USER INTERFACE DESIGN ENVIRONMENT

DTI/SERC Project: IED 4/1/1577

Information Engineering Advanced Technology Programme

Recursive HyperMenus
(The Hypermenu Command Interface)

Aaron Sloman

The University of Birmingham

UIDE.BHAM.T910.1.20/8/93

REPORT DATA SHEET

Report Reference	-	UIDE.BHAM.T910.1.20/8/93
Author	-	Aaron Sloman
Partners Involved	-	UoB
Date	-	Aug 20 1993

RECURSIVE HYPERMENUS

Aaron Sloman
The University of Birmingham

21 Aug 1993

CONTENTS

-- Introduction
-- -- NOTE on relevance to the UIDE project
-- The general problem
-- Types of option spaces
-- The need for a better theory
-- Purposes of an interface
-- The Hypermenu Command Interface
-- Assumptions about user requirements: activities, contexts, actions
-- -- Contexts
-- -- Action families and control panels
-- -- Direct and mediated actions
-- -- Action categories
-- Problems with menu-based action systems
-- -- Partial counter-example: TWM menus
-- How the recursive hypermenu system helps
-- Interface issues
-- The RHM solution
-- -- NOTES on the current implementation
-- Unintended side-benefit: an extendable menu interface to Unix
-- A menu notation for RHM
-- Format of menu definition list
-- -- Menu Label
-- -- Optional Screen Location
-- -- <Menu description>
-- -- <buttonlabel>
-- -- Format of <action> specifiers
-- -- -- A word:
-- -- -- A string:
-- -- -- A procedure
-- -- -- A vector containing a string

-- -- -- A list
-- -- -- 1. [ENTER <commandstring> {xloc yloc} <explanation>]
-- -- -- 2. [MENU <menuname>]
-- -- -- 3. [POP11 <pop11 code>]
-- -- -- 4. [UNIX <unix command> <shell specification>]
-- -- Additional special keywords
-- Unsolved problems
-- Acknowledgements
-- Incomplete bibliography
-- Trade Marks

-- Introduction

The Hypermenu Command Interface is a menu-based mechanism which combines a number of ideas from different sources, in particular (in no significant order):

Command menus
Control panels
Interaction/task contexts
Option spaces
Option space structures
Action families
User tailorability
Recursive data structures
Cyclic menus
Search lists
Autoloadable (dynamically linked) program libraries
Incremental compilation and re-compilation
Hypertext
Virtual memory
Garbage collection

-- -- NOTE on relevance to the UIDE project

Although this was not part of the original UIDE project plan, the idea arose when the author considered how best to provide a menu-based interface to the powerful new Poplog Graphical Object (GO) library developed as part of the UIDE project. It soon became clear that this was just an instance of a more general problem. Thus, although the paper is directly relevant to and arose out of the UIDE project, it addresses more general issues, and reports on a partial and tentative solution, for which a prototype implementation exists.

This could have been based on GO, which would have provided great flexibility as regards menu format and forms of interaction. However, since GO was still under development, and was not yet generally available, it was decided to use the existing “propsheet” library in the X subset of Poplog, which will make the system more widely available in the short run.

One disadvantage of the current implementation is that it works only with OpenLook. Extension to Motif would simply be a matter of digging into the Poplog PROPSHEET library (Meyer 1991) to extract and modify some of the code. However, an implementation based on GO would work equally well in both Motif and OpenLook implementations.

The system has already been tested by some students, colleagues and secretaries at Birmingham university, which led to considerable revisions. After further refinement it will probably be used by new students at Birmingham, starting October 1993, as a general interface to Poplog and its facilities.

-- The general problem

The general problem addressed by this package is to provide the user with help in driving a sophisticated computer system with many hundreds or thousands of possible commands, including commands for invoking utilities that have been produced by different people and have different combinations of arguments, flags and the like, so that there is no presumption that the total set of commands forms a well structured family.

Examples of such facilities include the complete (but constantly growing) collection of facilities provided on a modern multi-user computing system or workstation (e.g. a unix system), the facilities in a sophisticated software development environment such as Poplog, or the facilities in a powerful editor, such as Emacs or VED, which also provides access to other facilities, e.g. mail, news, printing.

Often each of the sub-facilities offers a variety of options and one has to learn to drive each of them separately. Some sub-facilities provide only a few commands, whereas others are large and sophisticated packages with very many commands.

In such systems the set of possible commands can number hundreds, or even thousands, especially if all the different combinations of flags and arguments are taken into account. This set of possibilities can be referred to as the “option space.” (Edmondson 1993 defines the notion of “selection space” which is very similar to this, though he constrains the space to be a pre-existing set of options. Since in many cases the user cannot tell whether the items do pre-exist or are created on demand, as sometimes happens with fonts, I prefer to use the more general concept.)

The task of finding out what is available and remembering how to drive all the different facilities can be daunting, imposing a considerable learning task and memory load.

Normally, for someone starting to use such a system there is documentation (on-line or off-line) pointing at various facilities and explaining how to use them (e.g. the Unix “man” files, Emacs “info”, Poplog “help” or “ref” files), though the documentation can be daunting for beginners and non-trivial even for experts to search rapidly. Even when there are many megabytes of information, finding out what facilities are available, or whether there is something that meets a current requirement can be a time-consuming process. (For instance, Poplog has about 10 Mbytes of on-line documentation.)

The “recursive hypermenu” (RHM) system described here is designed to provide a mechanism within the Poplog environment to help people learn about and develop fluency with facilities available in or via Poplog, including its editor. It can also be used to extend Poplog in various ways described below.

It is intended to provide a mechanism that can be used by people who design packages or libraries, to help new users drive those packages and become familiar with their facilities. In some cases RHM will provide a temporary interface, providing a path to learning about a more direct command interface. In other cases they may provide the only interface.

Because Poplog can itself invoke other Unix utilities RHM can be used as a mechanism to drive not only Poplog facilities, but also other facilities available on the system. For example, it is trivial to add a menu to invoke X facilities, such as xfontsel [Swick 1989], FrameMaker, a clock, or open a new xterm window.

Some of the features of RHM are already available in sophisticated window managers that allow users to define and modify the menu structure, for instance the "TWM" window manager [LaStrange 1989] provided with the MIT X11 distribution, and its predecessors and derivatives.

These, however, require the complete menu structure to be built into the running process, and every change, no matter how small, requires the complete structure to be rebuilt, which can be slow and cumbersome. Moreover, the "programming" language provided with such systems is generally very limited (it could not easily be used to implement an expert system, for example). More generally, a user-extendable menu system is not easy to provide without incremental compilation and automatic store management facilities of something like Poplog or a typical AI language.

The RHM system is not offered as an ideal or complete solution. Nevertheless it has a number of advantages over currently available alternatives, including simplicity, "softness" (easy tailorability by individual users or groups of users), sharing of subsets of menus between users, incremental modification whilst in use, and almost indefinite extensibility.

It makes use of X facilities now provided in Poplog together with several features of the Pop-11 sub-language, including list syntax, incremental compilation, autoloading and dynamically modifiable search lists. Automatic garbage collection is used to reclaim space when menus are recreated, for instance during development of a collection of menus.

In other respects RHM may be inferior to particular interfaces for particular applications. But as a general tool, it appears to be very useful. Mayhew(1993) writes:

> "Finally, because menus are such a rigid, structured, dialog style, they are best suited to tasks that are themselves highly structured, that is, in which a set sequence of steps is always or usually followed. Tasks that are highly unstructured, in which any sequence of individual steps or functions is possible and desirable, do not lend themselves well to a menu-driven interface, which forces the user into set sequences of steps."(p.120)

The RHM system is designed to show how this might be refuted.

-- Types of option spaces

The mechanism described here is not necessarily the ideal solution to all interface problems. It is not, for example, aimed specifically at a well structured space of possibilities defined by a number of orthogonal dimensions within each of which there is a fixed set of choices. For such a system a specially tailored interface might be far superior.

For example, for font selection RHM offers no advantages over the "xfontsel" utility provided as part of the MIT X11 system. The latter depends on the fact that the space of possible fonts has nearly orthogonal dimensions like foundry, family, weight, slant, pixelsize, and so on: by allowing users to make independent selections in different dimensions (and then propagating the effects of such selections to remove incompatible options from other dimensions) xfontsel provides a convenient way to bypass the usual awkward tree-structured menu mechanism that forces one to make decisions in a particular order, which often does not match one's needs or one's knowledge or the current task.

There are many other types of option spaces besides the N-dimensional space exemplified by fonts. For example there are trees, networks, and graphs, which may be cyclic or acyclic. The space may have some systematic structure, or it may be a largely unstructured collection of options. The total number of options may be small, or very large, or in the case of commands with a grammar it may be potentially infinite.

One common space of possibilities is a file system. Usually this is tree structured, so that finding a particular file requires a unique set of choices from the root of the tree down a path to the file. Some file systems, such as Unix, allow cross-links so that the same file or directory1 can be reached via more than one route, thereby reducing the memory load on the user (and the complexity of commands to access files that are commonly needed in different contexts.

Another type of option space is provided by a command language which has a well defined grammatical form. An interface to a calculator that allows the usual arithmetic expressions such as

(3+55)*88/45.732

is an example, with a potentially infinite set of possible commands. It would be pointless to try to provide a menu system to cover all the possible arithmetic expressions. On the other hand a menu of templates with facilities for combining them to produce more complex templates which could then be filled in with particular numbers could be a useful interface for people who did not wish to type in the expressions. An example might be a graphical device for building a parse tree to specify the desired computation.

Each space described above has a coherent structure deriving from its function. Usually when there is a simple and coherent structure a specific tool tailored to that structure will provide the best interface.

Some tools (e.g. dialogue boxes, expressions to be evaluated) allow the user to build up a description which then uniquely identifies the item to be selected (typically an action from a large or potentially infinite repertoire of possible actions).

Other tools (e.g. some bibliography interfaces) do not necessarily lead immediately to a unique selection, but support specification of constraints which prune the option space, allowing the user eventually to be presented with a very small sub-space that can be scanned quickly and a selection made explicitly. Depending on the structure of the space, different kinds of pruning are possible.

Some tools provide a coarse grained overview of the space which one can traverse in order to select a sub-space, which is itself treated the same way recursively, until the desired location is finally identified and activated or selected. Thus each choice prunes the space simply by eliminating paths to sub-spaces, unlike the kind of choice that prunes the space by removing some or all of a dimensions -- for instance choosing a point size in a font selector. One difference is that where a menu system is not tree structured but includes cycling networks, removing some routes may in fact not remove any nodes in the option space!

RHM is a tool of that kind: choices select routes and thereby eliminate other routes, but that may or may not eliminate nodes or subspaces, depending on the particular menu structure.

Note: RHM does not provide graphical support for navigation and route selection. It would not be difficult to add this, using the new GO library in Poplog, but so far resources required for this work have not been available, and it is not obvious in advance what the benefits would be and whether they would outweigh the costs of additional screen space for the selections mechanism rather than applications.In any case the layout of menus and the use of a persistent “pile” of menus, described below, is intended to reduce (though not necessarily to eliminate) the need for additional aids.

-- The need for a better theory

It would be helpful to have a complete overview of option space topologies, which then could be combined with a taxonomy of types of users and types of tasks to provide a basis for deciding which sort of tool or interface is best suited to a combination of option space, user and task.

This would require advances in cognitive science to provide the required analysis of users and tasks. Our work in Birmingham on architectures for intelligent agents indicates a need to distinguish all the following concurrent activities, all of which might require support in an interface:

o monitoring of current actions and the consequences

o monitoring of other “external” events (e.g. produced by other agents)

o generation of new goals

o evaluation and sequencing of goals

o selection or creation of plans

o internal or external execution of plans (including communication with external agents)

o triggering of internal or external “reflex” actions

o modification or expansion of plans (e.g. during execution)

o “meta-management” processes concerned with decisions about which higher level management (mental) tasks to perform when

o reasoning

o updating or accessing short term or long term memory

o learning of many kinds (new concepts, new facts, new skills, new heuristic short cuts, new ways of comparing priorities or importance of goals, etc.)

Designing interfaces on the basis of this sort of theory is currently beyond the state of the art, which is one reason why all interfaces require so much testing by real users and subsequent revision or refinement.

The RHM system provides a flexible tool that requires users and designers to address subsets of the problem according to their personal and changing evaluations of what is needed. It does not attempt to accommodate itself automatically to the user, as adaptive interfaces do, but it does allow the user (or an expert helping the user) to drive the adaptation, thus providing a way to meet the objectives of adaptive interfaces prior to the solution of all the long term research problems raised by attempting to automate the process.

-- Purposes of an interface

Many interfaces are designed to meet the following objectives:

(a) helping the user discover the structure and contents of the space of possibilities (this includes helping someone who is already familiar with part of the space to find out about new regions)

(b) helping the user who is already familiar with part of the space to access a desired item in the space, with minimal effort and delay.

Devices that serve (a) include paper manuals, on-line documentation, “help” commands that show which actions are available in the current context, menu interfaces, and other users.

Devices that serve (b) include function keys, commands based on key sequences (as in Emacs, and Framemaker accelerators), menus, “help” commands that show which actions are available in the current context, and, above all, command languages that allow the user who knows the name of a command to type it directly. There may or may not then be help available regarding the formats for use of the command.

RHM is aimed at both (a) and (b). It is designed to help users of system based on commands, (possibly with a lot of on-line documentation.) It can be used to help new users become familiar with a particular subset of the system, and when they know that region well and can drive it from commands and function keys (if desired) then an extended modified version of the set of hypermenus can be provided to help them gain familiarity with a new subregion of the option space.

This is not a solution to the total problem of making complex systems usable, but should help considerably in many cases. Full testing of that conjecture would require a lengthy and very complex experiment, with diverse types of users. This, unfortunately, will be infeasible in the foreseeable future, although feedback from real users has already influenced the design and will continue to do so.

-- The Hypermenu Command Interface

RHM is not designed to compete with special purpose interfaces to specialized command spaces. Rather it is a tool for coping with more messy collections of options, such as Unix, or Poplog. It does not offer a single general solutions for all users of the system, but provides a menu-based mechanism that makes it easy to help users with the particular subsets of commands that they need to use frequently, which will be different for different users.

The assumption is that for each user there will typically be a relatively small number (e.g. between about 10 and 20) subsets of commands that need to be readily accessible, and that each subset can be usefully accessed via a menu system that does not go very deep and is tailored to meet the preferences of the individual user. The menu system will typically help the user to find out which options are available in the chosen subsets and gradually learn to bypass them by using the more direct command level or function key interface.

When that happens the user will often be learning about new sets of facilities available within the system. (That is true of most but not all Poplog users -- some learn a small subset and stick with them forever.)

When learning a new subset, the user can (often with help from a library package designed for the purpose, or help from expert users) replace old menus that have become unnecessary (because more direct commands have been learnt) with a new set of options.

Changes made during a login session can be stored in the user's private menu library and will be invoked automatically in future.

-- Assumptions about user requirements: activities, contexts, actions

The user is assumed to be interacting with a system that supports:

o a variety of types of activities

E.g. browsing documentation, creating text, testing programs, reading mail, sending mail, etc.

o for each activity a variety of contexts

e.g. switching from one file to another, or switching modes

o for each context a variety of actions that can be performed in that context, some of which may be parametrized.

-- -- Contexts

The notion of context is not easy to define precisely. To a first approximation a context is defined by:

o an object with which the user interacts, and

o a family of actions available for interacting with that object.

The context may also include a number of state variables that can be used to control the actions or the behaviour of the object. A sub-context would be defined by a portion of the object and/or a subset of the actions.

Thus the text creation activity has different contexts depending on which file you are working on. Some files may provide action sub-contexts, e.g. checking spelling, searching, typing input, producing or modifying a diagram, etc.

Actions available in different contexts and different activities may overlap. For example the action of searching for a text string is relevant to actions as different in their objectives as browsing through on-line documentation, examining the output of test runs of programs, developing code, developing documentation, reading mail, etc. It is partly because of a desire to use the same tool in all such contexts that many users of Emacs, and VED, prefer to use the editor as a general interface to all other facilties.

Similarly there may be families of actions that are equally relevant in a lot of different contexts. For example “cursor move” options (next page, previous page, next line, previous line, word right, etc.) form a family relevant to many different contexts.

(Some of this “sharing” between contexts is what makes the statechart formalism attractive.)

-- -- Action families and control panels

I assume that it is useful to group the actions available in an activity into relatively small *families* of actions, where each family can be presented in a single menu, which is a sort of control panel for getting things done by choosing an item, i.e. pressing a button.

Presenting a group of actions together helps to remind users of what is possible partly because the juxtaposition of related options supports mutual disambiguation of potentially ambiguous short labels. This idea is already used in systems like FrameMaker, where different clusters of options (paragraph formats, character formats, graphical tools, etc.) are offered in different selection panels, where each panel presents a related set of facilities, often with switchable sub-panels.

However, for a very large and complex system such as Poplog, or Unix, there is no uniquely correct way of grouping facilities into action packets. So a requirement for an acceptable system is that it should be easy to provide different ways of grouping for different people. This requirement for tailorability is met by RHM, because of the role of search lists in locating menus dynamically.

-- -- Direct and mediated actions

Some buttons in a menu, or control panel, will immediately cause actions, whereas others will invoke a dialogue box to get a bit more information from the user (e.g. file name, search string, or other parameters), or allow constraints to be set, before the action is done.

-- -- Action categories

Actions to be supported by control panels can be categorized in various ways.Eg

o do something in the current context:
E.g. insert a line, delete a character, compile a file

o specify an action, where the action admits variants:
E.g. a dialogue box could prompt for more detailed specification, such as the search and replace dialogue box in RHM which allows the user to choose whether the searching should included embedded strings, whether the it should be interactive or automatic, and whether it should have as its scope the whole file, a marked range, the current procedure (or paragraph) or the current line.

o modify the current context:
E.g. toggle line-break mode

o switch to another context or sub-context:
E.g. go from writing code to reading mail, or from editing to interacting with a spelling checker/corrector, or from one file to another.

o get help/information:
There are many different kinds of help, including meta-help about the help facilities, or help about the menu system, or help on a particular button, as well as help about the facilities the user is ultimately interested in.

o quit current context:
This can either take you back to a previous context, or leave you in a neutral state where you have to decide what to do next. Quitting can either be back to a previous state, or to the top of some previous intermediate subnet, or to the top level of the whole system, or right out of the whole system. RHM provides mechanisms to support all of these.

A good command system should support all of these action types (and more). sIn principle, RHM allows them all to be invoked via menu buttons or dialogues in the context of any package implemented in Poplog, as long as the package itself supports the required action type.

-- Problems with menu-based action systems

Menus are often a useful device for people who cannot remember available commands: They aid recognition, which is generally much easier than recall. However, there are well known problems with using a menu interface to gain access to a very large number of possible actions.

Many menu systems require a menu tree that (a) must be built at compile time, (b) is hard to change while the program is running, (c) cannot be recursive: i.e. menus must usually form a strict tree, not a graph, still less a graph with cycles.

Often users cannot get at any of the sub-menus or actions except by starting with a top-level menu and making selections, i.e. walking down (or up) the tree.

In many case it is difficult to get a good structure for a menu tree (or graph) that makes a collection of options available in a way that suits all users, except where the collection is very small (e.g. at most a few tens of commands, rather than a few hundreds).

Normally, if user-defined menus are supported then a fixed set of functions is provided that can be associated with menu options: it is not usually easy to extend the option space dynamically by defining new functions while the system is running.

-- -- Partial counter-example: TWM menus

One partial exception to the above limitations is the "TWM" (or "TVTWM") menu system [LaStrange 1989] which:

a. allows a menu to point to another menu by name, so that two menus can point to the same third menu, or to each other.

b. allows the menu definitions to be re-read if you wish to change the menu structure at run time.

c. allows simple definitions of new functions by combining old ones.

The main snag is that if you wish to change any menu you have to get TWM to re-read the whole initialization file and the whole collection of menus has to be re-built, which can be a slow process, and if there is an error in the file it is possible to end up without a window manager running.

Moreover, if a large number of menus is specified, then they will take up a lot of space in the window manager process. In TWM you can't have menus added incrementally, as needed. Rather, if you need to have a very large number of menus they all have to be compiled at the same time.

Finally, the language available for building new functions in TWM is very limited.

-- How the recursive hypermenu system helps

RHM gets round some of these problems. It is implemented using the incremental compilation and autoloading facilities in Pop-11. Because of this it has the following features:

o A sub-menu is always accessed via its name, allowing arbitrary cycles in the menu graph. One consequence is that each menu can include a pointer to the top level menu so that from any menu one can quickly get back to the top from any sub-menu. (Or to the top of some suitable intermediate sub-graph.)

o When a menu is accessed, if the name has no value it will be autoloaded by Pop-11 from a file, found in a suitable directory on a user-modifiable search list. Similarly if a menu invokes a Pop-11 procedure or library, it need not be pre-compiled as it will be autoloaded from the source file(s) when that option is first invoked. Thus, very large numbers of menus and menu-invoked utilities can be made readily available with very little run-time cost as they are not compiled until needed. Only those menus and library procedures actually used will involve any extra cost in time taken to load menus or process space. An individual user or group of users can, however, reduce start-up time by building a saved image containing frequently used menus precompiled.

o Autoloading uses search lists, and different users use different search lists. Thus it is possible for all users to share a large collection of generally useful menus (control panels) while individuals or groups have local extensions (like the local library directories, or user directories, in Poplog or $PATH in Unix).

o Recompilation of any menu definition can be forced after the menu has been edited, without recompiling anything else. The next time that menu is invoked the new definition will be used. Thus editing and changing one menu is very quick. This both simplifies and speeds up testing during development of a collection of menus and also makes it very easy for a user to modify a menu or add a new one.

o The actions associated with a menu button can be arbitrary Pop-11 procedures, including procedures that run Unix commands and read the output into an editor window. Users can define arbitrarily complex actions to be invoked by selecting a menu button, either by defining a new procedure to be associated with a button, or by using the notation for specifying complex actions, described below.

o Because menus are accessed via global identifiers, it is not necessary for the user to start with a top level menu and crawl down a tree to get a particular menu. A command is available for getting directly to a menu by using its name.e.g. “ENTER menu news” will get the menu that gives news reading options.

So the designer of a new package that provides a new collection of commands doesn't have to make those commands fit into an existing menu hierarchy. The package can have its own top-level menu.

-- Interface issues

What is the best way to make available a set of menu-driven commands when the total number of commands is very large?

Clearly any deep tree structure will be hard to use because people usually cannot decide which branch to take near the top of the tree, and searching in deep trees can be a very slow and confusing process. There is considerable evidence that shallower broader trees are easier to use because people take the wrong option less often. This is probably because when a larger collection of options is displayed at each choice point, the mutual disambiguation of options makes it less likely that the wrong choice will be taken.

In any case, having to go down several branches to get to the required option is likely to be tedious especially for options that are often required.

Suppose each menu has at most about 15 entries (excluding Help, Dismiss, and access to the top level menu): then by using a hierarchy with only two layers below the top level, it is possible to accommodate 15*15 = 225 commands. Implementors of packages that need more options than can be accommodated in this way can choose between:

- allowing more than two steps to an option

and

- allowing a larger number of options per menu.

E.g. allowing a maximum of three steps from the top level would permit 3375 actions. Allowing four steps and a maximum of ten items per menu would permit 10,000 actions.

The RHM system does not in itself limit the number of steps required to get to an action nor the menu size: that is up to the designer of a particular menu family, or a user who extends it.

However, the default demonstration version manages to provide a great deal of commonly required VED-based functionality within a maximum of two or occasionally three steps from top-level and using many menus with fewer than 15 items.

-- The RHM solution

The draft RHM package attempts to get round the problems of using menus to drive a system, by means of the following combination of features and conventions (though the mechanisms used will allow other solutions):

1. A particular package has a top-level menu and several other menus accessible from the top-level package (or more directly).

2. All the other menus can be invoked from the top-level menu in at most two, or occasionally three, steps.

3. Each submenu includes an option to get back directly to the top-level menu.

4. Some sub-menus in a package are directly accessible from other sub-menus.

5. Several sub-menus, from different parts of the RHM network can be on the screen simultaneously. When a menu is invoked it automatically remains on the screen until it is dismissed (e.g. by clicking on its **Dismiss** button or clicking on the **DismissAll** button on the top level menu, or by using a suitable keyboard accelerator). Thus repeated commands from the same action family can be given quickly because they are on the same menu, which need not be re-invoked. Some of the menu buttons produce an action immediately. Some of them invoke a temporary control panel (dialogue box) that can be used to refine the action before it is executed: e.g. for search and replace, the user can specify the search and replacement patterns, and also various options such as whether embedded strings are to be replaced and the scope of the operation (whole file, current procedure, current line, marked range.)

6. The current convention in RHM is that all the menus for a package will appear in approximately the same place (e.g. bottom right of screen), except for the top level menu which appears to side. Thus after several menus have been selected there will be a pile of them, all of which are obscured except for the one at the top of the pile, and the top-level menu which is left uncovered. Then any desired menu in the pile can be summoned to the top of the pile instantly by selecting it from the toplevel menu, if it is included there, or alternatively by dismissing (or iconizing) others till the desired menu is re-exposed. If a particular sub-menu is used sufficiently often, it can be moved to one side so that it doesn't get covered by others. (If the window manager is “TWM” there is also the “raiselower” mouse option that can be used to cycle round a pile of overlapping menus.)

(Note: At present selecting a menu will not de-iconify it, but I think that's a bug in the propsheet_show library procedure, which calls XtPopup, which apparently doesn't de-iconify. Using “grep” on the x/pop/ref directory gave no clues as to how to do de-iconify a widget.)

7. Additional conventions to simplify navigation are as follows:

(a) Each menu ends with the “Dismiss” option, which removes the menu from the screen.

(b) All but the top level menu have as the second last option a button to get the top level menu visible. Some may also include pointers to intermediate "top level" menus for sub-packages.

(c) The top level menu has a DismissAll option, which removes all menus from the screen.

(d) Near the top of the menu is a button that will display a help file on the actions (or sub-menus) listed in the menu.

8. Because menus normally remain on the screen once invoked, and they remember where they were moved to if they are moved then dismissed, the user can add extra spatial structure to the menus by grouping them in particular locations on the screen.

(In the initial implementation using propsheets, options are constrained by the fact that all menus are vertical columns of labelled buttons. It should be possible to add an option to make certain menus horizontal, or to make then contain N rows and K columns, if that is preferred. This would add to the variety of spatial structures that users can deploy to help them remember where different sorts of commands are available on the screen. Use of a "virtual desktop" like TVTWM or OLVWM also helps.)

9. Many of the commands made available through the RHM package are of a type that experienced users would normally access directly via an ENTER command on the editor command line. To facilitate learning of these accelerated commands many of the RHM procedures put the appropriate command on the editor command line. So the user can then use REDO to repeat the command, or edit the command line to give a slightly different command. Some users have reported that this provides very useful learning.

10. A VED procedure, ved_menu, is provided that allows any menu whose name is of the form ps_menu_<name> to be invoked by the VED command "ENTER menu <name>"

(The prefix 'ps_menu' was chosen because the implementation uses Poplog's LIB PROPSHEET [Meyer 1991].).

11. It is easy for users to associate particular frequently used menus with VED function keys or key sequences, so that they can be invoked directly, without having to traverse a menu graph.

12. It would be easy to arrange the same family of actions in different forms for users with different degrees of expertise. E.g. for beginners only a subset of options might be shown. For experts, actions that are normally easily invoked from the keyboard might be excluded from the menu options and more advanced options shown as reminders. Users can also easily make changes such as moving a frequently used menu closer to the top level menu.

More generally, instead of having arbitrary numerical measures for degrees of expertise, or fragile automated restructuring by an "adaptive" system, user-driven re-structuring can be used to tailor the system closely to different kinds of expertise or lack of expertise.

-- -- NOTES on the current implementation

a. Desirable but not yet available is a simple menu mechanism for reorganizing menus. At present this has to be done in the editor VED. However, it is fairly easy to create a new menu from existing menus by copying lines from the menu definition, or to modify a menu definition by rearranging lines, adding lines from another definition, etc. VED's facilities for marking ranges and moving them or copying them are already accessible from one of the default menus.

b. All the above works in VED in an xterm window on a sun or X terminal, and is especially useful in conjunction with LIB VEDXGOTOMOUSE, which allows the mouse to be used to move the cursor in the window..

c. Most of the package works with Xved, but sometimes the action selected will not be done immediately. There seems to be an unfortunate interaction between Xved and the general X callback mechanism. The unfortunate proliferation of windows in Xved might be made more tolerable if the menu mechanism were extended to produce a new version of vedfileselect. I have not tried to do this, though ved_rb (rotate buffers) is available already as a menu option.

d. The expectation is that the layout of menus on the screen, in the form of a “persistent pile” of recently accessed and not yet dismissed menus, with the top level menu visible alongside them, reduces the need for additional navigational aids. So far this has been borne out in informal tests. There is no “stack” of current menus although the pile of menus roughly corresponds to a stack. It is not really a stack because the user can invoke a menu, look at it, then without dismissing it invoke another menu from the “top level” menu, on the left. At that stage the second menu on the pile is not one used to get to the current state: it was merely inspected then left on the pile.

So the notion of going back to the previous node on the route to the current menu is not explicitly supported, though it is often achieved simply by dismissing one or more menus from the top of the pile.A more explicit stack could very easily be added, though because the set of menus includes cycles and graphs there is no statically determined predecessor for a menu: the actual sequence traversed would need to be recorded.

It would be trivial to modify the mechanism to keep a list of menus used (including those accessed and dismissed) and to provide tools to allow the user to go back to an earlier menu or to cycle through the list. This would deal with the case where one vaguely remembers seeing an option which now seems to be relevant, but where one cannot recall exactly where it appeared. The record of menus traversed could be maintained in an easily examined and edited history file that persisted across login sessions. Whether this will be useful is a matter for further research.

Additional navigational tools may actually not be worthwhile if the network doesn’t get very deep. If the maximum depth is about 3, then going back to toplevel and working down again is not much of a problem. Alternatively the “raiselower” function in the TWM and TVTWM window managers can make cycling through a pile of menus by clicking on title bars very fast (faster than selecting a “lower” option from a window manager menu.)

However, visual reminders of previous selections could be useful in some cases, and it would be possible to use the Poplog GO facilities to implement this. A map of the complete system would take up too much screen space as the total set of nodes is very large. However a submap related to the user's recent choices might be manageable. There is no implementation of this at present. Further experiments with users may reveal a pressing need, or may show that it is not urgent.

e. Another desirable extension would be automatic (but suppressible) help on each menu as the cursor is moved over it. LIB PROPSHEET does not provide an easy way to support this, but when the package is reimplemented in GO it should be a simple addition, as would allowing clicking with a different mouse button to produce a help message. The real difficulty lies not in providing the mechanism, but in finding a good conceptual grouping for the options and finding good wording for such help messages.

-- Unintended side-benefit: an extendable menu interface to Unix

Although this was not part of the original intention, the menus can also be used to invoke unix commands, either via Pop-11 procedures like `sysobey`, or via `ved_sh`, `ved_csh` or `vedgenshell` (as described in the Poplog HELP PIPEUTILS file). This means that unix commands can be given via menu options with output then read into a VED file. This makes it very easy to create a user-tailorable menu-driven interface to Unix utilities (e.g. who, ls, ps, etc.).

In fact, through the use of Unix pipes it is easy to link together several Unix utilities and invoke them via RHM buttons. It may be desirable to extend the notation described below to support this, though most things are already easy for an experienced Unix user to set up. E.g. a menu button to show who is currently logged in could be defined thus:

```
[WHO 'csh who']
```

when invoked this would run the 'who' command using the C shell, and read the output into a VED buffer.

A button labelled “NewFiles” to show the 15 most recently created or modified files or directories in the current directory could be defined by

```
[NewFiles [UNIX 'ls -lt | head -15']]
```

This facility is described more fully below.

-- A menu notation for RHM

For setting up a large number of menus it is desirable to have a textual formalism that allows users to specify, for each menu, where it should go, what its label is, what explanatory or descriptive text it has, which set of buttons it has, what the labels are on the buttons, and which actions should be invoked if a button is selected. (For that reason it is not expected that the UIDE SIBAL mechanism would be used for initial creation of menus, though special menu-creation tools to help users who are new to VED would be useful.)

Currently each menu is specified by a Pop-11 list structure, with the following format, which is provisional, and may change.

(In Pop-11 it is so easy to define an extension to the compiler that almost any desired alternative notation could be implemented easily. The non-trivial part is not so much implementing a syntax but deciding what needs to be expressed in it.)

-- Format of menu definition list

Each menu definition (at present) has the following format:

```
global vars ps_menu_<name> =
  [<menulabel>             ;;; Label for title bar
     {<x> <y>}              ;;; optional screen location specifier
     <menu description>     ;;; string, with newlines for line
                            ;;;breaks, displayed at top of menu.
     [<buttonlabel> <action>]
     [<buttonlabel> <action>]
     .......
    ];
```

By convention many or all menus will end with the button specifier:

```
    ['MENUS...' [MENU ps_menu_toplevel]]
```

making it possible to go quickly to the toplevel menu. However, this is not a compulsory feature, and sometimes an intermediate “top level” can be used instead.

Each <buttonlabel> can be either a word or a string. Certain conventions are suggested but not prescribed regarding the formats used for different kinds of labels. A more detailed description follows.

-- -- Menu Label

The <menulabel> is a string that will appear at the top of the menu and specifies what the menu is about. E.g. 'Marked range'. Whether and how it actually appears will depend on the window manager. Typically it will be on the title bar. Choose a short label, as title bars are usually quite short on menu panels.

-- -- Optional Screen Location

The (optional) {<x> <y>} specification is a vector of two integers, representing pixel row and column of the top left of the menu box in screen co-ordinates, with {0 0} representing top left of the screen. Note that if you wish the title bar to be visible you should add its height to the y coordinate.

[[It should be possible for the coordinates to be incremental, e.g. of the form {+<x> +<y>} or possibly with “-” instead of “+”. This would allow relative placements to be specified, either relative to previous menu or relative to the current toplevel.]]

If the specification is not given then a default location will be used (at present based on the resource Poplog*geometry, though this may change).

-- -- <Menu description>

<menu description> should be a string which will be displayed in a panel at the top of the menu. The string can contain newline characters (as in 'Large\nMoves') to represent line breaks. Lines should not get too long as the menus should be narrow.

-- -- <buttonlabel>

Each <buttonlabel> is a word or string and should be a mnemonic to remind users of the action that will be invoked by clicking on that label. (The same label can be used with slightly different meanings in different contexts. E.g. buttons labelled “HELP” might get different sorts of help in different contexts.)

The labels should be short, as the width of the whole menu column will be determined by the widest label.

In the current menus the following conventions are used:

- o labels indicating other menus all end with '...', e.g. '**ReadNews**...'
- o labels that correspond to advice on VED ENTER commands all end with an asterisk, e.g. '**Search***'
- o The label '**MENUS...**' is used to indicate a button that invokes the “top level” menu.

(Any of this could easily be changed.)

-- -- Format of <action> specifiers

The <action> element following a button label can be any of the following forms:-

a string

a word

a procedure

a vector containing a string

a list starting with a keyword that specifies how the list is to be interpreted.

More detailed specifications follow.

-- -- -- A word:

This should be the name of a variable whose value is one of the other structures described below. It will be de-referenced and the appropriate action taken. (recursive_valof is used to get the value.)

-- -- -- A string:

The string will be given as argument to veddo. I.e. the string can be anything that could be put on the command line in VED. For multiple commands see HELP VED_DO. Example of such a button specifier could be

```
[HELP 'help ved']
```

I.e. pressing the button labelled “HELP” would cause veddo to be applied to the string 'help ved'. Thus it is easy to add help or teach options to a menu.

NOTE: if you are using Xved, then instead of veddo being applied to the string immediately, the action is put in the input queue for VED to be obeyed when control next returns to VED. It seems to be quite difficult to make a menu-driven command interface interact properly with Xved. It feels as if there is a design flaw somewhere, either in Xved, or in the callback mechanism.

-- -- -- A procedure:

The procedure will be invoked. E.g. a menu specification could include the following item:

```
[QGoIndex ^(ved_q <> ved_g)]
```

If selected, this button will cause the procedure ved_q and the procedure ved_g to be run.

-- -- -- A vector containing a string

The string is treated as Pop-11 text and is compiled. An example equivalent to the above would be:

```
[QGoIndex {'ved_q();ved_g();'}]
```

This is useful for defining compound actions without having to define a new procedure to be invoked.

-- -- -- A list:

The list defining a menu action can take one of the following forms

```
-- -- -- 1. [ENTER <commandstring> {xloc yloc} <explanation> ]
```

Where the {xloc yloc} vector and the <explanation> are optional.

An explanation box will pop up. If the location vector is not provided the centre of the screen will be used. The explanation box will give default instructions and in addition will display the <explanation>, if that is provided. The <explanation> can either be a string (including newlines) or a list of strings. In the latter case each string will be displayed on a separate line.

The <command string> will be displayed in an editable text field. The user can then modify the command, and then press RETURN or the "Do" button.

If the location vector is not given, the centre of the screen is used.

Examples of ENTER action specifications are

```
['NewFile*' [ENTER 'ved <filename>' 'Editing a new file']]

['GetHelp*' [ENTER 'helpfor <word>'
       ['Get a list of help files'
        'possibly relevant to' 'a word']]]

['GetRef' [ENTER 'ref <topic>']]

['Rename*' [ENTER 'dired -mvd <newname>' {600 20}
    ['Rename the file to right'
     'of current line to <newname>'
    '(i.e. rename in same directory).'
    'See "man mv" (and HELP DIRED)']
    ]]
```

```
-- -- -- 2. [MENU <menuname>]
```

Here <menuname> should be an identifier whose value is a menu, either a list defining a menu, or the propbox created to correspond to the menu. Examples of action specifications using this format are:

```
['Mark1...' [MENU ps_menu_mark1]]
['Mark2...' [MENU ps_menu_mark2]]
['Dired...' [MENU ps_menu_dired]]
['Move1...' [MENU ps_menu_move1]]
```

If one of these options is selected by the mouse, then, if the menu does not yet exist the name (e.g. `ps_menu_dired`) will cause its definition to be autoloaded, and the list defining the menu will be transformed into a propbox with menu buttons on it which then becomes the value of the variable, instead of the list. If the propbox already exists, it is merely displayed.

```
-- -- -- 3. [POP11 <pop11 code>]
```

The tail of the list can contain Pop-11 text items and is given as argument to the procedure `compile`. Arbitrary actions are possible. Although there will often be no benefit in using a list action of this form rather than a vector containing a string, this option is useful where the contents of the list (i.e. the command) should be in part be determined (at compile time) by the value of a global variable, or an embedded procedure call.

Moreover, if a string is used it will require itemization as well as compilation to be performed, whereas a list already contains separate items. However, the itemization process should not make a noticeable difference. It is possible that in some cases the list format will allow advantage to be taken of sections or lexically scoped variables.

```
-- -- -- 4. [UNIX <unix command> <shell specification>]
```

The <unix command> is run using vedgenshell and the output is read into a VED buffer.

If the <shell specification> is not given, then it defaults to systranslate('SHELL'). For example, here is a specification for a button that will list all the processes belonging to the current user (on SunOS 4.1.x):

```
[MYPROCS [UNIX 'ps -aux | egrep "$USER|USER"']]
```

On clicking on the button labelled “MYPROCS” the user would spawn a shell process to run “`ps -aux`”, the output of which would be filtered through the command

```
egrep "$USER|USER"
```

and the output of that read into a VED buffer, where it can be inspected or processed further. (Note that the first occurrence of “|” creates a Unix pipe, whereas the second expresses a disjunctive search string for egrep.)

This ability to link applications easily seems to be similar to that provided in the Microcosm system [Hill et al. 1992]. As I have seen only a short overview document I cannot comment on relative ease of use, though RHM potentially provides a lot more options because it is embedded in a powerful software development system.

And here is one that specifies the C shell, so that “~” can be interpreted properly: it lists all the directories in the user's top level directory.

```
[MYDIRS [UNIX 'ls -l ~ | egrep "drw"' '/bin/csh']]
```

-- -- Additional special keywords

It is possible that additional special keywords and formats will be introduced later to deal with frequently occurring types of actions.

There are various global variables and user settable resources that control defaults, e.g. colour and font of menu displays or explanation boxes.

-- Unsolved problems

There are many unsolved problems. Some of them are merely a matter of finding the time to implement an additional facility. Some of them require empirical research or detailed task analysis. (In some cases research has already been done elsewhere though typically only in connection with toy problems.)

1. Should the items within a menu be organized alphabetically or grouped logically? (Let the user decide?) With this system it is easy to experiment.

2. Is it better to accommodate a large number of options via several small static menus or via a single large scrolling menu?

3. Is it useful to allow string search to operate on long menus to help users find items? Or is it better to avoid long menus?

4. When, if ever, are icons preferable to textual labels on options? (Conjecture: icons are useful only when the number of items to be discriminated is no more than about 10. In part it depends on whether the options remain permanently on display. Icons tend to require more screen space. Large amounts of permanently occupied screen space can be intolerable.)

5. What is a good way to let the user specify where different collections of menus should appear on the screen? (Currently screen coordinates can be part of the individual menu specification.)

6. Can the basic menu system be naturally built up using a menu-based interface for creating menus?

7. Should the user be provided with some kind of graphical map of available actions, showing which bits of the map he/she has already used and which other bits are readily accessible from those?

8. How can mnemonic menu labels be chosen to suit lots of different kinds of users? (Answer: it's impossible?)

-- Acknowledgments

The current implementation makes heavy use of facilities provided in the Poplog PROPSHEET library implemented by Jon Meyer, at Integral Solutions Ltd. Adrian Howard, at Sussex University, provided a number of additional utility procedures to help drive the X facilities from Poplog.

I would also like to acknowledge help, comments, suggestions, testing of the implementation and consequential suffering contributed by Kath Bale, William Edmondson, Bob Hendley, Petra Hickey, Tom Khabaza, Sophia Langley, Jane Littlehales, David Mallaburn, Keith Marlow, John Williams, and student testers at Birmingham.

-- Incomplete bibliography

W.H.Edmondson (1993) A taxonomy for Human Behaviour and Human-Computer Interaction. Internal report, School of Computer Science, the University of Birmingham.

Gary Hill, Rob Wilkins, Wendy Hall, Open and reconfigurable Hypermedia Systems: A Filter-Based Model Report CSTR92-12, Dept of Electronics and Computer Science, University of Southampton 1992

Thomas E. LaStrange (1989) TWM - a window manager for X11 (Tom's Window Manager), (On-line X11 MAN file), Hewlett Packard Company Graphics Technology Division Fort Collins, Colorado

Debora J Mayhew (1992) *Principles and Guidelines of Software Interface Design*, Prentice Hall, New Jersey.

Jon Meyer (1991) TEACH PROPSHEET, Poplog on-line documentation file

Jenny Preece (ed) *A guide to Usability: Human factors in computing* Addison-Wesley 1993

Ben Shneiderman *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison Wesley, Reading Mass, 1987. (There is a more recent addition.)

A. Sloman (1993) HELP VED_MENU, Draft on-line Poplog documentation file for users of the RHM system

Ralph R. Swick (1989) Xfontsel- point & click interface for selecting X11 font names (on-line X11 MAN file), Digital Equipment Corporation/MIT Project Athena (1989)

-- Trade Marks

“Unix” is a trade mark of AT&T Bell Labs

“Poplog” is a trade mark of the University of Sussex.

“X Window System” is a trademark of the Massachusetts Institute of Technology.

“FrameMaker” is a trademark of Frame Technology Corporation.