "highlevel".

. Push **Save** button. "highlevel E1 is E2 and E3 then E4" is save into the same file defined for the first event.

. The highlevel event definition form is unmanaged.

. Use the event checking form to check the defined event. Details see section 5) below.

The completed this event definition is shown in Figure 11.



Name: E1

Op1: E2

Operator RadioBox: One or Two !

One: [✓ Two:]

Operator1: Please choose one of followings!

or	[✓ and]	conc	not
sfl	then	count	notbtwn

Op2: E3

Operator2: Please choose one of followings!

or	and	conc	not
sfl	[✓ then]	count	notbtwn

Op3: E4

highlevel E1 is E2 and E3 then E4

Please push the Save button to file the event!

Figure 11. The Completed "highlevel E1 is E2 and E3 then E4" Definition

c) Protocol Event Definition

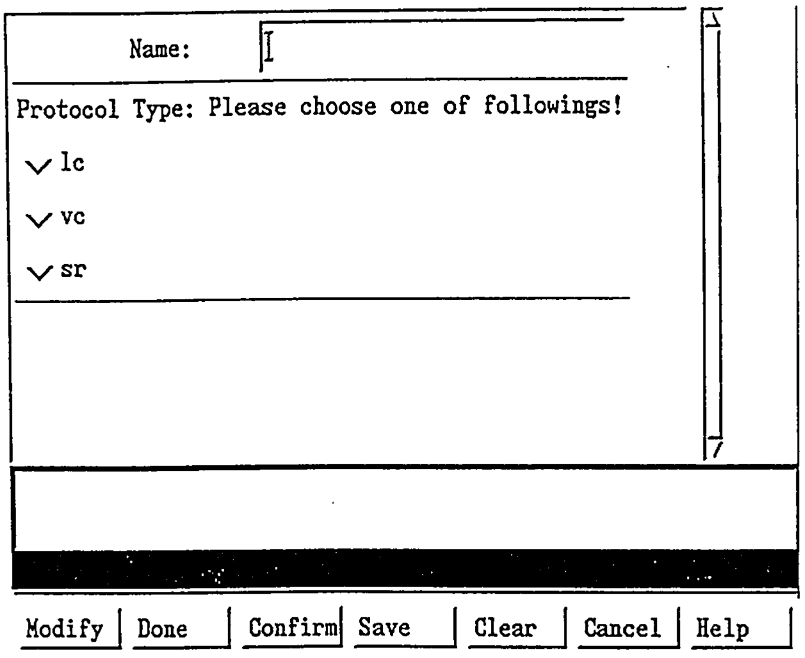
The protocol event definition form is shown below in Figure 12.

The protocol event is defined as:

protocol <name> is <type>

Example: **protocol all is lc**. To input this script:

. Push the pulldown menu Protocol. The protocol event definition form will open.


Name:

Protocol Type: Please choose one of followings!

- lc
- vc
- sr

Figure 12. The Protocol Event Definition Form

. Move the cursor to the first empty box, an *insertion point* appears at the far left side of the box. The text typed starts at the insertion point.

. Enter the protocol event *Name* **all** into this box, press ENTER.

. The information display box shows "protocol".

. Move the cursor to the Protocol Type radio box: "Please choose one of the followings!" Press the radio box **lc**.

. Push **Done** button.

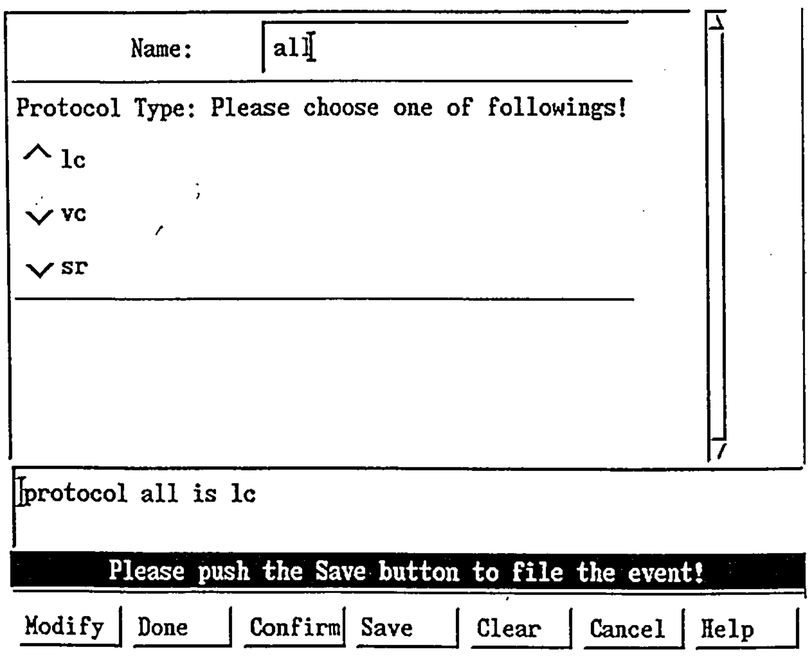
. Push **Confirm** button. The information display box shows "all is lc" next to the "protocol".

. Push **Save** button. "protocol all is lc" is save into the same file defined after the first event.

. The protocol event definition form is unmanaged. Another event can be defined or the protocol event can be defined again.

. Use the event checking form to check the defined event. Details see section 5) below.

The completed this event definition is shown in Figure 13.


Name: all

Protocol Type: Please choose one of followings!

protocol all is lc

Please push the Save button to file the event!

Figure 13. The Completed "protocol all is lc" Definition

d) Goal Event Definition

The goal event definition form is shown in Figure 14.

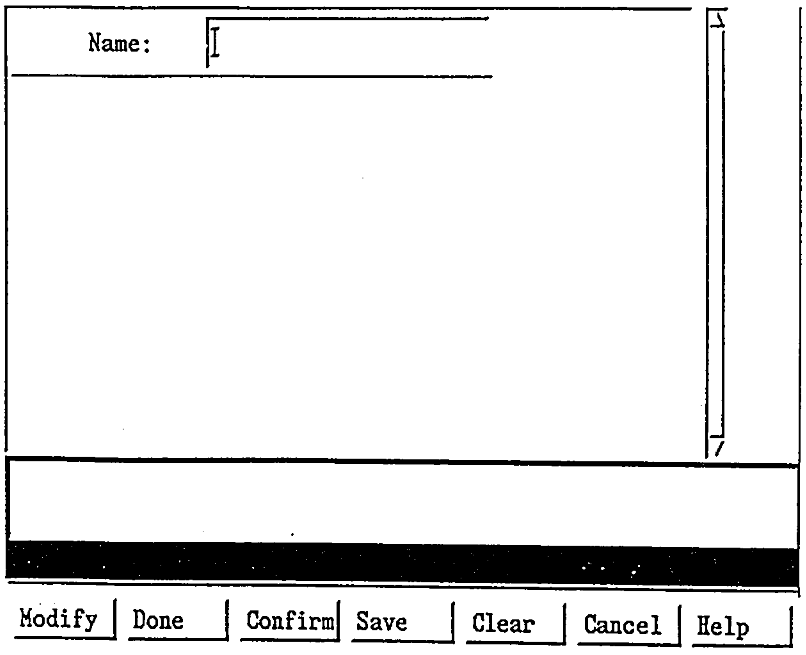


Figure 14. The Goal Event Definition Form

The goal event is defined as:

goal <name>

Example: **goal E7**.

To input this script:

. Push the pulldown menu Goal. The goal event definition form will open.

. Move the cursor to the first empty box, an *insertion point* appears at the far left side of the box. The text typed starts at the insertion point.

. Enter the goal event *Name* **E7** into this box, press ENTER.

. The information display box shows "goal".

. Push **Done** button.

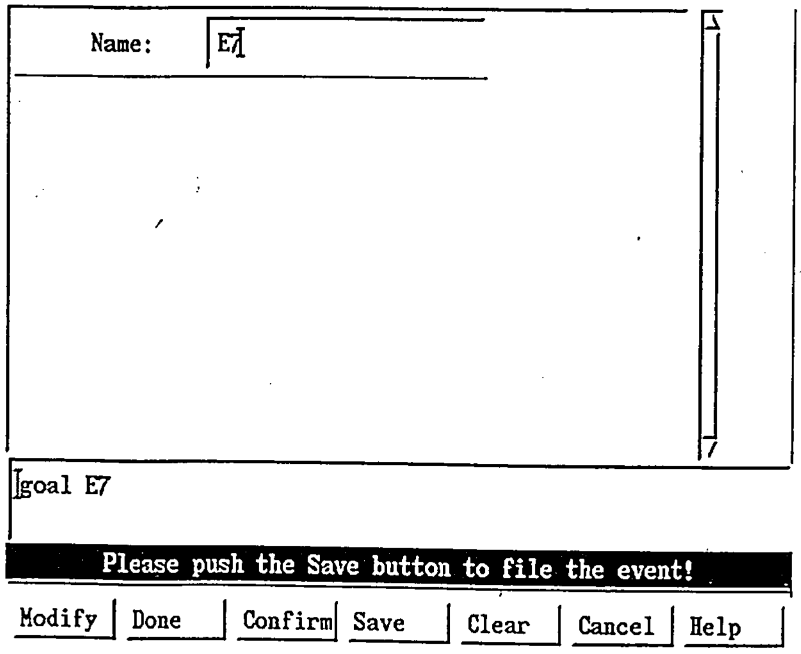
. Push **Confirm** button. The information display box shows "E7" next to the "goal".

. Push **Save** button. "goal E7" is save into the same file defined for the first event.

. The goal event definition form is unmanaged. Another event can be defined or the goal event can be defined again.

. Use the event checking form to check the defined event. Details see section 5) below.

The completed this event definition is show in Figure 15.



Name: E7

goal E7

Please push the Save button to file the event!

Figure 15. The Completed "goal E7" Definition

e) Local Event Definition

The local event definition form is shown in Figure 16.

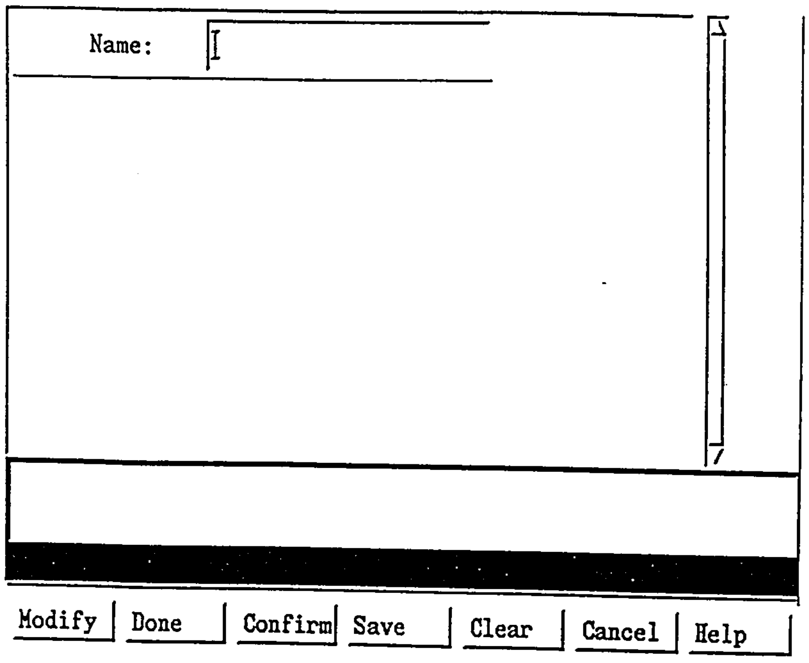


Figure 16. The Local Event Definition Form

The local event is defined as:

local <name>

Example: **local E7**.

To input this script:

. Push the pulldown menu Local. The local event definition form is open.

. Move the cursor to the first empty box, an *insertion point* appears at the far left side

of the box. The text typed starts at the insertion point.

. Enter the local event *Name* **E7** into this box, press ENTER.

. The information display box shows "local".

. Push **Done** button.

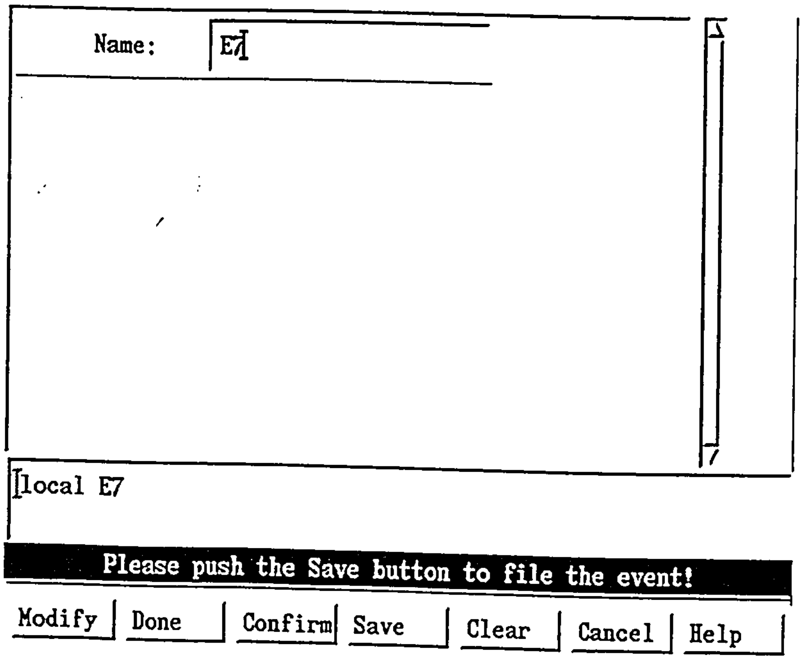
. Push **Confirm** button. The information display box shows "E7" next to the "local".

. Push **Save** button. "local E7" is save into the same file defined for the first event.

. The local event definition form is unmanaged. Another event can be defined or the local event can be defined again.

. Use the event checking form to check the defined event. Details see section 5) below.

The completed this event definition is shown in Figure 17.



local E7

Please push the Save button to file the event!

Figure 17. The Completed "local E7" Definition

f) Assign Event Definition

The assign event definition form is shown in Figure 18.

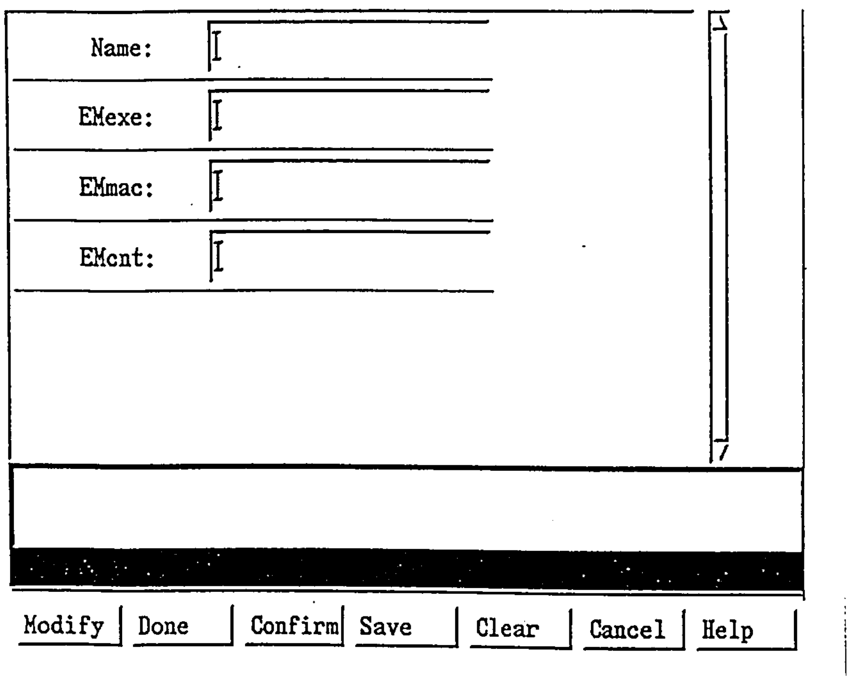


Figure 18. The Assign Event Definition Form

The assign event is defined as:

assign <name> to <EMexe> <EMmac> <EMcnt>

Example: **assign all to em pl118f.cc.lehigh.edu 1**.

To input this script:

. Push the pulldown menu Assign. The assign event definition form will open.

. Move the cursor to the first empty box, an *insertion point* appears at the far left side of this box. The text typed starts at the insertion point.

. Enter the assign event *Name* **all** into this box, press ENTER.

. The cursor moves to the second empty box automatically.

. The information display box shows "assign".

. Enter the assign event *EMexe* **em** into the second box, press ENTER.

. The cursor moves to the the third empty box automatically.

. Enter the assign event *EMmac* **pl118f.cc.lehigh.edu** into the third box, press ENTER.

. The cursor moves to the fourth empty box automatically.

. Enter the assign event *EMcnt* **1** into the fourth box, press ENTER.

. Push **Done** button.

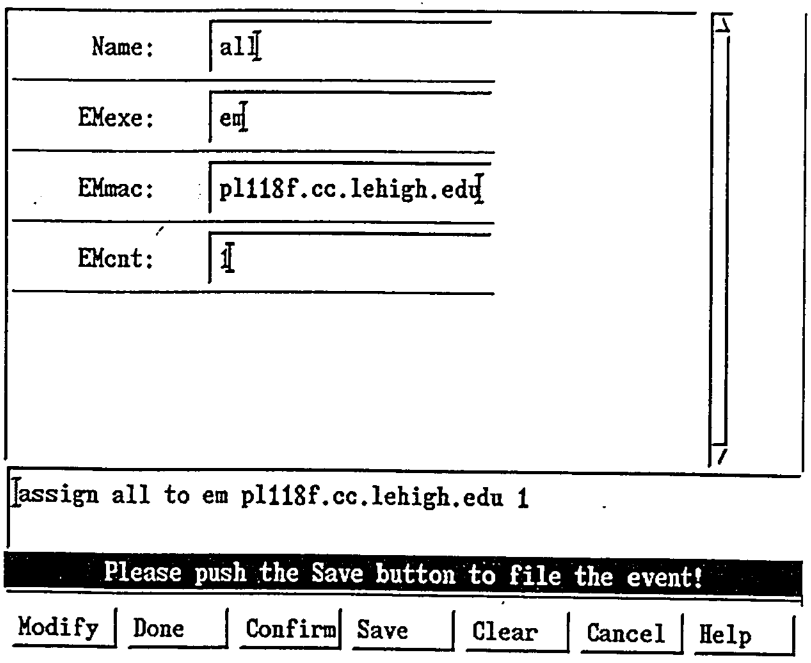
. Push **Confirm** button. The information display box shows "all to em pl118f.cc.lehigh.edu 1" next to the "assign".

. Push **Save** button. "assign all to em pl118f.cc.lehigh.edu 1" is save into the same file defined for first event.

. The assign event definition form is unmanaged. Another event can be defined or the assign event can be defined again.

. Use the event checking form to check the defined event. Details see section 5) below.

The completed this event definition is shown in Figure 19.



Name:	all
EMexe:	em
EMmac:	pl118f.cc.lehigh.edu
EMcnt:	1

```
assign all to em pl118f.cc.lehigh.edu 1
```

Please push the Save button to file the event!

Figure 19. The Completed "assign all to em pl118.cc.lehigh.edu 1" Definition

g) Place Event Definition

The place event definition for is shown in Figure 20.

The place event is defined as:

place <name> at <procexe> <procmac> <proccnt>

Example: **place all at myprog pl118a.cc.lehigh.edu 1**.

To input this script:

. Push the pulldown menu Place. The place event definition form will open.

. Move the cursor to the first empty box, an *insertion point* appears at the far left side of this box. The text typed starts at the insertion point.

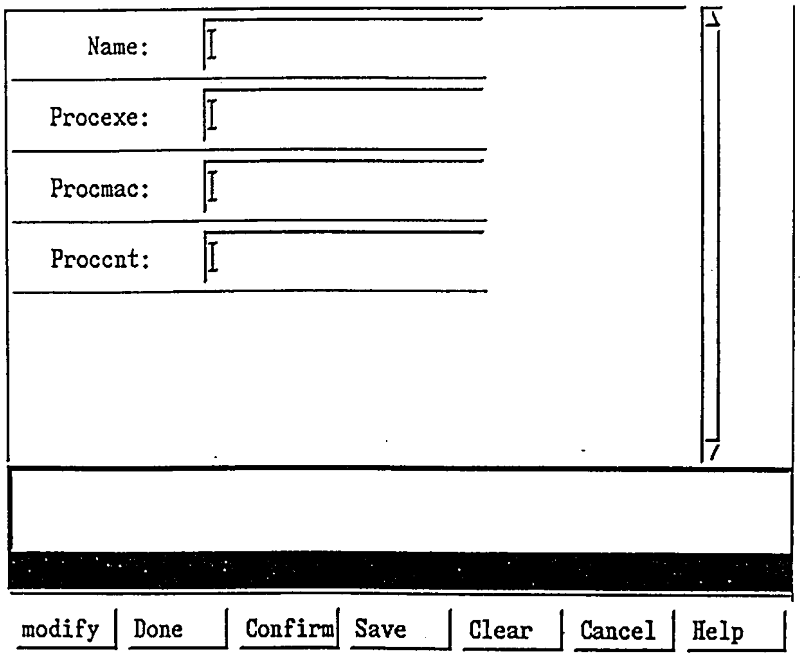


Figure 20. The Place Event Definition Form

. Enter the place event *Name* **all** into this box, press ENTER.

. The cursor moves to the second empty box automatically.

. The information display box shows "place".

. Enter the place event *procexe* **myprog** into the second box, press ENTER.

. The cursor moves to the third empty box automatically.

. Enter the place event *procmac* **pl118a.cc.lehigh.edu** into the third box, press ENTER.

~~. The cursor moves to the fourth empty box automatically.~~

. Enter the place event *proccnt* **1** into the fourth box, press ENTER.

. Push **Done** button.

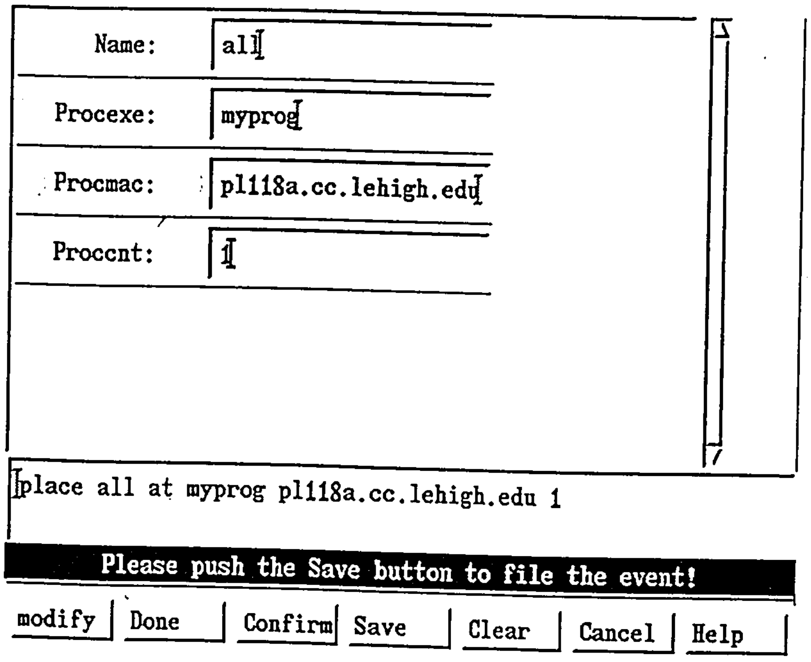
. Push **Confirm** button. The information display box shows "all at myprog pl118a.cc.lehigh.edu 1" next to the "place".

. Push **Save** button. "place all at myprog pl118a.cc.lehigh.edu 1" is save into the same file defined for the first event.

. The place event definition form is unmanaged. Another event can be defined or the place event can be defined again.

. Use the *event checking form* to check the defined event. Details see section 5) below.

The completed this event definition is shown in Figure 21.



Name:	all
Procexe:	myprog
Procmac:	pl118a.cc.lehigh.edu
Proccnt:	1

```
place all at myprog pl118a.cc.lehigh.edu 1
```

```
Please push the Save button to file the event!
```

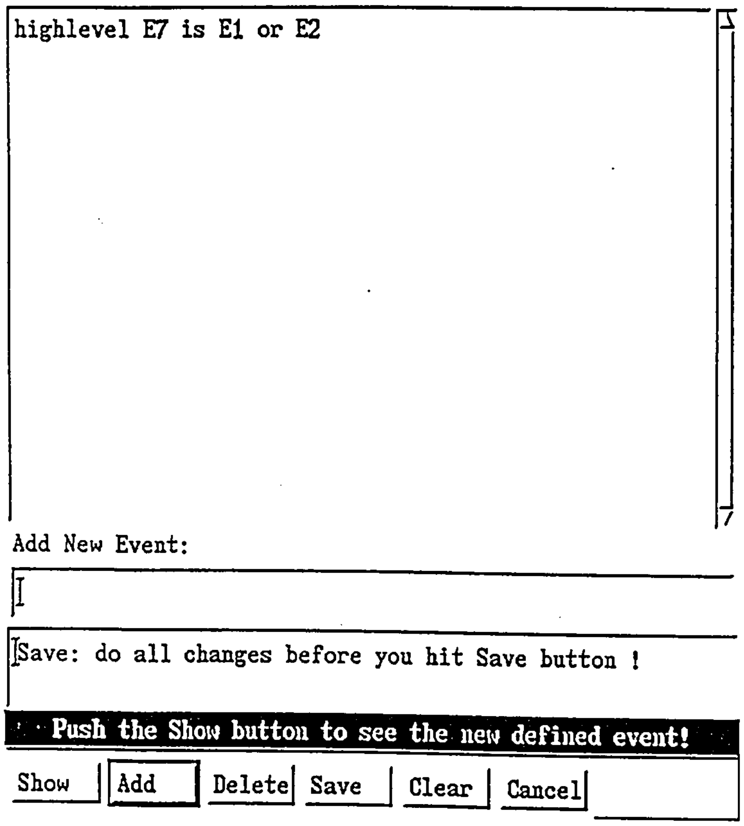
Figure 21. The Completed "place all at myprog pl118.cc.lehigh.edu 1" Definition

5) Checking User Defined Event and Event Script File

This tool provides several ways to check user defined event and event script file. By using *event checking form*, the **Output** menubar, or the **File** menubar to do this.

i) Immediate Checking Defined Event

Immediate defined event checking happens right after pushing the **Save** button on the *event definition form* (as shown in Figure 22). The *event checking form* is created by



```
highlevel E7 is E1 or E2

Add New Event:

Save: do all changes before you hit Save button !
Push the Show button to see the new defined event!
```

Figure 22. The Event Checking Form

event_define_display function, and is named as "List of Defined Events". The top of the form is the *scrolled list box* for displaying all the events defined. No items are available for a new list. The middle of the form is the *label box* called "Add New Event" and the *text field box* to show the event just entered. Below the middle of the form is the information display box and warning message box. The bottom of the form is six push buttons. They are: Show, Add, Delete, Save, Clear, and Cancel. The following shows how to do check defined event step by step.

. Push **Save** button on the first *event definition form.*

. The *event checking form* "List of Defined Events" pops up.

. Push **Show** button, the new entered event will display on the middle of the form.

. Push **Add** button, the new entered event will be added to the top of the form dynamically and disappears from the middle of the form.

. Repeat above steps. Another newly defined event can be checked after pushing the **Save** button on the *event definition form.*

. The user can use browse to select event in top list widget, selected item is highlight. Push **Delete** button, the selected item is deleted.

. **Clear** button is used to clear the top list box.

. **Cancel** button is used to unmanage this *event checking form.*

ii) Checking The Event Script File

The user may want to check all events defined. This tool supports two ways to do this. From the *event checking form*, the followings can be done:

. Push the **Save** button in the *event checking form.*

. All events are saved into "final.out" file.

. The *event checking form* is unmanaged.

Then one of the following ways can be used to check the user defined event scripts file. One way to check defined event scripts file by using the **Output** menubar:

. Push the menubar **Output** from main menu.

. Push the pulldown menu **Open** in menubar **Output**.

. All the events defined in the "final.out" file display on the main text box.

The other way to check defined event scripts file by using the **File** menubar:

. Select the menubar **File** from the main menu.

. Push the pulldown menu **Open** in the menubar **File**.

. A file selection dialog pops up.

. Select either the file named at first event definition form or "final.out".

. Click Ok button on the file selection dialog.

. The file selection dialog is unmanaged.

. The Contents of the selected file are displayed on the main text box.

6) Modifying User Defined Events and The Event Script File

This tool supports several ways to modify the user defined event and event script file. They can be modified on the *event definition form* stage, or on the *event checking form* stage. The event script file can also be modified after all events are defined using text editor on the **File** menubar.

i) Immediate Event Modifying

The *event definition form* has a **Modify** push button. It can be used to modify any event script parameters entered including label text enters and radio box selections. However this can only be done **before** pushing the **Done** button in the *event definition form*. Follow the steps shown below to modify the label enters:

. Push **Modify** button.

. Move the cursor to the label box that is to be modified and click on the text box on the right side of the label. The insertion point should be on the far right side of the text entered.

. BACKSPACE to delete all characters entered.

. Type in new enters.

. Repeat above steps until all modifications are completed.

. Press **Done** button.

. Press **Confirm** button.

Follow the steps shown below to modify the radio box selection:

. Push **Modify** button.

. Move the cursor to the radio box desired.

. Click on the radio box.

. The selected radio box is highlighted.

. Repeat above steps until all modifications are completed.

. Press **Done** button.

. Press **Confirm** button.

ii) Deleting The Defined Event

The *event checking form* provides event deleting capability to delete event after it has been defined. The push button **Delete** on this form can be used for this purpose. An individual event which has been added to the top list box can be deleted **before** pushing the **Save** button on this form. The steps below shows how to do this:

. Browse select item in the list box.

. The selected item is highlighted.

. Push **Delete** button on the *event checking form.*

. The selected item disappears.

. Repeat above steps needed to delete other items on this form.

. Push **Save** button.

iii) Modifying Defined Event Script File

The full-feature text editor provides complete modification of the defined event script file. The **File** menubar can be used to **Open** a file to modify, add, and delete the events. The user is familiar these steps:

. Select the **File** menubar in the main menu.

. Push the polldown menu **Open**.

. A *file selection dialog* pops up.

. Select the file name containing the events to be modified.

. Click **Ok** button on file selection dialog.

. The file selection dialog is unmanaged.

. The contents of the selected file are displayed on the main text box.

. Move the cursor to the place which is to be modified. The above outlined steps should be taken to do the followings (as desired):

. Modify events.

. Add new events.

. Delete events.

. Push the **Save** pulldown menu on the File menubar.

. The modified file is saved as an original file name.

. By pushing the **Save As** menu again, the file can be saved as a new file name also.

. A new file name dialog popups.

. Enter new file name.

18. References

1. Madalene Spezialetti, "Efficient Techniques for Monitoring Distributed Computation," National Science Foundation Project, 1993

2. Dan Heller, "Motif Programming Manual," O'Reilly & Associates, Inc., 1991.

3. Donald L. McMinds, "Mastering OSF/Motif Widgets," Addison-Wesly Publishing Company, Inc, 1993.

4. Marshall Brain, "Motif Programming: the essentials... and more," Digital Press, 1992.

5. Nabajyoti Barkakati, "Unix Desktop Guide to X/Motif," Hayden Books, 1991.

6. Douglas A. Young, "The X Window System Programming and Applications with Xt OSF/Motif Edition," Prentice-Hall, 1993.

7. Eric F. Johnson & Kevin Reichard, "Power Programming... MOTIF," Management Information Source, Inc., 1991.

18. Vita

Keyang Huang received the B.S. degree in Mechanical Engineering from Northern Jiao Tong University, Beijing, The People's Republic of China, the M.S. degree in Computer Science from Lehigh University, PA, USA, in 1994. He is currently a Ph.D. candidate of Computer Science, Lehigh University, PA, USA.

He has been a research engineer in the Computer Integrated Manufacturing laboratory (CIM Lab), Lehigh University, PA, USA, since 1988.

His research interests include distributed and parallel computing, multimedia systems, object oriented and visual programming.

END OF

TITLE