A SURVEY OF REUSE FACILITIES

Ian H Witten and Saul Greenberg
1989-374-36
December 1, 1989

Reuse facilities help people to recall and modify their earlier activities and re-submit them to the computer. This paper surveys existing reuse facilities under three main headings: history mechanisms, adaptive systems, and programming by example. The first kind relies on temporally ordered lists of interactions, the second builds abstract models of past activities and uses them to expedite future interaction, while the third collects and generalizes more extensive sequences of activities for future reuse. A companion paper (Greenberg & Witten, 1989) presents the results of a large-scale study of how users actually repeat their activities on computers and contrasts the multitude of opportunities for reuse with the relatively infrequent use of an actual history mechanism.

Department of Computer Science, University of Calgary, Calgary, Alberta, CANADA.

1 Introduction

Users often repeat activities that they have previously submitted to the computer. In command-driven systems, these activities include not only individual commands, but also complete command lines, including filenames and options. Likewise, people repeat the ways they traverse paths within menu hierarchies, select icons within graphical interfaces, and choose documents within hypertext systems. Often, recalling the original activity is difficult or tedious. For example, problem-solving processes must be re-created for complex activities, command syntax or search paths in hierarchies must be remembered, input lines retyped, icons found, and so on. Given these difficulties, potential exists for a well-designed "reuse facility" to alleviate the problems of activity reformulation by keeping previous activities ready to hand.

This paper explores interactive reuse facilities that allow users to recall, modify, and re-submit their previous entries to computers. Although the idea is simple—anything used before may be used again—it is only effective when recalling old activities is less work for the user (cognitively and physically) than submitting new ones. As we shall see, the main differences between reuse facilities arise from their ability to offer a small but good set of candidates for possible reuse, and from the user interface available to manipulate these candidates.

Reuse facilities have loose analogies in non-computer contexts. A cook can explicitly mark preferred recipes by using bookmarks. "Adaptive" marking takes place by the book naturally opening to highly used locations through wear of the binding and food-encrusted pages. Or consider the audiophile who places records just listened to at the top of the pile. Assuming that certain records are favored over others, popular records tend to remain near the top of the stack and unpopular ones near the bottom. A carpenter's workbench has an implicit reuse facility—the work surface is large enough to keep recently used tools ready to hand.

Existing reuse facilities for computers fall into three classes, as described and illustrated in the following sections. The first class includes *history mechanisms* that let users manipulate a temporally-ordered list of their previous command-line interactions. Depending upon the interface style, items are retrieved and selected through textual syntactic constructs, by pointing to menu items and buttons, or by editing dialog transcripts. Included in this category are path traces and "bookmarks" maintained by systems requiring user navigation, such as hypertext systems and menu hierarchies. The second class of reuse facilities, *adaptive systems*, use dynamic non-temporal models of previous inputs to predict subsequent ones which are then made available to the user. These include menu-based interfaces that dynamically reconfigure the menu hierarchy so that high frequency items are treated preferentially, and text predictors that maintain Markov models of the text entered so far to predict further submissions. The third class, *programming by example*, is concerned with reuse and generalization of long input sequences—its technology is significantly less developed

than in the other two cases.

A large number of implemented designs will be surveyed within this taxonomy of reuse facilities, illustrating the diversity of techniques available.

2 History mechanisms

History mechanisms are based upon the assumption that the last few submissions provide a reasonable set of candidates for reuse. This notion of "temporal recency" is cognitively attractive. Since users generally remember what they have just entered, they can predict the offerings available. Little time is wasted reviewing the list of candidates only to discover that the desired item is missing.

History mechanisms are by far the most common reuse facility available, and are implemented across diverse systems in a variety of flavours. Four fundamentally different interaction styles are described in this section: glass teletypes; graphical selection; editing transcripts; and navigational traces. The first three provide a reuse facility for command-line interfaces, while the last illustrates its application to systems in which users navigate within some information structure.

2.1 History in glass teletypes

Before graphical interfaces came into vogue, dialogs were simple command-line systems designed for the teletype—the VDU being a fixed viewport into a virtual roll of paper. Two functionally rich history systems designed for such physically limited "glass teletypes" are the UNIX *csh* and the INTERLISP-D Programmer's Assistant. In both systems, old commands are retrieved by "history" directives, themselves commands interpreted in a special way.

UNIX *csh* maintains a record of user inputs, where every string entered on the command line is placed on a numbered event list (Joy, 1980). Special syntactic constructs allow previous events to be recalled, either by position on the event list (relative or absolute), or by pattern-matching. Actions on recalled events include viewing, re-execution, retrieval of specific command line words, and text substitution. Although the set of predictions is unbounded in size, it is practically small—users forget all but the last few items, and reviewing a long list is cognitively unattractive.

Figure 1 illustrates a possible event list and a few examples of *csh* history in use. The inputs in the left column are translated by *csh* to the actions shown in the middle. The translation is described in the column on the right. As the examples illustrate, the syntax is quite arcane, and deters use of the more powerful features. Additionally, since the event list is generally invisible—snapshots of its current state are displayed only by special request— it is difficult to refer to any but the last few events.

—Figure 1 around here—

Another functionally powerful history mechanism is the Programmer's Assistant, designed for the INTERLISP-D programming environment (Teitelman & Masinter, 1985; Xerox, 1985). Although INTERLISP-D is window-based, the top-level "LISP listener" window resembles a glass teletype. The Programmer's Assistant history mechanism improves upon that of UNIX *csh*. More than one event can be retrieved and manipulated at a time, iteration and conditional specification are allowed, items can be edited, effects of previous entries may be undone, and so on. In normal use, events are selected and processed by special command directives entered in the LISP listener window. These tend to be verbose. For example, the request *USE cons FOR setq IN -1* will replace the string "setq" by the string "cons" in the previous command. Figure 2 shows a sample dialog in the window labeled "Interlisp-D Executive", where events 85 and 87 make use of the Programmer's Assistant. As with *csh*, neither duplicates nor erroneous statements are pruned from the event list.

2.2 History through graphical selection

The technology of terminals has evolved since the early glass teletypes. All terminals now have positional control of text on the screen, and high-resolution graphics terminals with locators are common. Interaction styles have also progressed from text-oriented menus and forms to locator-oriented graphical systems running within windows (Witten & Greenberg, 1985). Within the latter, history mechanisms have been extended to present a (possibly transient) menu of previous events. Items are selected and manipulated with a locator, usually a mouse. In contrast to glass teletype history, predictions are offered by presenting them explicitly on the screen.

One example is HISTMENU, which provides a limited yet simple way of accessing and modifying the INTERLISP-D Programmer's Assistant history list (Bobrow, 1986). Figure 2 illustrates its use. Commands entered to the INTERLISP-D Executive window are recorded on the history list, part of which is displayed in the History window (by default, the last 50 items are shown). Although the history list itself is updated on every command, the window is only redrawn when the user explicitly requests it. When pointed at with a mouse, items (which may not fit completely in the narrow History window) are printed in the Prompt window. Any entry can be re-executed by selecting it. Moreover, a pop-up menu allows limited further action: items can be "fixed" (*ie* edited), undone, printed in full including additional detail (the "??"), or deleted from view. The History window also has a shrunken form, as shown by the icon in the Figure.

—Figure 2 around here—

MINIT is another graphical package that combines command processing and the history list into a single "window management window" (*wmw*) (Barnes & Bovey, 1986). MINIT differs from other systems in that

only through this window can the user send commands to the other ones. The *wmw* is divided into three regions. The first is an editable typing line at the window's bottom, where commands are entered. Once entered, they are added to the second region which contains a scrollable history list. As with HISTMENU, the user selects items in the list through a locator and controls further action with pop-up menu options. A specialized history management menu comprises the final region. Options are available to

- scroll the history list, clear it, or save it for future use;
- textually search for specific items;
- delete specific items;
- insert text in the typing line without executing it.

Two more mechanisms complete MINIT's history management capabilities. First, the user can customize the system to prevent short commands which are easily retyped from being added to the list. Second, history is viewable in either alphabetic or execution order. Although duplicate lines are eliminated, the user can control whether a command entry which is repeated remains in its original position on the execution-ordered list or is relocated to the end. A side effect of moving recurrences to the end is that the most frequently used commands tend to cluster around one another.

How does one review and modify the actions performed within a purely graphical interface, such as those provided by painting or drawing applications? One solution, offered by Kurlander and Feiner (1988) in a system called *chimera*, builds on the visual metaphor of a "comic strip." The strip is a visual, graphical record of the user's past activities, where each panel in the sequence illustrates an important moment in a story. Instead of just showing miniatures, a panel emphasizes the objects being manipulated and the actions performed on them, removing unnecessary detail. The user can then expand the detail shown in a particular panel to a level appropriate to the task to be performed; delete, modify, undo and redo the actions expressed in the panel; and add new actions.

2.3 History by editing transcripts

Some systems do not have a command history mechanism *per se*, but provide similar capabilities through editing the transcript of the dialog.

The glass teletypes described previously are actually more limited than paper teletypes, for it is not possible to review text that has scrolled off the screen. This limitation is exacerbated by high-speed terminal lines, for text appears and disappears faster than it can be read. Although page-holding mechanisms that stop scrolling after every screenful offer a palliative, the advantages of the original paper teletype were finally

realized when scrollable transcripts of the dialog were maintained.[1] Unlike paper teletypes, these transcripts have potential as a history mechanism, for text appearing previously can be pointed at and used as input to the system.

In Apollo's DOMAIN window system, for example, text appearing within specialized windows called "pads" can be copied and then pasted and edited in any command input area (Apollo, 1986). Explicit history lists are not maintained except as part of the scrollable dialog transcript. Similar cut and paste capabilities are available in most modern-day window-based environments. The trade-off here is evident. Although *any* text in a pad is potentially executable, the mixing of previous input commands with output probably makes useful candidates difficult to find.

A second system which encourages use of command history through editing is *emacs*, an editing environment which provides multiple views of buffers through tiled windows (Stallman, 1981). Although buffers typically allow users to view and edit files, it is also possible to run interactive processes (or programs) within them. In most Unix-based implementations of *emacs*, it is a simple matter to call up a window running UNIX *csh* (for example: Stallman, 1987; Unipress, 1986). All capabilities of *emacs* are then available—commands may be edited, sessions scrolled, pieces of text picked up from any buffer and used as input, and so on.

As a further variant, consider the *zmacs* editor running within the Symbolics Genera LISP environment, which contains features of all history systems discussed so far (Symbolics, 1985). Within the top-level Lisp Listener, *zmacs* extends the functionality of *emacs*. Although used here primarily for entering and editing command lines, previous inputs appearing within the transcript become mouse-sensitive. A box appears around them as the mouse passes over them, while clicking one of the three mouse buttons causes some action to occur. For example, pressing the left button of the mouse over the old command line copies it into the input area, which is then available for further editing. Other combinations of keys immediately re-execute previous commands, copy arbitrary command words, show context-sensitive documentation, and so on. Using the standard editing commands within the one-line input area, a user can search, cycle through, and recall previous events, similar to the command-line capabilities of the VMS operating system (DEC, 1985), the UNIX *tcsh* (Ellis *et al*, 1987), and GnuEmacs (Stallman, 1987). Alternatively, part or all of the mouse-sensitive event list can be displayed within the Lisp Listener window.

2.4 History by navigational traces

While the above techniques deal only with command-line interfaces, history has also been applied to information systems and networks where items are retrieved through navigation. These are also considered to be

[1] An alternative solution to transcripts is to make every system facility responsible for formatting its output appropriately. The tradeoff between the two approaches is discussed by Pike and Kernighan (1984).

history systems as they are usually based upon temporal route paths taken through the data base.

There are many examples of systems and databases where users tend to retrieve items of information that have been accessed previously (Greenberg & Witten, 1985). History for hypertext systems, for example, are based on the assumption that previously-read documents are referred to many times. This assumption has been supported by a study of *man*, the UNIX on-line manual (Greenberg, 1984), Each user frequently retrieved the same small set of pages from the large set available, where sets differed substantially between users. By keeping a history list of the documentation retrieved, users can avoid re-navigating the hypertext route for a previously viewed topic. Since items on the list can be viewed as place-holders in a large document, they are known as "bookmarks."

The Macintosh HyperCard is a hypertext facility that allows authoring and browsing of stacks of information comprised of cards. Navigating cross links between stacks and cards is usually accomplished by simple button or menu selections. *Recent* is a bookmark facility within HyperCards that maintains a pictorial list of up to the last 42 unique cards visited (Figure 3). Each picture is a miniature view of the card, placed in the list by order of first appearance.[2] The last card visited is distinguished by a larger border, as illustrated by the second miniature in the first row of the Figure. A pull-down menu option pops up the *recent* display, and old cards are revisited by selecting their miniature from the list (Good, Whiteside & Jones, 1987). When more than 42 unique items have been selected, the first row of seven items is cleared and made available for new ones (even though a card in the first row may have recently been selected).

—Figure 3 around here—

The Symbolics environment includes a very large on-line manual viewable with the Document Examiner—a window-based hypertext system (Symbolics, 1985). The main window is divided into functionally different panes: a documentation display area; a menu of topics; a bookmarks area; and a command line. The bookmarks area displays a history list of titles of previously-viewed topics, where each title is a bookmark. Further bookmarks may be explicitly added by the user (these are visually distinguished from historical bookmarks). Selecting a bookmark displays either full documentation or a brief summary of the topic in the documentation area. A similar bookmark strategy has been proposed previously for videotext systems (Engel, Andriessen & Schmitz, 1983).

Of course, these techniques are not limited to hypertext. Navigation is ubiquitous in many human-computer interfaces, from hierarchical menu and folder systems, to structured browsers for programming systems. All could profit from maintaining a history map of the user's route through the structure.

[2]Figure 3 is a fairly accurate representation of the screen. As these miniature pictures are of surprisingly poor quality, the value of the current *recent* implementation is questionable. However, this problem could be overcome by a higher-resolution display or by including a "magnifying glass".

3 Adaptive systems

History mechanisms model the user's previous inputs by recording them in a history list. Adaptive systems build more elaborate dynamic models and use them to predict subsequent inputs, which are presented to the user for selection or approval. Here we describe two types of adaptive systems, one for accelerating selection in a hierarchical menu system and the other for the entry of free text. Both employ predominantly frequency-based, rather than recency-based, models.

3.1 Dynamic menu hierarchies

It is possible to devise interactive menu-based interfaces that dynamically reconfigure a menu hierarchy so that high-frequency items are treated preferentially, at the expense of low-frequency ones. This provides an attractive way of reducing the average number of choices that a user must make to select an item without adding further paraphernalia to the interface (Witten, Cleary & Greenberg, 1984). Consider a telephone directory where the access frequencies of names define a probability distribution on the set of entries (Greenberg & Witten, 1985) This reflects the "popularity" of the names selected. Instead of selecting regions at each stage to cover approximately equal ranges of names, it is possible to divide the probability distribution into approximately equal portions. During use, the act of selection will alter the distribution and thereby increase the probability of the names selected. Thus the user will be directed more quickly to entries which have already been selected—especially if they have been selected often and recently —than to those which have not.

Figures 4a and 4b depict two menu hierarchies for a very small dictionary with 20 name entries and their corresponding top-level menu. Figure 4c calculates the average number of menus traversed per selection. In Figure 4a, the hierarchy was obtained by subdividing the name space as equally as possible at each stage, with a menu size of 4. The number following each name shows how many menu pages have to be scanned before that name can be found. Figure 4b shows a similar hierarchy that now reflects a particular frequency distribution (the second number following the name shows the item's probability of selection). Popular names, such as Graham and Zlotky, appear immediately on the first-level menu. Less popular ones are accessed on the second-level menu, while the remainder are relegated to the third level. For this particular case, the average menus traversed by probability subdivision are reduced from uniform subdivision, although not as much as is theoretically possible (Figure 4c).

As probabilities also decay over time, once-popular (or erroneously-chosen) names eventually drop to a low value. A decay factor also builds in a way of balancing frequency and recency. While low decay will see frequently-chosen items migrate up the tree, a high decay rate gives more room to recently-chosen items.

—Figure 4 around here—

Given a frequency distribution, it is a surprisingly difficult problem to construct a menu hierarchy that minimizes the average number of selections required to find a name. Exhaustive search over all menu trees, while possible, is infeasible for all but the smallest problems. Witten, Cleary and Greenberg (1984) have studied the problem and describe simple splitting algorithms that achieve good performance in practice.

The novelty of dynamic menus is that previous actions are almost always easier to resubmit. Also, since no extra detail is added to the interface presentation, screen real estate is preserved. To their disadvantage, users must now scan the menus for their entries all the time, even for those accessed frequently. Since paths change dynamically, memory cannot be used to bypass the search process. However, experimental evidence suggests that this is not a problem in practice. As long as the database of entries is very large, the benefits far outweigh the deficiencies (Greenberg and Witten, 1985; Trevellyan and Browne, 1987).

3.2 Reuse through text prediction

History mechanisms assume that the last submissions entered are likely candidates for re-execution. They are the ones visible on the screen in graphical and editing systems, the ones most easily remembered by the user in glass teletypes, and the ones given a greater probability in dynamic menus (although this depends on the decay factor).

Two systems provide an alternative strategy for textual input—the "Reactive Keyboard" (Witten, Cleary & Darragh, 1983; Darragh, 1988) and its precursor *predict* (Witten, 1982). Although implementation details differ, both use a dynamic adaptive model of the text entered so far to predict further submissions. At each point during text entry, the system estimates for each character the probability that it will be the next one typed. This is based upon a Markov model that conditions the probability that a particular symbol is seen on the fixed-length sequence of characters that precede it. The order of the model is the number of characters in the context used for prediction. For example, suppose an order-3 model is selected, and the user's last two characters are denoted by 'xy'. The next character, denoted by ϕ, is predicted based upon occurrences of '$xy\phi$' in previous text (Witten *et al*, 1983).

Predict filters any glass-teletype package, although limited character graphics capabilities are required for its own interface. It selects a single prediction (or none at all) as the most likely and displays it in reverse video in front of the current cursor position. The user has the option of accepting correct predictions as though he had typed them himself, or rejecting them by simply continuing to type. Because only a single prediction is displayed, much of the power of the predictive method is lost; for at any point the model will have a range of predictions with associated probabilities, and it is hard to choose a single "best" one (Witten, 1982).

The Reactive Keyboard, on the other hand, has two versions of a more sophisticated interface that allows one to choose from multiple predictions. The first, called *RK-pointer*, displays a menu containing the best p predictions, which changes dynamically with the immediate context of the text being entered (Darragh, 1988). Figure 5 shows the user composing some free text in the window on the left, and the menu of predictions on the upper right. The underlined "context" string is derived from the last few characters entered, and is followed by the highlighted best prediction. These are shown in both windows. The menu items represent alternative pieces of text from which the user can choose the next characters. Interaction is through a pointing device, such as a mouse. Selection is two-dimensional, in that the user can point anywhere within a prediction to accept only the previous characters (Figure 5). Less likely predictions are available through page-turning.

4 Programming by example

The schemes discussed so far attempt to facilitate the reuse of individual items of activity, be they commands, command lines, menu selections, or characters predicted in context. This is sufficient if incremental activities have a one-to-one correspondence with tasks the user may wish to repeat later. Often, however, tasks are accomplished by sequences of several primitive activities.

"Closure" is defined as the user's subjective sense of reaching a goal, of completion or of understanding (Thimbleby, 1980). Previous sections have assumed that closure was associated with each individual user action (the entry of a command or command line, the selection of a document, etc). If the task to be redone involves a sequence of such activities, even though they are all independently available through a reuse facility, the user would have to mentally decompose his task into its primitives and choose them from the event list. For example, viewing a specific file can comprise two activities—navigating to the correct directory, and printing the desired file to the screen. It will often be easier for the user to think about and recall these items as a single chunk rather than as two separate activities.

When tasks are a fixed sequence of fixed activities, they constitute a simple procedure that can easily be specified by the user giving an instance of the sequence. The goal of "programming by example" is to allow sequences and more complex constructs to be communicated concretely, without resorting to abstract specifications of control and data structure (in a programming language, for example). At its simplest, the user performs an example of the required procedure and the system remembers it for later repetition. For example, the use of "start-remembering," "stop-remembering," and "do-it" commands enable a text editor to store macros of editing sequences (Gosling, 1981; Stallman, 1987). Except for these special commands, the macro sequence is completely specified by normal editing operations. With a little more effort, such sequences can be named and filed for later use. A practical difficulty with having a special mode—remembering mode—

for recording a sequence is that one has frequently already started the sequence before deciding to record it, and so must retrace one's steps and begin again.

The ability to generalize these simple macros could extend their power enormously. Some programming by example strategies allow inclusion of standard programming concepts—variables, conditionals, iteration, and so on—either by inference from a number of sample sequences, or through explicit elaboration of an example by the user. To illustrate the latter, an experimental system has been constructed for the Xerox Star office workstation that operates according to the direct manipulation paradigm (Halbert 1981 & 1984). In earlier versions of this system, a pop-up menu allowed one to indicate explicitly the generalization required. For example, icons selected at specification time are disambiguated by name, position, or by asking for a similar object. But because people found it hard to elaborate programming constructs when tracing through an example, a later version had users employ an editor to specify constructs after macro composition (Halbert, 1984).

Other research on programming by example has concentrated on inferring control constructs from traces of execution given by the user (Witten, MacDonald & Greenberg, 1987), and some systems use domain knowledge, teaching metaphors, and highly interactive interfaces to maximize the speed of transfer of procedures (*eg* Maulsby & Witten 1989; Maulsby, Witten & Kittlitz 1989). However, there has been little research on ways of naming, filing and accessing procedures taught by example, and particularly on knowledge- and history-based methods of splitting up a stream of activities into user-oriented "tasks." This limits the practical use of programming by example in reuse systems.

The appeal of programming by example is the belief that a user's activity follows a preconceived plan that can be encapsulated as a procedure. Intentions are realized as plans-for-actions that directly guide behaviour, and plans are actually prescriptions or instructions for actions. These plans reduce to a detailed set of instructions (which may also be sub-plans) that actually serve as the program that controls the action. Suchman (1985) disputes this notion by claiming that plans are derived from *situated action*—the necessarily *ad hoc* responses to the contingencies of particular situations. Initial plans must be inherently vague if they are to accommodate the unforeseeable contingencies of actual situations of action. It is only the *post hoc* analysis of situated action that make it appear as if a rational plan were followed. Assuming that user activity on computers does follow situated action, then a programming by example system would not suffice by itself as a complete user support tool, for it would not respond well to the changing circumstances of situations. When previous actions are collected as fixed goal-related scripts of events, flexibility is lost. It should be augmented by a reuse facility that collects the actual responses to given situations, allowing one to select, possibly modify, and redo the individual activities.

5 Summary

A reuse facility arranges for submissions entered to the application to be collected and presented so that they can be selected for reuse. Its appeal is the assistance it offers in any dialog that exhibits recurrence. Since no semantic knowledge of the domain is needed, it is quite a general approach. In addition, the mechanism underlying reuse facilities—monitoring the user's interaction and maintaining an internal model of it—has potential for supplying more extensive user support. Possibilities include using the transaction history for undo/redo to allow recovery from errors, providing an external memory aid that allows users to consult or recall information associated with past activities, and building user models (Lee, 1989).

A key deficiency in this general area is the dearth of empirical evidence justifying designs for reuse facilities, either *a priori* through knowledge of how people repeat activities, or *post hoc* by evaluating their actual use. Nor are there any guidelines for how intuitive and empirical knowledge gleaned from one application might generalize to others. As a consequence, existing reuse facilities—as surveyed in this paper—are based on *ad hoc* designs that may not adequately support a person's natural and intuitive way of working.

To rectify this, we have performed a large-scale study of how users actually repeat their activities on computers; preliminary results were described in Greenberg and Witten (1988a & 1988b). and a further report is given in the companion paper to this survey (Greenberg & Witten, 1989). Empirical foundations behind reuse are explored and design principles for reuse facilities offered through a model of human-computer interaction called *recurrent systems.* We find that users' activities are repeated to a surprising degree, and furthermore the probability distribution of the next submission repeating a previous one confirms that recency—the premise behind many history mechanisms—is in fact a reasonable predictor of what the person will do next. Still, there are ways to improve the predictive ability of reuse facilities. A variety of methods using differing predictive strategies are contrasted, some that are provably better than existing ones. In marked contrast to the theoretical potential of reuse facilities, one widely-available history system is shown to be used poorly in practice.

References

Apollo (1986) *DOMAIN system user's guide.* Apollo Computer Inc, Chelmsford, Mass.

Barnes, D. and Bovey, J. (1986) Managing command submission in a multiple-window environment. *Software Engineering Journal,* 1(5):177–183, September.

Bobrow, D. (1986) *HistMenu.* Lisp User Library Packages Manual, Koto Release. Xerox Artificial Intelligence Systems, April.

Darragh, J. (1988) Adaptive predictive text generation and the Reactive Keyboard. Master's thesis, Department of Computer Science, University of Calgary, Alberta, September.

DEC (1985) *VAX/VMS DCL concepts manual.* Digital Equipment Inc, Maynard, Mass, April.

Ellis, M., Greer, K., Placeway, P., and Zochariassen, R. (1987) *TCSH—Cshell with filename completions and command line editing.* Revised University of Toronto edition.

Engel, F., Andriessen, J., and Schmitz, H. (1983) What, where and whence: means for improving electronic data access. *International Journal of Man Machine Studies*, 18:145–160.

Goodman, D. (1987) *The Complete HyperCard Handbook.* The Macintosh Performance Library. Bantam Books, New York.

Gosling, J. (1981) *Unix Emacs Manual.* Carnegie-Mellon University.

Greenberg, S. (1984) User modeling in interactive computer systems. Master's thesis, Department of Computer Science, University of Calgary, Alberta.

Greenberg, S. and Witten, I. (1985) Adaptive personalized interfaces — a question of viability. *Behaviour and Information Technology*, 4(1):31–45.

Greenberg, S. and Witten, I. (1988a) Directing the user interface: how people use command-based systems. In *Proceedings of the 3rd IFAC Conference on Man-Machine Systems*, Oulu, Finland, June 14-16.

Greenberg, S. and Witten, I. (1988b) How users repeat their actions on computers: principles for design of history mechanisms. In *Proceedings of the ACM SIGCHI '88 Human Factors in Computing Systems*, pages 171–178, Washington, D.C., May 15-19.

Greenberg, S. and Witten, I. (1989) Supporting command reuse: Empirical foundations and principles. *International Journal of Man Machine Studies.* Submitted paper.

Halbert, D. (1981) An example of programming by example. Technical report, Xerox Office Products Division, Palo Alto, California. Also available as Master's thesis, Department of Computer Science, Stanford.

Halbert, D. (1984) Programming by example. Technical report, Xerox Office Products Division, Palo Alto, California, December. Also available as PhD thesis, Department of Computer Science, Stanford.

Joy, W. (1980) *An introduction to the C shell, volume 2c.* Unix Programmer's Manual. University of California, Berkely, California, seventh edition.

Kurlander, D. and Feiner, S. (1988) Editable graphical histories. In *Proceedings of the IEEE Visual languages workshop.*

Lee, A. (1989) A taxonomy of interaction history. Internal report, Department of Computer Science, University of Toronto, October.

Maulsby, M. and Witten, I. (1989) Inducing programs in a direct-manipulation environment. In *Proceedings of the ACM SIGCHI '89 Human Factors in Computing Systems*, Austin, Texas, May 2–May 4.

Maulsby, M., Witten, I., and Kittlitz, K. (1989) Metamouse: Specifying graphical procedures by example. *Computer Graphics*, 23(3):127–136, August. (Proceedings of the ACM SIGRAPH conference, Boston.

Pike, R. and Kernighan, B. (1984) Program design in the UNIX environment. *AT&T Bell Laboratories Technical Journal*, 63(8/2):1595–1605.

Stallman, R. (1981) Emacs: the extensible, customizable self-documenting display editor. *ACM Sigplan Notices — Proceedings of the ACM SIGPLAN SIGOA symposium on text manipulation*, 16(6):147–155.

Stallman, R. (1987) *GNU Emacs manual.* Free Software Foundation, Cambridge, MA, March.

Suchman, L. (1985) Plans and situated actions: The problem of human-machine communication. PhD Thesis ISL-6, Intelligent Systems Laboratory, Xerox PARC Research Center, Palo Alto, California.

Symbolics (1985) *User's Guide to Symbolics Computers, Volume 1.* Symbolics, Inc.

Teitelman, W. and Masinter, L. (1981) The Interlisp programming environment. *IEEE Computer*, 14(4):25–34.

Thimbleby, H. (1980) Dialogue determination. *International Journal of Man Machine Studies*, 13.

Trevellyan, R. and Browne, D. (1987) A self-regulating adaptive system. In *Proceedings of the ACM SIGCHI+GI 1987 Human Factors in Computing Systems and Graphics Interface*, pages 103–107, Toronto, April 5-9.

Unipress (1986) *UniPress Emacs screen editor — user's guide.* Unipress Software Inc, Edison, NJ.

Witten, I. (1982) An interactive computer terminal interface which predicts user entries. In *Proc IEE Conference on Man-machine Interaction*, pages 1–5, Manchester, England, July.

Witten, I., Cleary, J., and Darragh, J. (1983) The reactive keyboard: A new technology for text entry. In *Proc Canadian Information Processing Conference*, pages 151–156, Ottawa, Ontario, May.

Witten, I., Cleary, J., and Greenberg, S. (1984) On frequency-based menu-splitting algorithms. *International Journal of Man Machine Studies*, 21(2):135–148, August.

Witten, I. and Greenberg, S. (1985) User interfaces for office systems In Zorkoczy, P. (1985), editor, *Oxford Surveys in Information Technology, Volume 2*, pages 69–104. Oxford University Press.

Witten, I., MacDonald, B., and Greenberg, S. (1987) Specifying procedures to office systems. In *Automating Systems Development Conference*, Leicester, April 14-16.

Xerox (1985) *The Interlisp-D reference manual - Environment, Volume 2.* Xerox Artificial Intelligence Systems, April.

List of Figures

Example Event List
9 mail ian
10 emacs fig1 fig2 fig3
11 cat fig1
12 diff fig*

Examples and Results of History Uses		
User Input	**Action**	**Description**
!!	diff fig*	Redo the last event
!11	cat fig1	Redo event 11
!-2	cat fig1	Redo the second event from last
!mai	mail ian	Redo last event with prefix "mail"
!?ian?	mail ian	Redo last event containing the string "ian"
!! fig3	diff fig* fig3	Append "fig3" to the last event and redo
^diff^page	page fig*	Substitute "page" for "diff" in the last command
!!:p	diff fig*	Print without executing the last event
page !10:1-2	page fig1 fig2	Include the 1st and 2nd argument of event 10 and redo

Figure 1: Examples of the UNIX *csh* History Mechanism in use

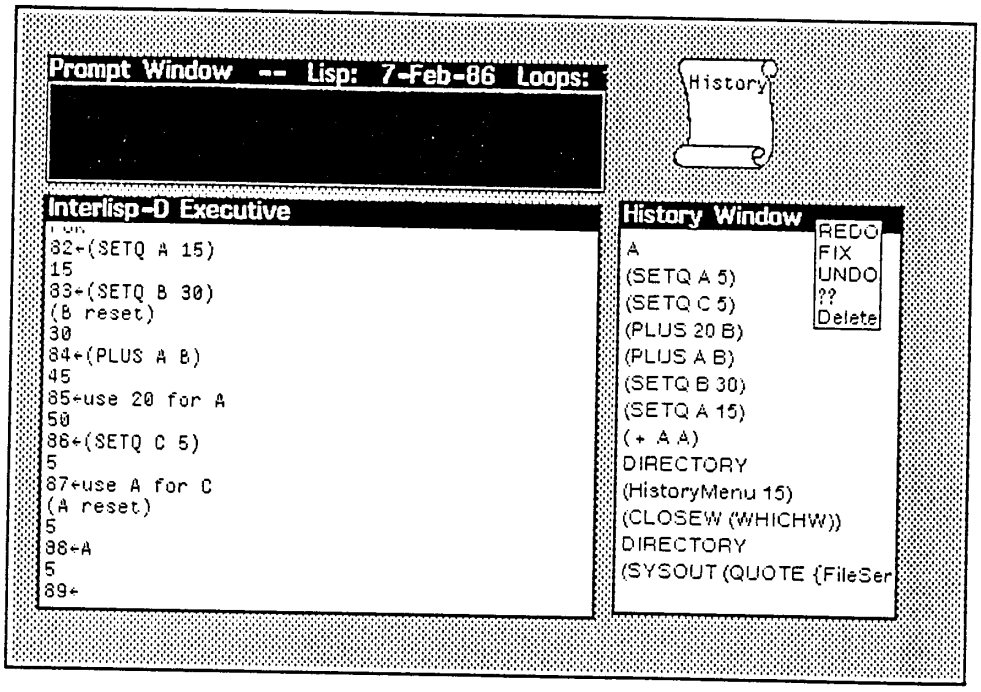

Figure 2: A portion of the INTERLISP-D environment, showing HistMenu in use

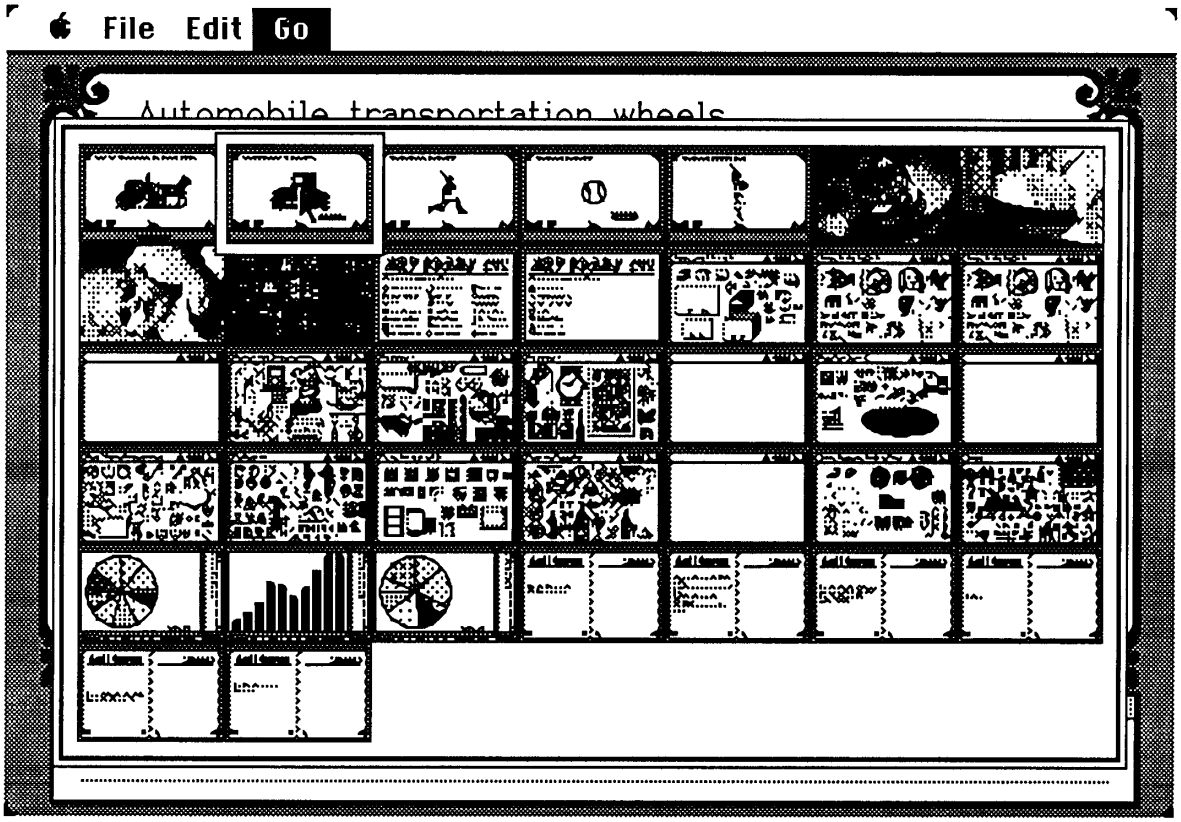

Figure 3: The HyperCard *Recent* screen

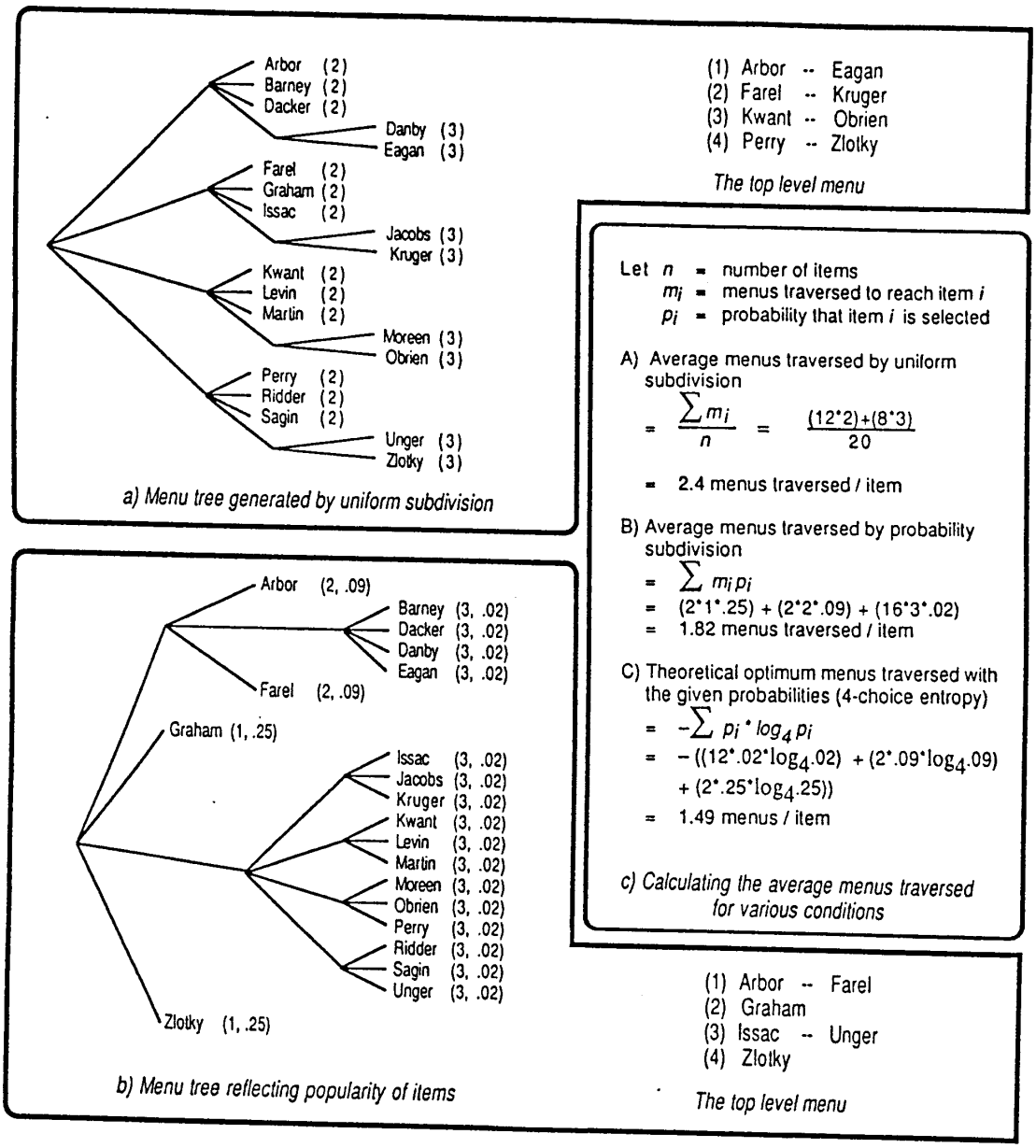

Figure 4: Menu trees generated by uniform and probability subdivision

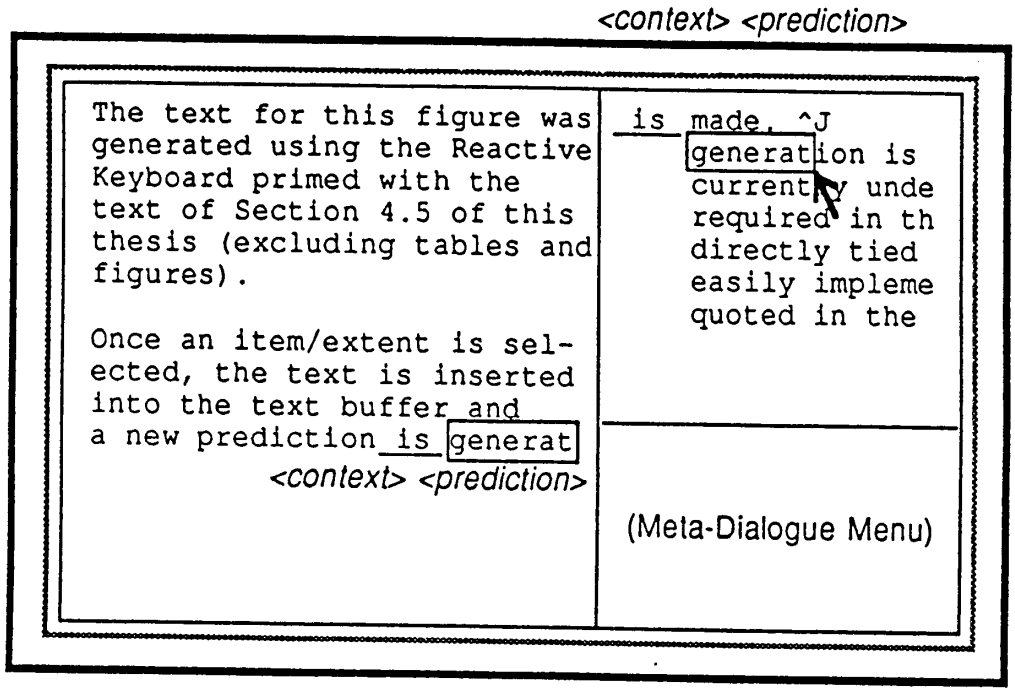

Figure 5: *RK-pointer* menu and feedback, from Figure 4.5 in Darragh (1988).