Yet Another History Mechanism for Command Interpretors

Francis C.M. Lau
Atul Asthana

Computer Communications Networks Group
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

1. INTRODUCTION

User level software has two main components, namely, functionality and form of presentation. Functionality refers to the facilities provided by the software, while form refers to the way in which these are presented to the user (user interface). These two components ultimately decide the "acceptance" of the software and need to be "balanced" with respect to each other, eg. a powerful application may loose or obscure much of its functionality if the user interface is not designed with care.

In this paper we discuss the requirements of a history mechanism with respect to the "functionality" expected of it and the "form" in which it is to be presented. Finally, we present our design that is based on these requirements.

2. DESIGNING HISTORY MECHANISMS FOR COMMAND INTERPRETORS

For any design project one has to study the application, specify user requirements, translate these requirements into design criteria and finally make a software specification that meets the user requirements. In this section we outline the application and discuss steps for arriving at the design criteria.

2.1. Application (Functionality)

History mechanisms have been extensively used in data base systems to ensure that information is not lost [GRAY81], in network file-servers and distributed file systems for consistency and error recovery [MITC79, STUR80], in the design of text editors [HAMM81, LAMP76, STAL81, GOOD81], and in many other less obvious situations. The technique is to record/log an update of an object, consisting of the name of the update procedure and its arguments. Based on the log one can provide facilities for "error recovery", "undoing" a command, repeating a command etc.

At the command interpretor level only a few of these facilities are realistic. Consider the undo command, where the previous command can be undone. A system will have to maintain "old copies" or an "old state" of the machine at all times, which is not practical in most cases. Besides, certain actions can not be undone at this level, eg. printing a file.

In general, commands issued by a user tend to be strongly correlated and a user often executes more or less an identical set of commands, with or without modification. eg. a think → edit → compile → run sequence may be executed several times. If a mechanism were available to aid the user in this regard, a user would reduce repetitive work and would save on typing Using a history mechanism, a user should be able to:

- Recall previous commands for viewing
- Repeat a command (or a set of commands) without changing any arguments.
- Modify a command (or commands) and execute it.

2.2. Model

The history mechanism is modeled as having two axes The horizontal axis contains the text of a single command (or a set of commands keyed in on the same line) The vertical axis corresponds to the time axis and contains the sequence of commands as they are issued by the user in time.

To modify a previous command(s) the user first selects a command or commands from the vertical axis. Then, within the command one selects one or more arguments - (1) word, (2) pathname, (3) character, or (4) some other defined object. Now, the possible actions are: (1) replace an object, (2) add an object (at the front, at the end, or

in the middle), and (3) delete an object (same as replacing it with a null argument).

An object can be selected in one of three ways

1. *Specify the object by content (or context)* - for example, 'fred'.
2. *Specify the object by position* - for example, the 5th object.
3. *Specify the object by pointing at it** - that is, move the cursor to the object in question and select it.

For the vertical axis, similar actions apply; however, the objects in question are events. Note for (1) and (2) we can specify (implicitly or explicitly) multiple objects simultaneously. The selection can be absolute (eg. line in which pattern was matched) or relative (eg. the line after the line in which the pattern was matched).

2.3. Users' Requirements

We took samples from a from a local user community** to compile a list of "desirable" qualities for a history mechanism. These requirements are listed in decreasing order of importance to the user. We have a small discussion with each point that identifies approaches to meet the requirements.

2.3.1. Easy to Learn and Easy to Use

People are already accustomed to conventional command input (generally true for most systems, except UNIX*** to some extent) and may not be naturally willing to learn something new, unless it is very helpful and/or easy to learn and use.

This requirement translates into providing a small repertoire of symbols, for both objects (selection) and operators. The symbols should reflect its meaning as far as possible. The premise behind the approach is that it is easier to perform complex functions by combining a relatively small set of primitive objects than to choose from a much larger set of commands.

Another consideration is the choice of keys employed. For any command we should keep the number of keystrokes small as well as minimizing the number of different keys hit, using as many default options as there are available. This minimizes command size and simplifies the interface.

2.3.2. Natural and Consistent

A user will use a system if it is natural to use (command syntax) and if it is consistent within itself and its environment. A user should be able to compose and execute naturally, and conveniently a command after studying the symbols.

The syntax chosen is of the *object-verb-modifier* form where one selects the objects before specifying the operation. Consistency with the environment can be provided by a correct choice of symbols, in that they do not clash with their use in other system facilities or at least they are similar in meaning. Consistency within the mechanism is improved by the lack of special cases. Specia cases may improve speed of executing a command but are unnerving to a casual user.

2.3.3. Fast

A user will only use those facilities that help him in getting a thing done easily and faster than other means. In this application it means that editing should be faster than retyping. Hence, symbols are bound to a single key that is convenient to use.

2.3.4. Reliable and Robust

The user expects the mechanism to be forgiving, in that a user should be allowed to prevent and correct mistakes. After a long or complicated editing he may not anticipate the consequences of the actions he performs, thus leading to feelings of tension and uncertainty.

Confirmation is provided for indicating the effect of a command, to the user. It is helpful for long edit's or before issuing a dangerous command, but can be annoying for simple or short commands.

2.3.5. General, Flexible and Powerful

Most users issue simple commands and some systems provide facilities for command script (eg. makefile in UNIX) thus they do not require excessive power or flexibility. Advanced users always constitute the minority but they should not be frustrated by the lack of features

We provide an extensible system, through a profiling and an aliasing mechanism. Profiling allows a user to "tune' the environment to suit his purpose, while aliasing allows him to construct keystroke macros and bind them to simpler keystrokes. Thus an advanced user makes higher level constructs once and uses them just as if the mechanism had provided them.

---

* In general, this feature requires cursor control capability.

** Computer Communications Networks Group, University of Waterloo

*** UNIX is a trademark of Bell Laboratories, Inc.

3. HISTORY MECHANISMS

We present, in moderate detail, one existing history mechanism and our proposal. We evaluate each of them with respect to the above criteria and finally point out directions for future mechanisms.

3.1. Csh's History Mechanism [JOY83a]

Csh, running on UNIX, is the first implementation of a command language interpretor incorporating a history mechanism. It consists of a list whose size is controlled by the history variable (set in the user profile). History substitutions reintroduce sequences of words from these saved commands (log) into the input stream. This allows a user to repeat a command, reuse arguments from a previous command, or fix spelling mistakes in previous commands.

Selection is achieved both by context and by position, where position refers to the number of the "word" (argument). Selection in the vertical axis is specified by (1) event number, (2) relatively (-number), (3) by a prefix of a command, or (4) by a string contained in a word in the command (does not match across word boundaries). Within a line an object can be (1) one or more characters (within a word) or (2) one or more words. A total of 10 symbols are provided for selection and a symbol has meaning in the context of its use; eg. a number (decimal) refers to an event number if the context is selection of an event (command line) or to an argument number if the context is selection within the event. There are a host of special cases where keystrokes may be saved and a default selector is assumed for missing selectors.

There are a set of 10 operators that operate on the selected words. These allow the user to (1) search for string and substitute - {s/l/r/}, (2) remove trailing pathname, leaving the head - {h}, (3) remove trailing '.xxx' component, leaving the root name - {r}, (4) remove all but the extension '.xxx' component - {e}, (5) remove all leading pathname components, leaving the tail - {t}, (6) repeat the previous substitution - {&}, (7) apply the change globally, prefixing the above, eg. 'g&' - {g}, (8) print the command but do not execute it - {p}, (9) quote the substituted words - {q}, (10) like {q}, but break into words at blanks, tabs and newlines - {x}.

History substitutions begin with the character '!' and may begin anywhere in the input stream (nesting is not permitted). The '!' may be preceded by the escape character '\' to prevent its special meaning. All selections and operations are preceded by ':', the delimiter.

3.1.1. Evaluation of the History Mechanism

We feel that this mechanism is not easy to learn as it has a a relatively large number of symbols to be learnt and special cases to be studied. Command naming is not natural for some operations (eg. h, r, e, x, etc.). This may keep him from using those commands. The mechanism is easy to use for simple operations, eg repeat a command, use all the arguments of a previous command, etc., but in more complicated substitutions the syntax is cumbersome. One can avoid the complications if he can remember all the special cases, eg. the '! seperating the event specification from the word designator can be omitted if the argument selector begins with a '|', '$', '*', '-', or '%'.

The selection procedure is quite natural. The problem of consistency arises with the special cases. These are neither natural or consistent and may result in confusing a user.

The mechanism is helpful to the user in reducing his typing burden specially for simple substitutions. A user conversant with all special cases is provided a powerfu tool but an average user is not able to use all the facilities to an advantage.

The facility of confirmation is provided to the user on a per command basis. If confirmation is requested, a command is logged as the last command and is printed. To execute it one has to type the code for repeating the last command.

The mechanism is very powerful and flexible. The user has tremendous choice in his approach of modifying a command.

One of the features not supported by this mechanism is that of executing a set of commands. The vertical axis is only used for selection of either full commands or a set of arguments from different commands.

3.2. A New History Mechanism - Modification by Position

Csh's provision for substitution, we believe, is too powerful but inefficient for general use. It is easier and faster in most cases to retype certain string than to correct some character in it. In this new mechanism, the smallest object to be referenced is a word (not a character) and selection in the horizontal axis is achieved only by position (not by content). A word here is either an argument of a command or a pathname component. For modification of a single command (one horizontal entry in the history), we provide operations to: (1) match a word, (2) delete a word, (3) replace a word, and (4) add a word (at the front, in the middle, or at the end). In order to minimize the number of keystrokes, the default is to include all words (ie. a '*') that are not explicitly

matched, deleted, or replaced.

3.2.1. Command Structure

An history invocation has the following general structure:

! *command-selector* ▽ *command-modifier*

We use a '▽' whenever a space is to be emphasized. We shall first discuss the command-modifier. The basic designators for operations are:

Match

.	matches a word
*	matches zero or more words

Delete

-	deletes a word
=	deletes zero or more words

Replace

word	matches and replaces a word with *word*

Add

[	starts addition
]	ends addition

For pathnames

{ }	matches a pathname
{*modifier*}	a word is now a pathname component, for all the above operations

Note that matching (ie. '.', '-', '{ }') occurs from left to right (the natural way) and is left-justified. On the other hand, '*' and '=' match the maximum possible number of words; however, when there are two or more conflicting such characters (eg. '=={}=') the leftmost one is given priority. A pathname is any word that has a '/' in it. Therefore '/tmp/write.c' is a pathname and 'write.c' is not. '/' is considered a pathname component; thus, '-lsys/termlib' is composed of the pathname components '-lsys', '/', and 'termlib.' Any symbol can be escaped with '\'. As a result, there are altogether ten special symbols used by this mechanism: '.', '*', '-', '=', '[', ']', '{', '}', '!', '\'.

An example.

```
4    cd /u/sysdir/nextdir
5    vi +$ write.c
6    cc -O -I/usr/curses write.c -lsys/termlib
7    pr -i12 write.c | lpr &
8    a.out
9    cd
```

Match

!5	equals '!5 *' which matches the entire event 5 and repeats it

Delete

!6▽*-	deletes '-lsys/termlib' from event 6
!6▽-	deletes 'cc' from event 6!
!cc▽.-	deletes '-O' from event 6
!pr▽...=	deletes '| lpr &' from event 7
!6▽{.=.}	deletes '/usr/' from event 6 (remember that matching is left-justified)

Replace

!5▽*read.c	replaces 'write.c' to 'read.c' in event 5
!pr▽p	changes event 7 to 'p -i12 write.c ...'

Add

time▽!cc	produces 'time cc -O -I/usr/curses ...'
!cc▽*[▽r.c -lx▽].	adds the arguments in brackets after 'write.c' in event 6
!4▽*[.bak	changes 'nextdir' to 'nextdir.bak in event 5
!cc▽*{.[/lib	changes '-lsys/termlib' to '-lsys/lib/termlib' ('*' here matches the maximum number of words and therefore includes the first pathname in the event)
cb▽<!5▽=.]▽>out	produces 'cb <write.c >out'

Next, the command-selector. Here, all we need is matching, that is, matching a command (event) prefix (by content), an event number, or an event position. To be consistent, the semantics of the symbols '.', '*', '-', and '=' are more or less preserved (the latter two do not actually delete the event(s) from the history list, they do a "non-match"). The "fill character" is now '=' instead of '*', therefore everything that is needed has to be explicitly matched. Matching is right-justified (since most recent

events are more likely to be selected), that is, if we view the vertical axis as (refering to previous example)

4 5 6 7 8 9

matching will try to occur as close to the right end as possible (unless reverted by '=' or '*' which matches the maximum number of items). Some examples will show how commands are selected (refering to the above history list):

!*	selects all events (ie. 4 to 9)
!=	selects the null event
!c	matches event 9 ('cd') and selects it
!cc	matches event 6
!*pr	equals '!*pr=' and selects events 4 to 7
!5*a.out	selects events 5 to 8
!-.=.	selects events 4, 6, and 9
!-.=	selects events 4 and 6 since the '=' would match the maximum number of events
!.	selects the previous event
!--	selects the third previous event

Note that a modifier following a selector would apply its actions to all the selected events. Therefore, commands like

!vi*pr▽*read.c

have to be cautiously issued. Of course, the one we show here will not replace all 'write.c' with 'read.c'.

To provide some safeguard when number of events are to be repeated, the user may choose to be prompted for modification for each event in turn. This is done by entering just the command selector and the space (without the space, the selected event(s) will be executed right away without any modification); the history mechanism would then print each event (or "the event" if only one event is selected) and wait for a modifier from the user terminal (terminated with carriage return) which is to be applied to this current event. This option also facilitates the user to "visualize" the changes that he is going to make.

While going through the list of events prompted by the system, the user might want to skip over certain events or he might decide to stop the process altogether for the rest of the list and return to the shell level; to do this, the user hits <rubout> for the former and <break> for the latter. The same applies to a singly-selected event, and in this case any one of these keys will do.

3.2.2. Options

We provide optional features that make the system flexible enough to accomodate users' diverse needs. These options settings are given in a file called '.histrc' and allow the user to tune the history environment to its best. These options include:

- *Size of History.* The default is one.
- *Aliases.* Some often-used keystrokes can be aliased. For example,

alias $!. =.]	makes $ the last argument of the previous event
alias PP !. =\{\}=]	makes PP the last pathname of the previous event; to select the first pathname is not possible since the left '=' is defaulted to be stronger.

Here are some examples of their use:

echo !$	echos last argument of previous event; note that the first '!' is needed for history invocation, and the second '!' will select the event.
!24 *$	replaces last argument of event 24 by last argument of previous event
grep if !PP	which is obvious

The user can use any combination of characters for aliases and it is his responsbility to choose the appropriate combinations so that no confusion may arise. The history mechanism will, at invocation time, find all aliases and do simple expansion on them. In case of both 'P' and 'PP' are aliase names, the latter will be expanded.

Confirmation. If this option is set, all history modifications have to be "passed" by the user - by hitting a carriage return after the modified command is displayed on the screen. By hitting the <rubout> key instead of the carriage return, the command will be discarded.

Static Events. Instead of having each modified event to be added to the end of the history list like a new command, the user may choose to have the event modified "in place." This will tend to keep the history small and the user doesn't have to worry about "losing track of previous commands." However, in this case, the vertical axis is no longer a time axis. Further options in this respect may perhaps allow the command to be

"extracted" and placed as the last command (the space previously occupied is recovered).

Saving History between Sessions. When the user signs off, the history list is saved in a file which is retrieved and revived as initial history when the user starts a new session.

3.2.3. Evaluation of the New Mechanism

We claim that this history mechanism satisfies most of the criteria we presented in previous sections, ie. easy to learn, easy to use, consistent and natural, fast, robust and reliable, flexible, and powerful. The one major disadvantage of this scheme is in dealing with long commands. For example, to replace a word which is at the 50th position in a 100-word event is inefficient and cumbersome. If there happens to be some nearby pathname, perhaps the user can first position to this pathname and then fill in rest of the dots. However, we do not think long commands like this would appear too often (in our UNIX's here, the longest commands exist in makefiles).

4. CONCLUSIONS

With the current popularity of pointing or positioning devices future history mechanisms would most certainly employ graphical techniques. These mechanisms can be classified into those that use cursor capability with text driven command interpretors, and those that use a menu driven interpretor.

A history mechanism using cursor capability has the same functionality as the on we discussed. The difference would be in the way it is presented. A great advantage with cursor capability is that the user now sees changes instantly. To indicate the scope of selection (vertical) relevant commands can be displayed in the editing window. Editing functions made available should be a small subset of those available in screen editors, eg. Vi [JOY83b]. Such a mechanism would provide an easy to use, powerful and consistant environment, with inbuilt confirmation.

A mechanism for a menu driven interpretor has inherently different requirements. Such an interface generally leads the user down a menu tree after each selection (a hierarchical menu structure) and a user does not make "mistakes". A history mechanism can be helpful if it helps one in maintaining a log of previous commands, repeating a command or at least descend to a certain level in the tree, for the execution of another command.

5. REFERENCES

GOOD81 Good, M. "Etude and the Folklore of User Interface Design." *Proc. ACM SIGPLAN SIGOA Symp. on Text Manipulation,* Portland, Oregon, Jun 1981. 34-43.

GRAY81 Gray, J. et. al. "The Recovery Manager of the System R Database Manager." *Computing Surveys,* 13, 2, Jun 1981. 223-242.

HAMM81 Hammer, M. et. al. "Etude: An Integrated Document Processing System." *Proc. of the 1981 office Automation Conference,* AFIPS, Mar 1981.

JOY83a Joy, W. "An Introduction to the C Shell.' *UNIX Programmer's Manual,* 4.2 BSD, Aug 1983.

JOY83b Joy, W. and Horton, M. "An Introduction to Display Editing with Vi." *UNIX Programmer's Manual,* 4.2 BSD, Aug 1983.

LAMP78 Lampson, B. W. "Bravo Manual." *Alto User's Handbook,* Xerox Palo Alto Research Center, Palo Alto, Calif., Nov. 1978.

MITC82 Mitchell, J. G. and Dion, J. A. "A Comparison of two Network-Based File Servers." *CACM,* 25, 4, Apr 1982. 233-245.

STAL81 Stallman, R. M. "EMACS, the Extensible, Customizable Self-documenting Display Editor." *Proc. ACM SIGPLAN SIGOA Symp. on Text Manipulation,* Portland, Oregon, Jun 1981. 147-156.

STUR80 Sturgis, H. E. et. al. "Issues in the Design and Use of a Distributed File System." *Operating Systems Review,* 14, 3, Jul 1980 55-69.