marble

AD-A267 159

A Command Editor Tool for X and Motif

Design Document
July 1, 1993

Sponsored by
Defense Advanced Research Projects Agency (DOD)
Defense Small Business Innovation Research Program
ARPA Order No. 5916

DTIC
ELECTE
JUL 2 8 1993
S A D

Issued by U.S. Army Missile Command
under Contract # **DAAH01-93-C-R013**
Effective Date: January 15, 1993
Expiration Date: July 15, 1993

prepared by
Patrick Dean Rusk, Minh Huynh, Greg Burd
Marble Associates, Inc.
38 Edge Hill Road
Waltham, MA 02154
(617) 891-5555

This document has been approved for public release and sale; its distribution is unlimited.

DISCLAIMER: The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

93 7 [illegible] 087

423910

93-16748

July 1, 1993

A Command Editor Tool for X and Motif

MARBLE ASSOCIATES, INC.

The X/Motif design paper prepared for DARPA under
Contract # DAAH01-93-C-R013.

Design Document

1.0 Executive Summary

Accesion For
NTIS CRA&I
DTIC TAB
Unannounced
Justification
By
Distribution /
Availability Codes
Dist | Avail and / or Special
A-1

DTIC QUALITY INSPECTED 8

Marble Associates, Inc., is engaged in the development of customizable menus in NEXTSTEP and the X windowing environment. In this document, we present the design and implementation path for realizing end-user customizable menus in X and Motif.

The differences between the NEXTSTEP and X development environments motivate an approach to bringing customizable menus to X and Motif that differs from our approach in NEXTSTEP. Initially, we intend to provide the basic functionality of customizing menus via modal dialog boxes, and we will investigate the feasibility of providing the same functionality through drag and drop (DND) in later stages of our implementation path.

We do not foresee any difficulties in the initial stage of our implementation. However, the success of our DND effort is contingent on the flexibility of Motif. Specifically, though "subclassing" widgets is a known process, we must determine any loss of functionality in the Motif widget set when we add functionalities to an existing widget through this "subclassing" mechanism.

This rest of this paper is organized as follows.

- Background—This section provides a brief discussion of our work in realizing the *Command Editor* for NEXTSTEP.
- Functional Specification—This section introduces the functionalities we intend to provide to the end-user and the interface we will present to the developer.
- Constraints—This section discusses the inner workings of the X window system and Motif. Specifically, it focuses on the challenges we face in realizing the functionalities presented in the previous section.
- Implementation Path—This section presents the process by which we can realize the desired functionalities in the face of system constraints. The process is divided into stages that represent the evolution of our product from a

minimal implementation based on Motif dialog boxes to one that employs DND.

- Questions and Impact—This section discusses the critical issues that will affect the implementation path.
- Related Documents—This section lists the other documents submitted under Contract # DAAH01-93-C-R013.
- Sources—This section lists the sources used in our research.

Many of the terms and concepts in this document originate from the NEXTSTEP implementation of the *Command Editor*. Subsequent discussions assume that the reader is familiar with the documents listed in the "Related Documents" section and has a general understanding of X and Motif. Despite the similarities in concepts and goals, the implementations differ vastly, and we will introduce new terminology and new names for this implementation to prevent any confusion and inappropriate analogies.

2.0 Background

Our Phase I proposal drew inspiration from a Microsoft Windows application that provided a customizable menu bar; we thus focus on bringing this functionality to NEXTSTEP, with the intention that we would do likewise for the X environment.

The rich toolset of the NEXTSTEP development environment facilitates efficient reuse of existing functionalities and quick prototyping of a graphical interface on behalf of the developer. Through Objective-C, the native language of NEXTSTEP, the developer can easily subclass complex objects and inherit their functionalities with a few lines of code. Through Interface Builder (IB), the developer can construct a graphical interface to his application by dragging and dropping objects from "palettes" onto a canvas. Through tools integrated into IB, the developer can integrate the interface with the application logic.

In addition to these conveniences, the NEXTSTEP development environment is extendable, providing a process through which tool builders can easily integrate customized objects into IB known as "paletting." This integration, in turn, provides new functionalities to the developer in a manner keeping with the IB interface construction mechanism: the user can drag and drop the customized objects onto his canvas as if they were default system objects.

The *Command Editor* for NEXTSTEP provides customizable menus to the developer through IB, making use of the convenient "paletting" process. We construct the *Command Editor* itself using the subclassing methodology to bring new functionalities to existing system classes while retaining inherited capabilities.

By contrast, the X development environment lacks many of these convenient mechanisms. Bringing new functionalities to an existing widget type in X, also termed "subclassing" a widget, entails rewriting that widget. This could encompass hundreds of lines of code and is necessary since the native language of X is C. Furthermore, X lacks a standard development methodology that would allow us to present the *Command Editor* in a packaged form. Any toolset for the X environment, typically consist-

ing of modified system widgets, provides to the developer the full source listing for that widget set.

The impact this has on our effort is two-fold. First, we cannot target the same level of functionality for the *Command Editor* here as we did for NEXTSTEP. Second, we can only provide the final product as complete source code. Though we will provide examples, the process of integrating our components into the application is left completely up to the developer.

3.0 Functional Specification

Our objective is to provide to the developer a mechanism by which he can integrate customizable menus into his Motif application. This mechanism allows the end-user to customize the application's menu system at runtime. The implementation path for the *Command Editor* divides our development effort into two stages defined by the functionality it brings to the user. We discuss the functional differences between the two stages from the perspectives of the user and developer in the following sections.

3.1 End-user Functionality

The functionality targeted in the initial development stage allows the end-user to do the following at runtime:

- rename, add, and remove menu items from a Motif menu curtain;
- specify accelerator keys and mnemonics for a menu item;
- add and populate submenus; and
- load Motif buttons with iconic representations of menu items.

A series of Motif dialog boxes will facilitate these functions. The user selects the "Menu" item from the "Customize" cascade button in the menu bar and brings forth the **CEMenuConfigurator** (Figure 1). This selection dialog box contains a hierarchical list of menu items representing the current configuration of the application menu system. The user then selects an entry from the browser and specifies its mnemonic and accelerator key. Entries designated as "default" by the developer are not customizable and will be "grayed out" by the **CEMenuConfigurator**. The user can also remove the menu item or add a new menu item under the selection. Selecting "Add" will activate the **CECommandBrowser**, a selection dialog box that displays a hierarchical listing of the full set of commands configured by the developer (Figure 2). The user can now peruse the set of commands available and select the additional menu item.

The user can similarly add a cascaded menu curtain by selecting the "Submenu" button, which activates a popup dialog box that contains a text entry field, allowing the user to enter the submenu title. The user can populate the submenu via the "Populate-Submenu" button.

FIGURE 1 The CEMenuConfigurator in Motif: An Approximation

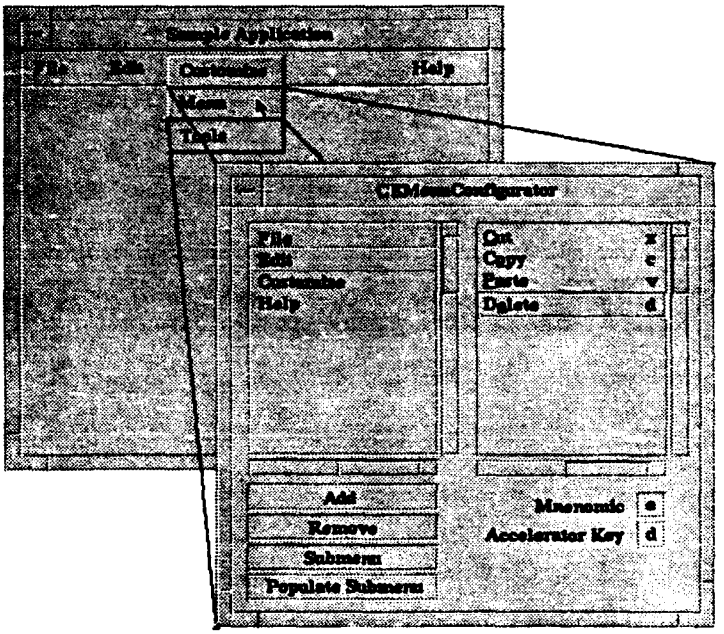
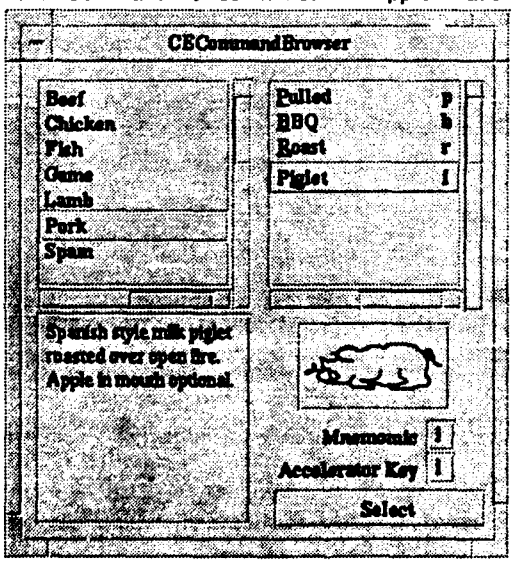


FIGURE 2 The CECommandBrowser in Motif: An Approximation



To configure the **CECommandToolButtons** provided by the developer, the user selects "Tools" from the "Customize" cascade button in the menu bar. This selection changes the cursor to a "Customize Tools" cursor, indicative of the special mode the

user has just entered (Figure 3). The user then selects a **CECommandToolButton** with the mouse and, in doing so, activates a dialog that allows the specification of a menu command that button is to trigger. The **CECommandToolButton** now displays the icon associated with the selected menu command.

The second stage of our development cycle will provide DND functionality to the end-user. The user can now configure the menu system by dragging a menu item from the **CECommandBrowser** onto the **CEMenuConfigurator** (Figure 4). The user will customize **CECommandToolButtons** in similar fashion—by dragging a menu item from the **CECommandBrowser** onto a **CECommandToolButton** (Figure 5).

FIGURE 3 Customization of CECommandToolButtons in Motif: An Approximation

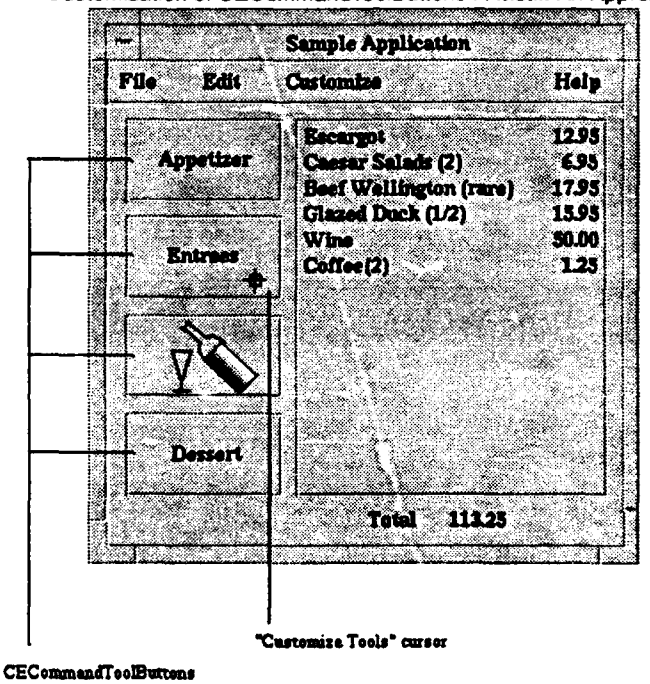


FIGURE 4 Customization of Menus in Motif via DND: An Approximation

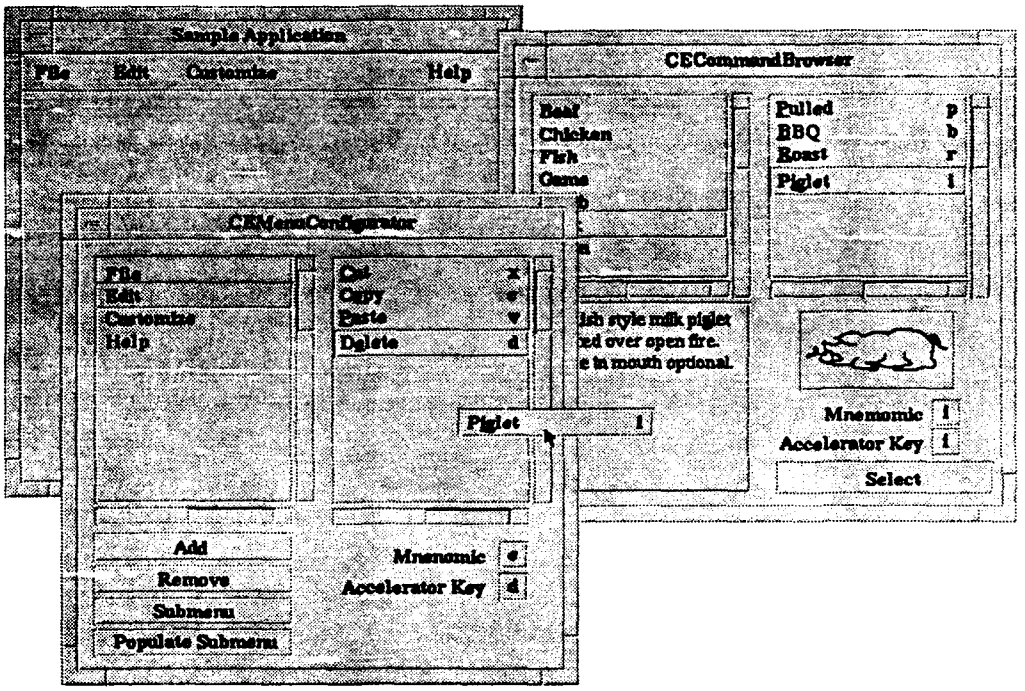
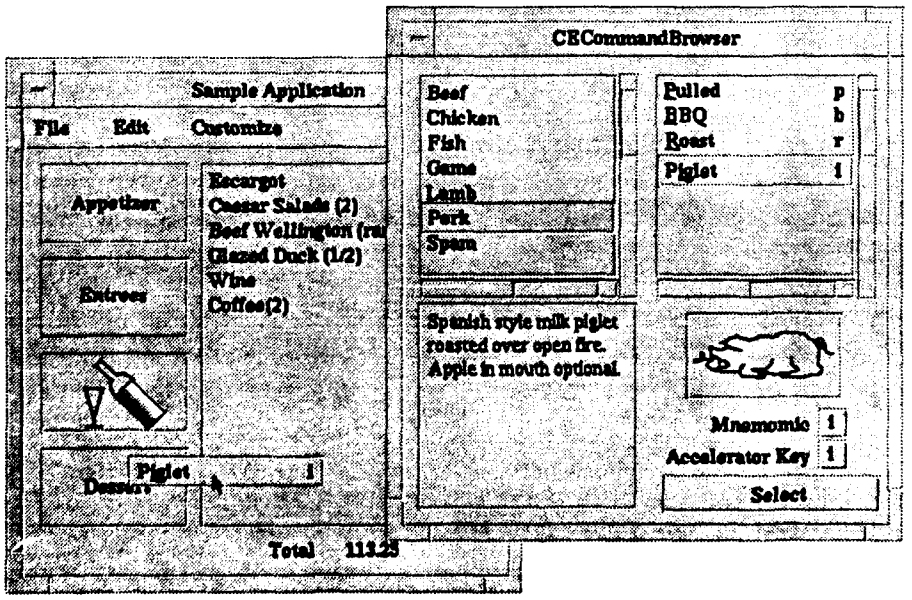


FIGURE 5 Customization of CECommandToolButtons in Motif via DND: An Approximation



3.2 Developer Interface

The *Command Editor* in X and Motif provides a framework in which the developer utilizes our components to facilitate customizable menus. This framework consists of source modules and guidelines that, together, dictate specific methods the developer must employ. This organization results directly from the lack of a standard development environment for the X and Motif community. We simply cannot provide the *Command Editor* as a custom palette in a particular interface building tool; doing so would require the developer to purchase additional software. Any loss of flexibility on behalf of the developer results from the constraints of the X window system and Motif.

In contrast, NEXTSTEP provides the developer with a uniform methodology for writing graphical applications. This methodology is the framework presented by Interface Builder, which is included in every installation of NEXTSTEP. This provision, in turn, allows tool developers to provide to application developers new functionalities. These new functionalities are packaged in a manner that facilitates ease of integration—through IB itself.

The **CECommandBrowser** and **CEMenuConfigurator** are special purpose Motif dialogs that the developer must integrate into his application code. Example code will be provided to illustrate this integration.

The developer must create menus through a prescribed interface. Though the resulting menus will be standard Motif Widgets, the configuration data must be maintained by a module in the *Command Editor*, and the standard Motif method of creating menus is no longer available to the developer should he desire customizable menus. The developer fills in a **CEMenuCell** structure with customizing parameters (default, configurable, icon, description, *etc.*) in addition to standard Motif information (title, accelerator key, mnemonic, and callback). This **CEMenuCell** can then be loaded into the *Command Editor* through an **CEAddMenuItem** call. Parameters to this call reveal the placement of the new **CEMenuCell** in the menu hierarchy. The developer employs this menu creation process to compose the full set of available commands as well as to configure the default application menu system.

Once the appropriate components are developed, we intend to provide a Motif tool that allows the developer to compose the hierarchical menu system visually. The developer can specify menu item order, titles, mnemonics, accelerator keys, and submenu topology. Callbacks and icons associated with each menu item will be stubbed out for the developer to fill in later. The resulting configuration will be formatted such that the developer can integrate the menus into the application. This facility will be used to configure both the application menu and the full set of commands available for customization.

The **CECommandToolButtons** necessitate a separate mechanism to allow the developer to place them anywhere in the application and to enable customization by the end-user at runtime. The developer creates a **CECommandToolButton** as a standard Motif PushButton. The developer configures **CECommandToolButtons** in the standard Motif way—setting the widget's resources explicitly to determine the label or pixmap displayed and the callback procedure that the button invokes. To capture the customization capability, the developer must register the resulting Motif widget with the *Command Editor.*

The second stage of our development will not modify the interface to the developer. Composition of the full set of commands available to the end-user and configuration of the initial menu system will adhere to the methods detailed in the previous discussions.

4.0 Constraints

From the developer's perspective, the X window system consists of two layers of libraries: Xlib and Xt. While the routines in the Xlib layer handle the low level duties required to display windows on a screen, the calls in Xt provide a measure of abstraction for the developer. Xt packages many of the sequences of Xlib calls that typify an X application into convenience functions. In addition, Xt provides Widgets (screen elements) such as buttons, scrollbars, and forms that are useful to a developer. Motif and Open Look are additional layers that enforce a particular interface style (the placement of menus, buttons, and scrollbars in an application) and furnish their own set of convenience functions to implement those styles. The motivation for conforming to a particular interface style is acceptance of the application by users of that style. An additional motivation for working above the Motif and Open Look layers is the availability of convenience functions that not only implement the particular style but also save the developer from reams of Xt code necessary to produce functionally equivalent visual elements (menus, text widgets, *etc.*). We focus our efforts at this top layer and will develop in Motif, since Motif is by far the most widely accepted interface style in the Unix community, undoubtedly outpacing Open Look in user acceptance and number of conforming products.

4.1 The X Development Environment

The lack of object-oriented technology in X (and consequently Motif) poses serious constraints on our development effort. The layered architecture of X, which affords it portability and remote display capability, is not abstracted from a developer's point of view. Indeed, Motif applications are allowed to make calls to all layers of the X environment. These applications are "Motif" only in look—using resources defined in the Motif libraries that define this look. Motif calls simply map to Xt calls.

This allowance violates the basic tenets of object-oriented development. The dependency between the application layer on all layers of the X and Motif environment inhibits reuse. For instance, X lacks a generic hierarchical display widget that allows the user to traverse and select items from a tree. Development of such a widget will be specific to the data type of the nodes. There is no concept of messaging objects in X. The lack of this capability dictates that the hierarchical display widget knows the type of the data node explicitly in order to (1) display the text label of the node and (2) traverse the tree.The Motif FileSelectionDialog widget, for instance, is closely tied to the Unix file system and cannot be customized to display any other hierarchy.

One key contributor to the lack of object-oriented design of X is the C programming language itself—through which X presents itself. Objects do not truly exist in the X environment in that they cannot be sent messages. The application simply links in library calls to the underlying layers. "Object instantiation" entails the processing of all library calls that allocate memory for the data structure and sets its resources. "Subclassing" a widget entails copying the widget's definition files and modifying its resource values. The term "subclassing a widget" is synonymous with rewriting a widget, where the developer must provide extensive code to do all of the following:

- modify the widget's resource list, effectively adding instance data;
- implement the initialization procedure to set instance data;
- implement the drawing procedures to resize and expose the screen element; and
- implement all widget specific actions that are performed given some user input.

This replication of code, even by include files, is not efficient reuse. Furthermore, the term "subclass" implies inheritance in object-oriented technologies, whereby subclassed objects obtain functionalities of their parent class. Copying definition files of X widgets allow the functionalities to replicated in any new widget. The availability of the same capability in Motif is unclear. For instance, the Motif implementation of menus is private, and subclassing the components of a Motif menu may compromise functionality. In short, subclassing does not necessarily imply inheritance in Motif.

By contrast, the object-oriented design of NEXTSTEP and even SmallTalk affords the developer the capability to augment existing system objects rapidly with inheritance of functionality. Many of these system objects correspond to widgets, providing a graphical interface and handling user input. In addition, subclassing objects and providing new methods entails little work in Objective-C (NEXTSTEP) and even less work in SmallTalk relative to the X environment. In summation, the realization that we can modify the development environment quickly and preserve existing functionality allows us to target a higher level of functionality in the *Command Editor* for NEXTSTEP.

4.2 Motif Menu Usage

A Motif application creates its menu system by creating a form widget and attaching it to the application window. This form widget is the application's menu bar. The application then creates individual CascadeButtons to populate the menu bar. A Motif PullDownMenu is then created and populated with individual PushButtons that represent menu cells. The PullDownMenu is then associated with a CascadeButton. Figure 6 illustrates the relationships among these widget classes. The code to implement a menu bar follows (Figure 7).

FIGURE 6 The Layout of Standard Motif Menus

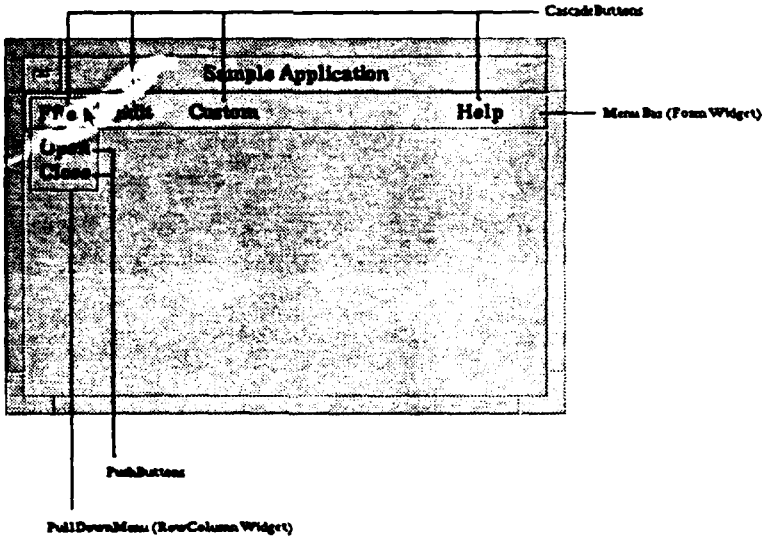


FIGURE 7 Using Motif Menus

```
void main(argc,argv)
int argc;
char *argv[];
{
        Arg al[10];
        int ac;

        Widget menu_bar, cascade, fileMenu, openPB, closeCB;

        /* create the toplevel shell */
        toplevel = XtAppInitialize(&context,"",NULL,0,&ac,
                                argv,NULL,NULL,0);

        ...
        /* create a form widget */
        ac=0;
        form=XmCreateForm(toplevel,"form",al,ac);
        XtManageChild(form);

        ...
        /* create the menu bar */
        ac=0;
        menu_bar=XmCreateMenuBar(form,"menu_bar",al,ac);
        XtManageChild(menu_bar);

        /* attach the menu bar to the form */
        ac=0;
        XtSetArg(al[ac],XmNtopAttachment,XmATTACH_FORM); ac++;
```

```
        XtSetArg(al[ac],XmNrightAttachment,XmATTACH_FORM); ac++;
        XtSetArg(al[ac],XmNleftAttachment,XmATTACH_FORM); ac++;
        XtSetValues(menu_bar,al,ac);
        ...
        /* create the CascadeButton */
        XtSetArg(al[ac],XmNlabelString,
                        XmStringCreateLtoR(menu_name,char_set)); ac++;
        cascade=XmCreateCascadeButton(menu_bar,menu_name,al,ac);
        XtManageChild(cascade);
        /* create the PulldownMenu */
        fileMenu=XmCreatePulldownMenu(menu_bar,"File",NULL,0);

        /* associate the PulldownMenu with the CascadeButton */
        XtVaSetValues (cascade, XmNsubMenuId, fileMenu, NULL);


        /* populate "File" PulldownMenu */
        ac = 0;
        XtSetArg(al[ac],XmNlabelString,
                        XmStringCreateLtoR("Open",char_set)); ac++;
        openPB=XmCreatePushButton(fileMenu,"Open",al,ac);
        XtManageChild(openPB);
        XtAddCallback(openPB,XmNactivateCallback,openCB,client_data);
        XtSetSensitive(openPB,True);
        ...
        ac = 0;
        XtSetArg(al[ac],XmNlabelString,
                        XmStringCreateLtoR("Close",char_set)); ac++;
        closePB=XmCreatePushButton(fileMenu,"Close",al,ac);
        XtManageChild(closePB);
        XtAddCallback(closePB,XmNactivateCallback,
                        closeCB,client_data);
        XtSetSensitive(closePB,True);
        ...
        XtRealizeWidget(toplevel);
        XtAppMainLoop(context);
}
```

Menu items (Open, Close) are displayed from top to bottom in the order they are created. In fact, the PulldownMenu widget is actually a Motif RowColumn widget with its resources set to act as a pulldown menu. While additional resources can specify the orientation (vertical or horizontal) of the PulldownMenu, child widgets are always ordered in the sequence they are created. No method is available to insert a menu item (PushButton) in a specific location. To allow the user to add and delete menu items at runtime, the *Command Editor* will destroy and recreate the entire PulldownMenu widget with new configuration data. Though this inefficiency is incurred only when the user customizes the menu system and hence will not affect the application's performance, we will nevertheless investigate alternatives to recreating the PulldownMenu widget.

Motif menus are modal and do not facilitate drag and drop customization. Whether Popup, PullDown, or PullRight, a "posted" (visible) Motif menu does not allow the user to interact with the application until the menu curtain is "unposted." The unposting occurs when (1) the user selects a menu item or (2) the user makes an invalid selection outside the menu curtain altogether. Motif does provide the TearOff menu, however, which remains posted even after the user makes a selection. TearOff menus

mimic the NEXTSTEP style of menus by remaining on the screen throughout the application's runtime.

We can accomplish drag and drop customization of Motif menus in three ways.

- *TearOff Menus* – This method requires the developer to designate all customizable menus as TearOff menus and similarly requires the user to tear off menu curtains before attempting customization. The user can drag a new menu item (or submenu item) directly onto the posted TearOff menu. Conversely, the user can <Control> drag to remove menu items. This functionality poses several challenges. The first is simply subclassing the Motif TearOff menu to add DND logic. The second challenge involves updating a modified TearOff menu to reflect additions and deletions. An additional challenge is the communication between the TearOff menu and its PulldownMenu representative in the application menu bar; the PulldownMenu must reflect any additions or deletions made to the TearOff menu.
- Posted Menus – This method allows the user to drag a menu item onto the menu bar. While continuing the drag, the user can activate a drag-sensitive cascade button in the menu bar, which pulls down its menu curtain. The user can then drag into the menu curtain, position the drag icon, and drop the menu item to be added. Once again, subclassing the Motif menu (now PulldownMenu) to add DND logic is key. The CascadeButtons will also need to be sensitive to DND.
- Special Purpose Shells – This method allows the user to drag a menu item onto a representation of the menu system. This representation, possibly implemented as a hierarchical browser, will detect the drop, calculate the mouse position relative to its displayed menu items, and add the menu item to the actual menu system. This method involves developing a hierarchical browser capable of DND.

Our design will adopt the last method, the motivations for which include reusability of the drag-initiating hierarchical browser developed to display the developer's full set of commands and simplicity of implementation. This browser is necessary in all three methods. The development of the DND hierarchical browser will itself entail widget subclassing and allow us to assess the feasibility of the other two methods.

4.3 Motif Menu Implementation

Our concern for the difficulty of the first two drag and drop methods described above stems from the opacity of Motif's implementation of menus. The following is an excerpt from Volume 6 of the O'Reilly and Associates "X Window System" series, Motif Programming Manual:

> "Motif does not use Xt's normal methods for creating and managing menus. In fact, you *cannot* use the standard Xt methods for popup menu creation or management without virtually reimplementing the Motif menu design...Instead, the Motif toolkit abstracts the menu creation and management process into generic routines that make the menu opaque to the programmer; the internal implementation is irrelevant."

This statement implies that we may not be able to subclass Motif menu components such as PullDownMenu and add functionality via Xt routines. For instance, subclassing the PullDownMenu and adding DND logic may prohibit the attachment of the altered PullDownMenu to the CascadeButton in the menu bar. The feasibility of implementing Motif menus with augmented widgets is unclear.

5.0 Implementation Path

5.1 Initial Development Stage

This stage of development targets a fully functional *Command Editor* package that allows the end-user to customize menus through modal dialog boxes. We begin by implementing the underlying logic of the package and conclude by developing the graphical tools for the end-user and developer.

5.1.1 Underlying Logic

The central module responsible for storing the developer's configuration of commands and processing the end-user's customizations is the **CEController**. This unit maintains two hierarchical structures: a list of commands composed by the developer and another list representing the application's current menu configuration. The **CEController** provides an interface through which these lists are composed, traversed, and manipulated. In this initial phase, the **CEController** also maintains the list of Command Tool Buttons available to the user for customization (Figure 8).

In the process of realizing the **CEController**, we must first define the atomic unit that represents a command. The **CEMenuCell** from our NEXTSTEP implementation will be used in this migration of the *Command Editor* to X and Motif, with slight modification. While the NEXTSTEP **CEMenuCell** embodied a "menu item" object, complete with PushButton functionality, the Motif equivalent will be implemented as a structure instead of a subclassed widget. Widgets are created relative to some parent widget and this relationship dictates the placement of the child widget on the screen. The **CEMenuCell** structure will contain information as to the title, mnemonic, accelerator key, drag icon, default status, reconfigurability, and activation callback of the command. The **CEController** will parse the **CEMenuCell** structure to create the appropriate PushButton in the application's menu structure.

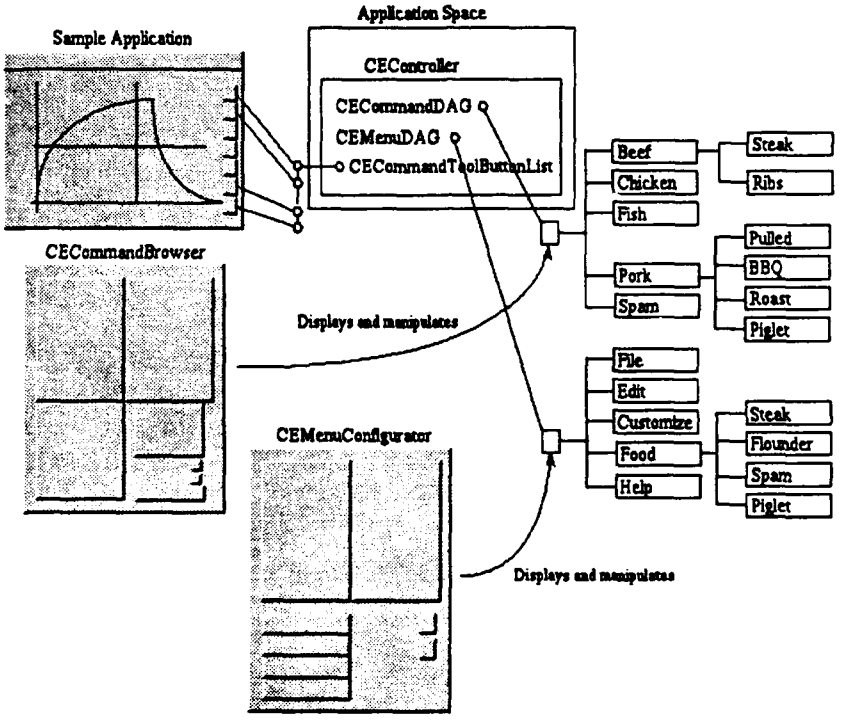
The hierarchy of **CEMenuCells** and the **CEMenuCells** themselves will be stored in a directed acyclic graph (DAG). There are two such lists: the **CECommandDAG**, which stores the developer's composition of the full set of commands available to the end-user, and the **CEMenuDAG**, which mirrors the current menu configuration. The **CEController** will provide the interfaces to its DAGs to the developer and other components of the *Command Editor*.

The **CEController** will provide the developer with interfaces to its **CECommandDAG** and **CEMenuDAG**. These interfaces make possible the composition and configuration of both DAGs without the **CECommandComposer** discussed in subsequent sections.

The **CECommandBrowser**, which allows the user to traverse and select from the full set of commands available, will use a subset of the developer's interface to the **CECommandDAG**.

Lastly, the **CEController** will provide to the **CEMenuConfigurator** an interface through which the application menu structure is customized at runtime. Each customization call in this interface will modify the **CEMenuDAG** and, in addition, make the appropriate Motif calls to put the customization into effect.

FIGURE 8 The Command Editor for X and Motif: The Underlying Logic



5.1.2 The Hierarchical Browser

Despite the plethora of widgets provided by X and Motif, we are unaware of any widget that facilitates the display and manipulation of a hierarchical structure. Ideally, such a widget would operate on a generic datatype—a tree of items—and only require that the tree implement a small set of functions, namely a display function and some traversal routines. Such a widget will allow graphical traversal of the tree and selection of tree nodes. The Motif FileSelectionDialog fulfills the desired functionality for a Unix file system. Whether this canned widget can be permuted to display and traverse a generic tree structure is unclear. We will investigate this possibility while actively searching the Internet and other sources for a generic hierarchical browser widget. If both efforts are unfruitful, we will develop the browser from a collection of Motif widgets.

Once the generic hierarchical browser is available, we will integrate the browser with the **CEController** and its DAGs. The browser will interact with the **CEController** through the interface defined for the DAGs to display and compose **CEMenuCells**. By the completion of this step, the generic browser has effectively been subclassed into the **CEMenuTreeBrowser**.

5.1.3 End-user Tools

The **CEMenuTreeBrowser** will be instantiated and incorporated into dialog boxes to provide configuration tools to the end-user. The **CECommandBrowser** dialog box allows the end-user to traverse and select from the full set of commands configured by the developer. The **CEMenuConfigurator** dialog box will display the current application menu structure and allow the user to modify this menu structure's configuration. The **CEMenuConfigurator**, in particular, will need to communicate user customizations to the **CEController**, which, in turn, will manipulate the application menu structure to reflect the user's modifications.

5.1.4 Developer Tools

The **CEMenuTreeBrowser** will be incorporated into a dialog that facilitates the developer's composition of the full set of commands available to the end-user at runtime. This dialog, the **CECommandComposer**, also allows the developer to configure the default menu structure that is initially displayed in the absence of user customization. The **CECommandComposer** will write the command hierarchy in a format palatable to the **CEController**. We will also develop an ingest procedure by which the **CEController** can (1) reconstruct the full set of commands at runtime and (2) reconstruct the application menu structure at runtime.

5.1.5 A Special Case

The **CECommandToolButton** is an interesting implementation departure for the *Command Editor*. While the menu system relies on the **CEController** to update its configuration and resources, the **CECommandToolButton** handles the customization itself. When the user selects the "Tools" item in the application "Customize" menu, the callback invoked will set all registered **CECommandToolButtons** to "Customize Tools" mode. The **CECommandToolButtons** will register a special callback to bring forth the **CECommandBrowser** when activated. The user then selects the target **CECommandToolButton** and the **CECommandBrowser** becomes visible. The user then selects the desired command and the targeted **CECommandToolButton** registers the new resource data unto itself. This mechanism is an elaboration of callback chaining. The only dependency on the **CEController** occurs in the initial stage of obtaining the full list of registered **CECommandToolButtons**. This is the motivation for the requirement that the developer register all **CECommandToolButtons** with the *Command Editor*.

5.2 Adding Drag and Drop

This stage of our development effort targets the DND capability of Motif by augmenting the components developed in the previous stage. To add the functionality discussed in Section 2.1, the following tasks need to be completed.

- The **CECommandBrowser** needs to be cognizant of drag starts and capable of transferring the correct data to the drop site. The OSF/Motif Programmer's Guide outlines the procedures necessary for dragging. Specifically, button translations will be added to allow the user to use a particular mouse button to initiate a drag. We also need to set the drag icon to indicate the type of data the drag represents. Finally, the drag start site needs to format and transfer the drag data to the drop site.
- The **CEMenuConfigurator** needs to process drag drops and communicate configuration changes to the **CEController**. To do this, the **CEMenuConfigurator** must register itself as a valid drop site for the **CEMenuCell** data type, register the operations it is capable of performing (COPY, LINK, MOVE), and implement the callbacks to process the drop. Likewise, the **CEMenuConfigurator** will process drag starts to allow the user to remove menu items from the hierarchy.
- The Command Tool Button will be implemented as a subclass of PushButton with additional methods that allow the user to drag a command onto the button and configure its icon and callback procedure via the subsequent drop. The procedures to implement a drop-sensitive PushButton are analogous to those for the **CEMenuConfigurator**. The **CECommandToolButton**, however, will not communicate its configuration changes to the **CEController**. Rather, it can simply modify its own icon and callback procedure. By the completion of this stage, the developer will no longer need to register each customizable **CECommandToolButton** with the **CEController**; he can simply instantiate the button and place it anywhere in the application.

6.0 Questions and Impact

This section relates the concerns raised in the section "Constraints" to the implementation path we envision. The critical issues, whose resolution will alter the course of our implementation, are

- the utility and difficulty of widget subclassing,
- the feasibility of customizing Motif menu components while retaining standard menu functionality,
- the availability of a hierarchical browser, and
- the ease of adding DND to a widget.

We have charted an implementation path that minimizes the impact of the aforementioned issues. The only assumptions necessary for the success of our development are the latter two issues: that a hierarchical browser will be available and that DND logic can be integrated into a widget with relative ease. The first two issues, should they be resolved favorably, will redirect our development effort to provide improved functionality to the user. In short, our implementation path prepares for the worst and targets the maximum functionality given that assumption.

Aside from the critical issues, a few considerations remain for later revisions of our package. We have purposefully omitted a few issues in the previous discussions that detract from the intent of our design. One such issue is the Motif PushButton's com-

plete set of resources—class resources as well as inherited resources. We have limited our discussions of customizing a menu item and **CECommandToolButton** to title, callbacks, icons, mnemonics, and accelerator keys. We could just as easily incorporate the full set PushButton resources. For instance, a PushButton that senses multiple clicks can be customized to perform a specific action on a double-click. A concern is raised on the behalf of the **CEController**, which affects the user's customizations by overriding an existing PushButton's resources. The **CEController** must now modify all resources to ensure that a menu item does not exhibit the mixed behavior indicative of two dissimilar resource sets.

Motif also allows PopUp menus. We have not determined what customizing functionality to provide for PopUp menus. Customizing PopUp menus would most likely be done via a representation of the PopUp menu with drag and drop.

We have not incorporated multiple states in this discussion of the *Command Editor*. Implementation of multiple states entails keeping a set of callbacks and titles with each **CEMenuCell**. This, as well as the previously mentioned features, can be added in later revisions of our package.

7.0 Related Documents

A Command Editor Tool for NEXTSTEP and X-Windows Systems (SBIR Phase 1 Proposal) for DARPA/OASB/SBIR, Submitted on July 1, 1992.

A Command Editor Tool for NEXTSTEP and X, Quarterly Status Report under Contract # DAAH01-93-C-R013, Submitted on April 26, 1993.

A Command Editor Tool for NEXTSTEP, Final Report under Contract # DAAH01-93-C-R013, July 1, 1993.

The Command Editor: A Manual for Users and Developers, under Contract # DAAH01-93-C-R013, July 1, 1993.

End-user Customizable Menus in the X Windowing Environment (SBIR Phase II Proposal) for Contract # DAAH01-93-C-R013, July 1, 1993.

8.0 Sources

Brain, Marshall MOTIF PROGRAMMING The Essentials and More Digital Press, 1992, ISBN 1-55558-089-0

Flanagan, David Programmer's Supplement for Release 5 of the X Windowing System O'Reilly & Associates, Inc. 1991, ISBN 0-937175-86-2

Heller, Dan Motif Programming Manual O'Reilly & Associates, Inc. 1992, ISBN 0-937175-70-6

Nye, Adrian Xlib Programming Manual O'Reilly & Associates, Inc. 1990, ISBN 1-56592-002-3

Nye, Adrian XToolkit Intrinsics Programming Manual O'Reilly & Associates, Inc. 1992, ISBN 1-56592-013-9

Nye, Adrian Motif Programming Manual O'Reilly & Associates, Inc. 1992, ISBN 0-937175-70-6

OSF/Motif Programmer's Guide, Release 1.2 Prentice Hall 1993, ISBN 0-13-643107-0