ITERATIVE DESIGN OF USER INTERFACES FOR AN OFFICE SYSTEM USING A UIMS

P. Constantopoulos, G. Drettakis, E. Petra,
M. Theodoridou and Y. Yeorgaroudakis

January 1988

RCC/CSI/TR/1988/010
Technical Report Series
Computer Science Institute, Research Center of Crete
P.O.Box 1385, Heraklio, Crete, 711-10 Greece
Tel.: +30.81.221171, Telex: 262389 CCI GR
E-mail – UUCP: *mcvax!ariadne!maria*

Iterative design of User Interfaces for an Office System using a UIMS.

P. Constantopoulos, G. Drettakis, E. Petra,
M. Theodoridou, Y. Yeorgaroudakis.

Institute of Computer Science,
Research Centre of Crete.

ABSTRACT

The design of the user interface for an office application using a UIMS is described. In particular we report on the iterative development and improvement of the user interface of MUSE, a multimedia document filing system, using ACHILLES, a UIMS designed to produce interfaces for office applications.

1. Introduction.

It is commonly accepted that, to the eyes of the user, a computer-based office system *is* its user interface. It is also a fact that a user interface should be tailored not only to the specific application but also to a particular environment or even individual user. Furthermore, producing a satisfactory user interface (UI) is really an iterative task including prototyping, testing and redesign stages [LeLo85]. The perfection and even feasibility of implementation is restricted by economic considerations. Indeed, each iteration is a tedious, time-consuming process, requiring high programming talent, let alone the initial design and implementation [Hull86].

An attempt to relieve this problem comes in the form of User Interface Management Systems (UIMS) which are recently the subject of several intensive research and development efforts (e.g. Sassafras [Hill87], MIKE [Olse87], Menulay [Buxt83]). UIMSs are meant to be productivity tools for the UI designer, by supporting the interactive specification and generation of user interfaces. The expected benefit from their use is not as much a cut in the initial effort for designing and implementing a user interface, but rather a drastic reduction in the effort involved in the subsequent iterations, in other words the "variable cost" of the process.

Very little has been attempted in using UIMSs for real world systems production. Applications to date are usually in specialised areas such as cartography [Wong82] or CAD/CAM [Kasi87]. Collecting experience from the use of UIMSs in real systems, especially in an office environment, is important first to establish that the UIMS approach is viable, and then to assess the effectiveness of a UIMS and to improve it.

In this paper we present a very first experiment in using a UIMS to generate and then refine the user interface of an advanced, existing system. The UIMS, called *ACHILLES* (A Computer-Human Interface Lexical Library and Extensible Syntax), has been developed, though not fully yet, at the Institute of Computer Science, Research Centre of Crete, in the context of ESPRIT project no. 82 (Intelligent Workstation - IWS) [Espr84, Espr86]. The guinea pig, on the other hand, is the user interface of MUSE [Gibb87, Yeor87], a prototype multimedia document filing system, also developed at the Institute of Computer Science in close connection with ESPRIT project no. 28 (Multimedia Office Server - MULTOS) [Mult86] and in conformance with requirements identified by the GR-Offices project of the Science for Stability Programme. This specific choice was made for two main reasons:

(a) The user interface of MUSE was developed locally, thus enabling the comparison of the original development process with that using the UIMS.

(b) By its very nature, MUSE is very demanding on the user interface which should further be tailored to particular applications. Therefore the design of a good user interface for MUSE actually involves several iterations.

In the next section we present the ACHILLES UIMS, while in section 3 we discuss its use for building user interfaces. In section 4 we briefly describe the MUSE system and its user interface. In section 5 we describe how ACHILLES was used to replicate and then to improve the user interface of MUSE, demonstrating the feasibility as well as some of the benefits of the approach. Conclusions are drawn in section 6.

2. The UIMS "ACHILLES".

A *User Interface Management System (UIMS)* is a system for specifying and producing User Interfaces. It serves to separate the design and specification of the user-application interaction from the design and specification of the application software itself [Kasi82].

By simplifying the initial design and modification processes, and by automating the implementation phase, a UIMS supports *consistency* in interface design, produces *high quality* and *reliable* interfaces, and reduces the cost of design and implementation significantly [Hull85, Hull86, Sigr87].

ACHILLES is a UIMS developed with the following goals: isolation of the design of the user interface from the development of the application, integration of a variety of media in the interface (graphics, text, voice); concurrent operation of I/O devices with the UI; and provision for easy modification of the environment in terms of I/O devices and communication media [Espr86, Brow86].

ACHILLES is an *external control* UIMS, in which the implementation of the user interface as developed with the UIMS, calls application functions in response to user commands.

The interaction dialogue between the user and the application is viewed as a language. The model, as in most User Interface Management Systems, consists of three primary levels [Gree85]:

• The *semantic* level, embedding the semantic interpretation of the interface in the application code.
• The *syntactic* level, enabling the UI designer to define the human-computer dialogue. At this level response to user actions is defined.
• The *lexical* level, consisting of the input recognized by the UI and the output. The run-time support of ACHILLES provides the lexical component of the UI.

2.1. The lexical level

The lexical level is responsible for handling the input and output for all the available I/O devices. These are separated into *channels* that accept a specified set of input events. Conceptually, a channel is a collection of lexical objects that are related by sharing a common I/O device or virtual device such as a window. In ACHILLES four channels are available [Espr86].

The first channel is the window. The window channel is a window on a bitmapped screen in which graphical objects including boxes, circles, lines, menus and boxes containing text may be displayed. The window channel supports two types of input events. Those related to objects with a screen image (such as menu selection), and those related to other input devices such as keyboard and mouse.

The other channels that ACHILLES will provide are the voice input and voice output channels, and a separate channel for high quality electronic document presentation.

2.2. The syntactic level

The syntactic level of the UI, is written as a set of rules, where each rule has the form:

<event> => <response>

The UI description is written at this level as a set of rules. Each rule is a statement of how the UI will respond to some event. Therefore the effect of a rule is called its *response*. When the designer decides how he wants the UI to respond to some event, he writes a rule for that response, which is a set of sequentially executed statements, and adds it to the syntactic level. The UI description is made entirely with such rules.

A simple example of a response rule would look as follows:

```
mousebutton1:-
        ! # pop_up [ object : #first_menu ]
```

The first step in writing such rules is to declare what event this rule will respond to. This is done by writing the event/message name followed by the two characters ":-". The event "mousebutton1" invokes the response in the example above.

Four types of activities may be contained in a response to an event.

(a) Output events may be *sent* as requests to the lexical level, such as the '! # **pop_up**' message in the example above.

(b) Application function calls may be inserted in a response. Parameters and return values implement the communication between the syntactic level and the application.

(c) Local operations can be performed on variables adding flexibility to the language.

(d) *Artificial messages* can be sent to rule groups providing a method to overcome the limitations of the event based syntax. The statement '@ # **art**' will cause the response to the rule '**art:-**' to be executed, as if the event **art** had come from the input.

Furthermore, the response grammar allows the user to activate and deactivate rules at run-time by collecting them into *rule groups*. When a rule group is deactivated, it is ignored by the UI. Once activated, the rules belonging to a rule group will respond normally.

2.3. The user interface of ACHILLES.

The ACHILLES User Interface has been written in the ACHILLES UI specification language itself. Some functions, such as file or directory reading, have been implemented in Lisp or C.

The interface provides a *control window* (see Figure 1.). This window automatically loads the UI description files and the Lisp application files representing them as icons. The user may select the icons and either edit them with a window running an editor on the file, or load them into the Lisp environment. To make a change in the UI description all that is needed is to modify the UI description file and then load it into the environment.

ACHILLES also allows the designer to view the rule groups that are currently active and follow the flow of control through rule groups. This is done using the *tree window* shown in Figure 1.

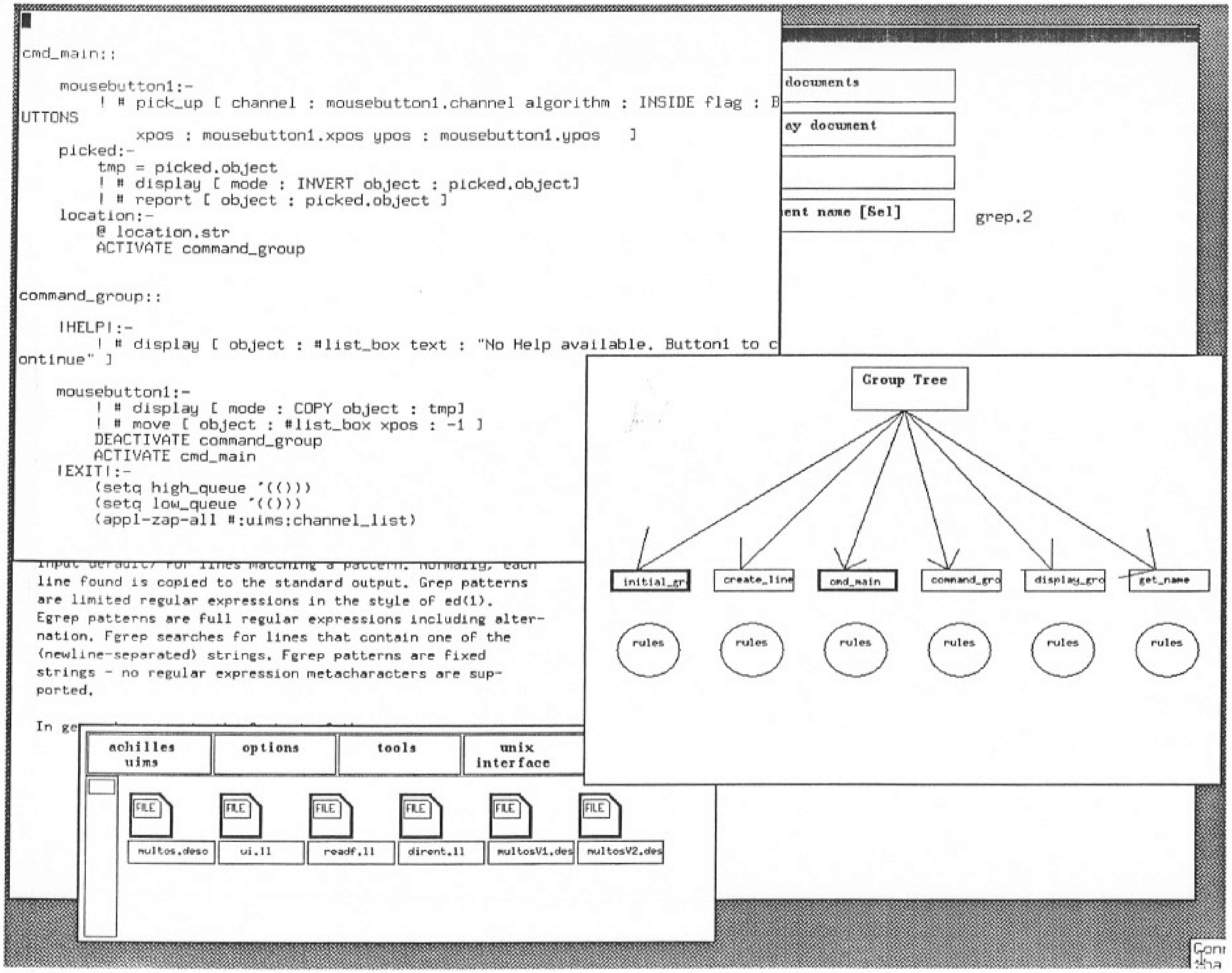
One of the enhancements planned for the immediate future is to allow the UI designer to select groups and rules in the *tree window* facilitating separate editing and loading of the rules and groups. This will speed up the process of UI definition significantly.

3. Building interfaces with the use of ACHILLES.

ACHILLES provides the UI designer with a set of tools that support the design process of human computer dialogues. ACHILLES in its present state produces interfaces easily adaptable and extendible, simplifying the use of direct manipulation methods and iconic interfaces. The high level lexical library relieves the UI designer from device dependent details, and the code reusability facilitates the use of the same UI description across applications. With the use of the LeLisp environment and the *suspended time editing* technique described below, iterative UI design becomes an attractive process.

3.1. Adaptability and extensibility.

The ACHILLES environment allows tailoring of the UI to meet specific needs. The UI of the same application may have to be different depending on the end user. Traditional methods would require recompilation of the whole application and, quite possibly, modifications to the application code itself, as the limit between UI code and application routines is often hazy. By contrast, the adaptation of the UI to meet differences in end user needs is simple to implement with ACHILLES. All that is required is to modify the UI description file pertaining to the specific portion of the UI. This allows easy modifications, as well as fast testing and evaluation of various alternatives. Equally important is the capability of the system to adapt as user expertise grows with time. A larger selection of choices



```
cmd_main::

    mousebutton1:-
        ! # pick_up [ channel : mousebutton1.channel algorithm : INSIDE flag : B
UTTONS
            xpos : mousebutton1.xpos ypos : mousebutton1.ypos   ]
    picked:-
        tmp = picked.object
        ! # display [ mode : INVERT object : picked.object]
        ! # report [ object : picked.object ]
    location:-
        @ location.str
        ACTIVATE command_group


command_group::

    |HELP|:-
        ! # display [ object : #list_box text : "No Help available. Button1 to c
ontinue" ]

    mousebutton1:-
        ! # display [ mode : COPY object : tmp]
        ! # move [ object : #list_box xpos : -1 ]
        DEACTIVATE command_group
        ACTIVATE cmd_main
    |EXIT|:-
        (setq high_queue '(()))
        (setq low_queue '(()))
        (appl-zap-all #:uims:channel_list)
```

```
line found is copied to the standard output. Grep patterns
are limited regular expressions in the style of ed(1).
Egrep patterns are full regular expressions including alter-
nation. Fgrep searches for lines that contain one of the
(newline-separated) strings. Fgrep patterns are fixed
strings - no regular expression metacharacters are sup-
ported.

In ge
```

Figure 1. ACHILLES User Interface.

should be provided to the expert user [Fole84]. Incorporating UI description code to support such a capability is simple with the use of ACHILLES.

The demand for modifications to the interface as the system is used is also comprehensively supported. Traditional methods make modifications to a graphical interface for a high resolution workstation a tedious task. In many cases the modifications required by end users are simple matters of *syntactic* nature as far as the UI is concerned. Such *syntactic* changes are the modification of responses depending on events, feedback, reporting facilities etc. These changes do not affect the semantic level, i.e. the meaning of a dialogue defined by the application itself, and can be made without modifying the application code.

3.2. High level lexical support and code reusability.

A lexical library is provided that handles all input and output, in conjunction with the runtime support module. Interaction techniques such as menus, scroll-bars and picking are provided at this level. This shields the UI designer from machine, device or system specific details such as interrupt handling or complicated graphics library invocations. The UI designer simply sends requests to the lexical level of ACHILLES to perform one of the complex interactions. The response to the messages/events returned from the lexical level must then be prescribed.

The system allows code, which implements a specific interaction, to be used in various applications. This convenience has the additional benefit of producing interfaces that look the same across different programs, enhancing consistency and relieving the user from the frustration due the proliferation of incompatible interfaces.

3.3. Direct manipulation, suspended time editing and iterative design.

The wide range of graphical objects supported by ACHILLES, and the fact that dragging and picking are supported orthogonally for all objects, allow extensive use of direct manipulation. The importance of direct manipulation interfaces is today generally accepted and has already been noted in many cases [Schn83].

The time spent on the initial design and implementation phase of a User Interface with ACHILLES may not actually be less than that using traditional methods. This feature has shown to be typical of UIMSs [Hill87]. On the other hand the repeated modification, evaluation and testing of a User Interface becomes much less tedious than with other approaches. Major changes to screen layout and object positioning can be made in short periods of time, by changing a few commands. Syntactic specification is equally easy to change, by modifying the active rule groups in a context or the responses to appropriate events.

Another important feature of ACHILLES is that it allows the UI designer to suspend the application, make a modification to the UI description and resume the execution of the process at the point of suspension. This method of UI construction is known as *suspended time editing* [Tann84, Hill87].

The overall time period consumed to implement, test, evaluate and modify a UI is greatly reduced, making the development of such interfaces cheaper. The cumulative effect is that iterative design, previously prohibitively expensive, now becomes feasible resulting in better interfaces.

4. The user interface of MUSE.

In this section we briefly describe the multimedia filing system MUSE, its user interface and related design issues. In the next section we shall see how ACHILLES has been used to reproduce and then improve the existing user interface design.

4.1. MUSE overview.

MUSE is an experimental multimedia document filing system based on an open and distributed architecture [Gibb87]. MUSE documents are structured collections of attribute data (integers or fixed-length character strings), text data (arbitrary length character strings from extended ASCII character set with embedded font control sequences), image data (raster bitmaps, either color or black/white), graphic data (sequences of graphic primitives, such as vectors or circles), or audio data.

Documents in MUSE are structured according to a hierarchy of *document types*. A document type is described by a number of attributes. Subtypes inherit the attributes of their ancestors. The root type is called GENERIC and contains only the attribute BODY.

MUSE comprises three main modules. a) A number of independent document servers which store documents and process document queries. Server processes may run on one or more systems. b) A number of clients which issue document queries and display documents. Client processes may also run on more than one systems. c)A special module called Name Server which runs on one system only. The Name server offers to the user an abstract, uniform view of the data base, which in reality is distributed over several servers.

4.2. MUSE functions

A detailed description of MUSE can be found in [Gibb87, Yeor87]. The functions supported by MUSE can be grouped into four sets.

• Status Commands	These commands are used only for testing the network's behavior and observing the operational status of the servers. They are not expected to be used by the average user.
• Local Commands	These commands handle local documents i.e. documents that are stored in the private local workspace of the user and are always available to him. Every user has his own disk area and he is the only one that has all the rights on the information kept there.
• Query Commands	This mode includes the set of operations that are necessary to interface with the document server. It also includes commands for query formulation and browsing query replies.
• Utilities Commands	The latest version of MUSE interfaces certain independent systems which are necessary for work in the office. Such systems are a) the mail subsystem b) the memo subsystem that helps the user to edit, modify and display short notes and c) the Mikrotek-300A digital scanner.

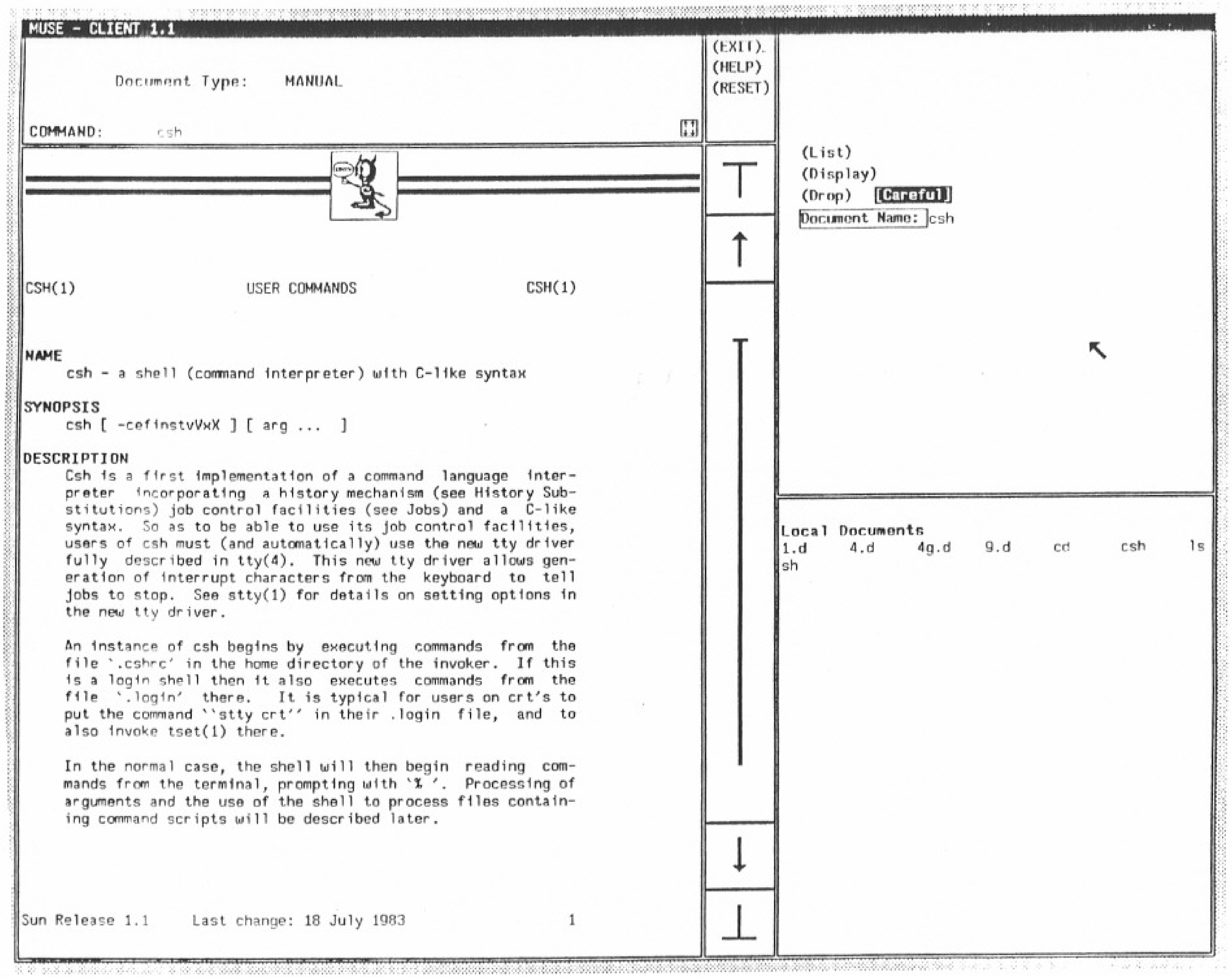
4.3. Initial design of the user interface of MUSE.

The user interface is unquestionably one of the most important components of an office system. As such, it cannot be considered apart from the rest of the system but should be designed in conjunction with it. However, it is quite difficult to ensure the suitability and effectiveness of a user interface unless it has been prototyped and tested. Successful user interfaces have gone through the design, implementation, testing and evaluation phases iteratively [Olse84].

The MUSE user interface has been developed on SUN-2 workstations. Each machine included a high resolution (1152 x 900) bitmapped black and white monitor and a three-button mouse. One workstation is also equipped with a color monitor of medium resolution (640 x 480 x 8). The user interface software was entirely written in C and extensive use of the tools provided by the SunWindows operating environment and the Sun window system was made.

The display screen in MUSE is divided into six subwindows (Figure 2), each one assigned to a specific task:

- The *attribute subwindow*, used to enter queries and to display document attributes. The attribute window can be enlarged or reduced by selecting the icon appearing in the lower right hand corner.
- The *exit subwindow*, used for a small set of commands which are always available. The help command is relative to the command mode.
- The *command subwindow*, used for command selection and parameter specification. Command selection is done with the use of the left mouse button while parameter specification is keyboard driven.
- The *display subwindow*, used for document display. Values which cannot be presented on the screen, are presented by small icons (e.g. voice). Users can operate on them by selecting the icon



MUSE - CLIENT 1.1

Document Type: MANUAL

COMMAND: csh

(EXIT)
(HELP)
(RESET)

(List)
(Display)
(Drop) [Careful]
Document Name: csh

Local Documents
1.d 4.d 4g.d 9.d cd csh ls
sh

```
CSH(1)                   USER COMMANDS                   CSH(1)



NAME
     csh - a shell (command interpreter) with C-like syntax

SYNOPSIS
     csh [ -cefinstvVxX ] [ arg ...  ]

DESCRIPTION
     Csh is a first implementation of a command  language  inter-
     preter  incorporating  a history mechanism (see History Sub-
     stitutions) job control facilities (see Jobs) and  a  C-like
     syntax.  So as to be able to use its job control facilities,
     users of csh must (and automatically) use the new tty driver
     fully  described in tty(4).  This new tty driver allows gen-
     eration of interrupt characters from the  keyboard  to  tell
     jobs to stop.  See stty(1) for details on setting options in
     the new tty driver.

     An instance of csh begins by  executing  commands  from  the
     file `.cshrc' in the home directory of the invoker.  If this
     is a login shell then it also  executes  commands  from  the
     file  `.login'  there.   It is typical for users on crt's to
     put the command ``stty crt'' in their .login  file,  and  to
     also invoke tset(1) there.

     In the normal case, the shell will then begin  reading  com-
     mands from the terminal, prompting with `% '.  Processing of
     arguments and the use of the shell to process files contain-
     ing command scripts will be described later.



Sun Release 1.1     Last change: 18 July 1983                 1
```

Figure 2. MUSE display screen.

with the mouse.

- The *scroll subwindow*, used for browsing through the pages of a document.
- The *message subwindow*, used for printing system responses, error messages, help messages and other information resulting from user actions.

The design and implementation of the user interface depends on the overall MUSE functionality. Although we wanted to separate the user interface from the implementation of the MUSE functions, this was not fully achieved. For example, the user interface routine that handles document presentation depends on the MUSE document structure.

4.4. Improving the user interface of MUSE.

The implementation of the MUSE user interface was completed by the end of 1986. The system was tested and a number of drawbacks regarding its user interface were identified. The most important enhancements that were pointed out as essential are the following:

- extension of the functionality of the display subwindow so as to support document editing as well as document presentation.
- adaptability of the user interface to the expertise of the user.
- minimization of the required keyboard input by allowing selections in windows other than the command subwindow (e.g. the message subwindow).
- extension of the browsing capabilities inside the pages of a document.
- display of help facilities and error messages where the user expects to see them.

The implementation of such enhancements is a difficult and time consuming task. For instance, porting the user interface on the SUN-3 workstation required one full week of implementation, just to adapt the UI to the SunView package used on the SUN-3s, without altering its functionality.

It is obvious that there is a need for an environment in which we could easily specify, design, implement, test and then iteratively redesign the MUSE user interface.

5. Designing the local mode interface of MUSE with ACHILLES: an iterative approach.

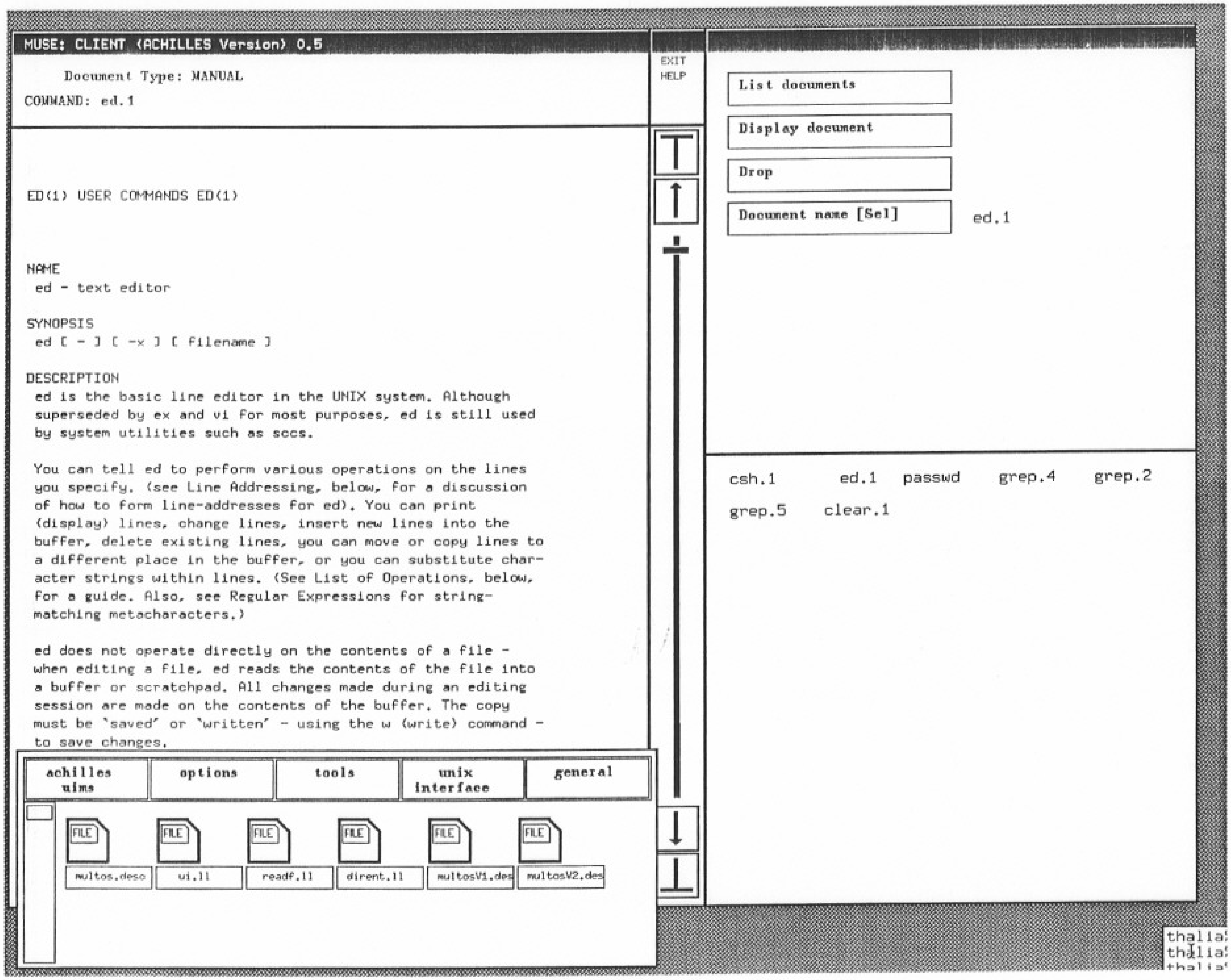
This section outlines one of the first applications of ACHILLES to a real interface other than its own. We describe the implementation of the local commands interface for MUSE using the ACHILLES UIMS. The initial interface of the MUSE local mode includes three commands: a) the **List** command which lists the names of the documents filed in the local workspace, b) the **Display** command which presents a document from the local workspace and c) the **Drop** command which deletes a document from the local workspace. The user specifies the document name by typing in the field **Document Name**. MUSE operating in local mode is shown in Fig. 2.

5.1. Replication of the existing interface.

Replicating the existing interface with the UIMS proved a comparatively simple task. The whole exercise took approximately one day to complete. The time necessary to implement such an interface is expected to drop to a few hours as the UIMS itself reaches a more stable state.

The result of this construction is shown in Fig. 3. The subwindows described in the section on MUSE have been implemented using *channels* for each subwindow. The actual ACHILLES code for this interface, excluding the initialisation statements is less than 200 lines. This is substantially less than that written in the original version, however one should be cautious in drawing general conclusions as only a section of the MUSE interface was implemented with ACHILLES.

The filenames appearing in the *message subwindow* are essentially text-boxes that contain the name of the file as a string. The 'buttons' in the *exit* and *command* subwindows are also implemented as selectable text-boxes. Each text-box is associated with the action described in the string it contains. The method in which the dialogue for these actions is written results in understandable UI description code.



```
ED(1) USER COMMANDS ED(1)


NAME
 ed - text editor

SYNOPSIS
 ed [ - ] [ -x ] [ filename ]

DESCRIPTION
 ed is the basic line editor in the UNIX system. Although
 superseded by ex and vi for most purposes, ed is still used
 by system utilities such as sccs.

 You can tell ed to perform various operations on the lines
 you specify. (see Line Addressing, below, for a discussion
 of how to form line-addresses for ed). You can print
 (display) lines, change lines, insert new lines into the
 buffer, delete existing lines, you can move or copy lines to
 a different place in the buffer, or you can substitute char-
 acter strings within lines. (See List of Operations, below,
 for a guide. Also, see Regular Expressions for string-
 matching metacharacters.)

 ed does not operate directly on the contents of a file -
 when editing a file, ed reads the contents of the file into
 a buffer or scratchpad. All changes made during an editing
 session are made on the contents of the buffer. The copy
 must be `saved' or `written' - using the w (write) command -
 to save changes.
```

Figure 3. MUSE interface with ACHILLES.

In MUSE the buttons in the *exit subwindow* have the label 'EXIT' and 'HELP'. To implement these two text-boxes marked with the flag 'BUTTONS' are displayed in the *command subwindow*. When the first mousebutton is clicked the following code is executed:

```
mousebutton1:-
      ! # pick_up [ flag : BUTTONS        ; Request a picked message if inside a button
                   algorithm : INSIDE ...]
picked:-                                   ; The button selected responds.
      ! #  report [                        ; Report causes a location
              object : picked.object ]     ; message that contains the boxes' string.
location:-
      @ location.str                       ; Send the string to the rule group
                                           ; The string will be either HELP or EXIT
HELP:-
      Action for help ...
EXIT:-
      Action for exit ...
```

The process of understanding and consequently debugging an interface described in this manner is thus greatly simplified. Locating an action connected to a 'button' is straightforward, as the corresponding rule has the name of the action itself.

5.2. Refining the interface.

The interface in its current state needs several improvements. Here we briefly discuss how these can be effected using ACHILLES. Also a first view of how interfaces are constructed using ACHILLES is provided.

In the original interface the 'Drop' command does not update the list of documents in the *message subwindow* to reflect the deletion. To correct this, two lines are all that is needed in ACHILLES

```
Drop:-
      Actions for drop...
      ACTIVATE list_group ; 'list_group' contains the rules
                          ; for listing the directory used as actions
                          ; to the 'List' event
      @ # start_list      ; and is initiated with this
                          ; artificial message
```

One apparent drawback of the initial UI is that the user is not permitted to select a name of a document in the *message subwindow*, and then to request some action on the specified file. This again is simple to do in ACHILLES. First a flag, eg 'FILEBOX', is defined that differentiates the boxes containing filenames from other boxes. After this has been done a simple rule group is written that handles mouse events for the channel *MSG_SUBW*.

```
select_file_group::

  mousebutton1:-
        ! # pick_up [ flag : FILEBOX ... ]           ; As above
  picked:-
        document_name = (get_name picked.object)         ; The boxes
                                                         ; have the filename for their name.
        ! # display [ object : picked.object             ; Display the box
                      mode : INVERT ]                    ; in reverse video
```

After such a document selection is made it is desirable to see the name selected next to the box containing 'Document name' in the *command subwindow* (see Figure 3.). This only takes adding a request to the response to the *picked* message above.

```
picked:-
        actions above ...
        ! # display [ object : #name_box
                    text : document_name ]
```

The modifications up to this point supply remedies to some very obvious pitfalls in the previous interface. The ease with which changes can be made encourages the testing of other alternatives for the same interface.

One thing obviously wrong in this interface is that the users' attention is continuously been diverted from one point of the screen to another [Fole84]. When selecting a document the users' attention is transferred to the *message subwindow*. After selection the user must revert his attention to the *command subwindow* to issue a command. A good interface should allow the user to perform the desired operations without ever having to leave the window in which the object to be acted on has been selected.

A solution to this problem can now be easily provided. The first action of the UI designer is to define an operation menu containing the two commands 'Drop' and 'Display'. After this has been accomplished the response to the rule *picked* is again modified, and the appropriate actions filled in.

```
picked:-
        actions as above ...
        ! # pop_up [ object : #op_menu        ; Pop up operations menu.
              xpos : cur_x ypos : cur_y ]     ; Cur_x and y have been set
                                              ; by the rule mousebutton1 to
                                              ; the current position of the mouse.
selection:-                                   ; Message produced by a menu selection.
        @ selection.choice                    ; Send the selected item.
drop:-
        ACTIVATE command_main                 ; Activate appropriate group
        DEACTIVATE select_file_group
        @ # Drop                              ; Initiate drop actions
Display:-
        ACTIVATE command_main
        DEACTIVATE select_file_group
        @ # Display
```

The *message subwindow* with its new capabilities can be seen in Fig. 4.

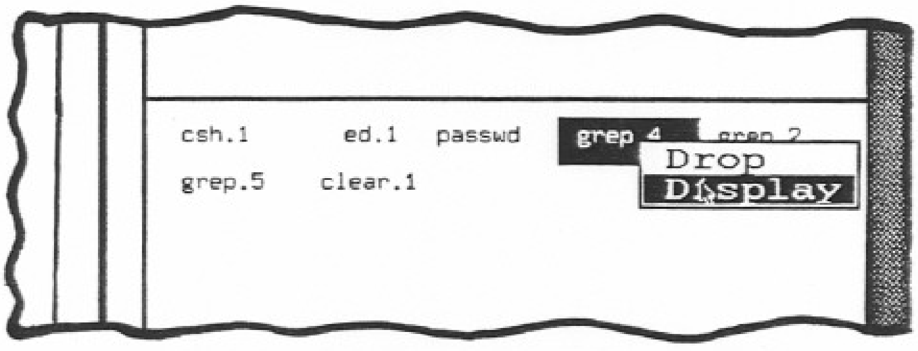


Figure 4. The message subwindow with the operations menu.

As a user becomes more familiar with the system, menus become tiring to use. One method widely used in existing systems such as the Apple Macintosh, and the Sun Workstations [Sun85], is the provision of *accelerators*. An *accelerator* allows a task that may be done with a complex set of actions to be carried out in a simple, but usually less informative (user friendly), manner. For instance, the tool manager in the SUN environment allows a window to be moved either by selecting the item *MOVE* in a menu or simply by clicking the border and releasing the mousebutton when the window is moved to the desired position. In our case an appealing accelerator could allow the two operations on documents, *Drop* and *Display* to be associated with two mousebuttons. This accelerator lets the user click with, e.g., mousebutton2 on a document name and have it displayed, and then click with mousebutton3 to have it dropped. The modifications to implement the accelerator should by now be obvious:

```
select_file_group::

        mousebutton:-          ; generic message for all mousebuttons
            ! # pick_up [ flag : FILEBOX ... ]        ; As above
        mousebutton1:-
            cmd = nil
        mousebutton2:-
            cmd = # Display
        mousebutton3:-
            cmd = # Drop
        picked:-
            Previously described actions...
            @ cmd              ; send the command
                               ; either Display or Drop

        Previously described rules for Drop, display and menu...
```

The process of refining the UI may continue endlessly. For instance it would be relatively easy to provide a Macintosh-like 'trashcan' for dropping documents as dragging and selecting are fully supported in ACHILLES. The limits of these modifications are simply the imagination of the UI designers.

6. Conclusions.

From the examples of the previous section it is obvious that an iterative approach to user interface design becomes feasible using ACHILLES. Some of the major advantages of the system are the following:

- Changes to the UI can be made quickly and simply.
- Interfaces can easily be adapted to changing needs and expertise.
- User-computer dialogues are described in an understandable way, allowing easier comprehension and maintenance of UIs.

Other capabilities of ACHILLES, such as suspended time editing, and reusability of its code make the design of a user interface more pleasant as well.

The experience presented here shows that ACHILLES is a powerful system for producing user interfaces. It displays promising capabilities that allow quick construction of clean and reliable interfaces. Future plans include the complete implementation of the entire user interface of MUSE, as well as of other interfaces in the IWS and other projects of the Institute of Computer Science, as ACHILLES evolves. Moreover, in what concerns MUSE, once the complete user interface is successfully implemented, we plan to take advantage of the ease of modifications provided by ACHILLES to follow on the recommendations of the evaluation of MUSE, thus demonstrating a complete, purposeful, iterative user interface design.

Acknowledgements

The ACHILLES UIMS owes its existence to the ideas of Ed Brown, and in part S. Hull. Substantial contribution to the implementation has been made by Dionissis Pnevmatikatos.

References

Brow86.
Brown, E., *ACHILLES: UIMS for an Intelligent Workstation - Development Specifications*, October 1986.

Buxt83.
Buxton W., Lamb, M.R., Sherman D., and Smith K.C.,, "Towards A comprehensive User Interface Management System," *SIGGRAPH 83 Conference Proceedings Comp. Graphics*, vol. 17, no. 3, 1983.

Espr84.
Naffah N., Kempen G., Rohmer J., Steels L., Tsichritzis D., and White G., *Intelligent Workstation in the Office: State of the Art and future perspectives*, November 1984.

Espr86.
Brown E., *Overview of a UIMS for an Intelligent Workstation*, November 1984.

Fole84.
Foley J., Wallace V.L., and Chan Peggy., "The Human Factors of Computer Graphics Interaction Techniques," *IEEE CG&A*, November 1984.

Gibb87.
S. Gibbs, D. Tsichritzis, A. Fitas, D. Konstantas, and Y. Yeorgaroudakis, "Muse: A multimedia filing system," *IEEE Software*, vol. 4, no. 2, pp. 4-15, March 1987.

Gree85.
Green, M., "The Design of Graphical User Interfaces.," *Technical Report, CSRI-170, University of Toronto*, 1985.

Hill87 .
Hill, R., "Supporting Cuncurrency, Communication and Synchronization in Human-Computer Interaction," *Thesis-University of Toronto*, 1987.

Hull85.
Hull, S., "A Study Of User Interface Management Systems," *Departement of Computer Science, University of Toronto,*, 1985.

Hull86.
Hull, S., "A Survey Of User Interface Management Systems," *Institute of Computer Science, Research Center of Crete,*, January 1986.

Kasi82.
Kasik David J., "A User Interface Management System," *Computer Graphics*, vol. 16, no. 3, July 1982.

Kasi87.
Kasik David et al., "Using a User Interface Management System.," *User Interface Management Systems, Siggraph '87, Course Notes*, Anaheim, California, July 27-31, 1987.

LeLo85.
A. Lee, and F. Lochovsky, "User Interface Design," *Office Automation*, Springer-Verlag, 1985.

Mult86.
P, Constantopoulos, Y. Yeorgaroudakis, M. Theodoridou, D. Konstantas, K. Kreplin, H. Eirund, A. Fitas, P. Savino, A. Converti, L. Martino, F. Rabbiti, E. Bertino, C. Thanos, and T. Beestra, "Office Document Retrieval In Multos," *Procs of Esprit Technical Week '86*, September 1986.

Olse84.
Olsen Dan et al., "A Context For User Interface Mangament," *IEEE CG&A*, December 1984.

Olse87.
Olsen Dan R., "Mike: The Menu Interaction Kontrol Environment," *User Interface Management Systems, Siggraph '87, Course Notes*, Anaheim, California, July 27-31, 1987.

Schn83.
Schneiderman Ben., "Direct Manipulation: A Step Beyond Programming Languages," *IEEE Computer*, , August 1983.

Sigr87.
"User Interface Management Systems," *Siggraph '87, Course Notes*, Anaheim, California, July 27-31, 1987.

Sun85 .
"Suntools(1) ," *Sun Workstation Commands Reference Manual, Sun Microsystems*, May 1985.

Tann84.
Tanner Peter and Buxton William., "Some Issues in Future User Interface Managment Development," *Seeheim Workshop on User Interfaces Managment Systems*, Springer-Verlag, 1984.

Wong82.
Wong P.C.S., and Reid E.R.,, "FLAIR-User Interface Dialog Design Tool," *Computer Graphics*, vol. 16, no. 3, pp. 87-106, July 1982.

Yeor87.
Y. Yeorgaroudakis, P. Constantopoulos, and M. Theodoridou, "Muse: A an experimental system for filing and retrieval of multimedia documents," *Information and Software Technology, (Submitted for publication)..*