1987

Shells in an Interactive System

Balachander Krishnamurthy

Report Number:
87-707

SHELLS IN AN INTERACTIVE SYSTEM

Balachander Krishnamurthy

CSD-TR-707
September 1987

Shells in an Interactive System

Balachander Krishnamurthy

Department of Computer Sciences
Purdue University
W Lafayette, IN 47907

September 7, 1987

Abstract

In considering the process of interaction between the user and the machine, a frequently used communication medium is the command interpreter which mediates between the user and the operating system. It provides access to all the utilities in the system—revision control systems, time based schedulers, compilers, mail handlers, and other application programs. We confine our interests to command interpreters in an interactive system where the window system is the front end to the various application programs and the command interpreter is an application program that the user can access. We seek to replace the traditional command interpreter with a shell that permits the user to specify frequently executed actions in a simpler manner. By looking at the most frequently executed commands we can see what commands can be eliminated in the environment of a workstation with advanced input devices and display techniques. If a command cannot be eliminated or replaced we seek to provide an improved interface both globally and specific to a command. Another motivation is that the interface remain *user-oriented*. An aid towards providing a user-oriented interface is dynamic tracking of a user's pattern of usage. By keeping track of a user's usage pattern we can build an interface that evolves with and is specific to the user.

1 Introduction

In this paper, we consider the role of a shell in an interactive system. Throughout this paper we will treat the term shell as in the UNIX sense, that of a program that provides a front end to the various operating system capabilities usually at a higher level. By an interactive system we mean a software subsystem that usually resides on a workstation connected to a network of hosts that run traditional operating systems like UNIX. An interactive system comprises of input devices, a window manager, and a set of application programs. The window manager demultiplexes the input generated by the user via the devices and directs it to the application programs which are processes running in distinct physical regions called *windows*. The output from these processes is also multiplexed and displayed by the window manager in the appropriate window. Our model of interaction in an interactive system is explained in a companion paper[6].

The remainder of the paper is divided into eight sections: first, we provide some historical background to shells in the UNIX environment. After a look at the syntactic aspects of the shell we see what portion of the shell has been moved into the window system already. In the next section, we provide the motivation for our argument to move away from the current notion of shells in distributed interactive system. We then discuss our model of a shell. This is followed by a discussion of our modifications to an existing command interpreter and the analysis of the usage data gathered. We conclude with a look at our novel history mechanism implemented under a prototype window system.

2 Motivation

We provide some historical background on the shells available for the UNIX environment and look at the current role of shells.For the UNIX operating system, there are three popular command interpreters. The first one was the *Bourne-shell* that was designed by Steve Bourne [3]. The next shell that gained popularity was Bill Joy's *C-shell*, originally written in 1978. Recently another shell has been developed at Bell Laboratories—the *Korn-shell* [5], designed by David Korn. While the functions of these command interpreters are similar, the features vary: *C-shell* for example has a *history*

mechanism, by which a user can keep track of the commands he has issued and re-execute a previously issued command. *C-shell* also has job control so that a user can suspend a running task and move it to the background. The *Korn-shell* is similar to *C-shell* in that it has an history mechanism ad job control, but retains the flavor of the *Bourne-shell*. Also, the *Korn-shell* features an editing interface. A feature comparison of the three command interpreters can be found in [7].

Command interpreters in the UNIX domain are traditionally used for creating processes, redirecting input and output, and setting environment parameters. The three command interpreters are similar to each other in the sense that they parse the command line and pass it to the underlying operating system to be executed. Our objection is that, while such an interface is fine on ordinary terminals, it can and *should* be superior on workstations with bitmapped displays and pointing devices for input. Our observation was also that only a small subset of commands are being used a vast majority of the time and we could use this knowledge in designing a better interface to the commands. If we are able to gather user-specific information we would be able to construct an user-oriented interface.

3 Syntactic aspect of shells

As mentioned earlier the shell's role is to provide access to all the utlities in the system such as revision control systems, time based schedulers, compilers, mail handlers, and other application programs. The mechanics involve a syntactic component and we consider that first. At the highest level, the shell accepts command input from the user in an interactive fashion (asking for correct input or offering corrections), and returns the output of execution of the command. The user can find out state information – about his files, his processes, the network, other users etc. After parsing the command line the shell creates a process to execute the requested command. The user can use meta-characters as a short cut to specify one or more arguments. Meta characters are not exclusive to the shell though they are found in shells more commonly than in other utilities. Example of meta characters are '?', which matches a single character, and '*' which matches zero or more characters. UNIX pipes—a mechanism by which output from processes can be sent as input to other processes, are made available to the

user via the shell. The user can combine the output of various commands through the pipes. Arguments can be specified as part of the command line. Tasks can be redone in the shell with different arguments.

4 Why move away from present notion of shell?

- Shell is not a context specific tool.
- Input to the shell is restrictive.
- Presenting uniform output is difficult.
- Commands and arguments to shells are specific to the application program.
- Shells are processor specific.
- There is no provision for graphical command input.
- Moving towards application specific tools lessens dependency on shell.

A shell, intended to provide a uniform front end to various application programs is not an application specific tool. The various commands that it offers results in a loss of uniformity in interface. Grafting traditional shells on to modern window systems without any modifications to its interface capabilities is not an improvement. The ability to specify files as input is no longer sufficient. Operations on chunks of text, or windows are frequently performed in interactive systems. For example, the user may wish to sort the information displayed in a window or ensure that it is spelt correctly. Neither of these are traditional editing operations. Operations like sorting and spelling have been traditionally done on larger and more permanent objects like files via the shell. An example of a window system equivalent of such operations is Emacs' *filter-region*, which permits marking a region of text and filtering that region through a shell command, such as *sort* or *spell*. The granularity of the argument to traditional shell commands has thus been altered.

The next major drawback of shells in a distributed interactive system, after input, is that of output. Once again, the mono-dimensional world of the shell, prevents presentation of the output in different ways depending on the application. Further, the output domain of all processes created under the shell is the same shell region. Flexibility in deciding where and how output should be displayed is lower. Merely having the ability to redirect output to a device or a file is not enough. For example, it should be possible to send the output to another window. Objects such as windows can be treated as arguments to shell commands [1]. The output of two processes created simultaneously can become interspersed. While pipes provide an elegant way to filter the output of one process through another process, there is no mechanism to show intermediate output in the pipeline. It should be possible to have a graphical interface with the results of different stages of the pipeline being displayed. The inability of the shell to display intermediate stages of the pipeline is a result of the shell being a one-in-one-out mechanism. Commands, even when combined, produce only a single final output, though the same output can be sent to two different channels simultaneously.

The interface to each of the commands, as far as flags, options, arguments are concerned is left to the individual command. The shell merely parses the command line and transfers control to the function that implements the command. Individual application programs have their own notion of how an individual command line should look like and how it should be parsed. Some programs prompt the user for proper input if he entered incorrect arguments, while others do not. Even though the user is conversing with a single program—the shell, he has to speak different languages. Unifying the interface alone is not enough—the user needs to be able to impose his view of interaction at every step.

Shells do not provide the best interface for all applications. For example, to play board games like *chess* and *othello* on the machine, the user would traditionally input moves by typing them in—an awkward and tedious interface. However, with the advent of bitmap displays and pointing devices it has become significantly easier to provide a friendly interface. For example, the user can just point to an object on the board and move it. The seeing and pointing paradigm of interaction is a simple interface model as expounded in [10]. Shells are unnecessary for such applications.

In a distributed interactive system, we detect a clear trend towards divesting the shell of some of its traditional functions. For example, there are application specific tools such as *mailtool*, *dbxtool* in interactive systems such as SunView [11]. Traditionally, the shell provided the front end to the mail subsystem and the debugger. Separate interfaces have been developed for these subsystems, complete with a mouse interface. In the past, users could either enter the mail subsystem and perform their mail transactions or intersperse other shell commands with mail subsystem specific commands as in MH [8]. With debuggers, there has always been a notion of *entering* a debugger and exiting it after finishing the debugging. In *dbxtool*, the tool is invoked and a new file can be read in. Also, the file being debugged can be modified rather than exiting and re-entering the debugger.

The move toward application specific tools such as *mailtool*, *dbxtool*, apart from providing separation of context, eliminates the need for the application specific commands in the shell. For example, a window that exclusively deals with a time based scheduler relieves the need to converse with the shell. A window dedicated to a particular application enables monitoring of its status at any instant without having to run a shell command. A separate context provides a more coherent view and the interface to these application specific tools can be tailored to the user.

5 What needs to be changed?

We will see how several of the features of the shell have been rendered less useful and how an alternate interface can be provided by our view of a command interpreter. Let us consider job control. The *C-shell* provides job control by which the user can *suspend* his current process, move it to the background and presumably execute another command. Alternately, he can just stop a process that has been running. With the advent of window systems, job control can be eliminated to a reasonable extent. With each process potentially having a window for itself, the user can alter the z-coordinate of the window and push it behind in the stack of overlapping windows on the screen. To continue the analogy, the *notify* mechanism of the shell, by virtue of which the user is notified of the completion status of the process, can be displayed via changing icons. An example can be found in the Sapphire window system of the Spice [2] environment where various

stages of the process are displayed via icons.

Another common use of the shell is to manipulate files. The *dired* package of Emacs points us in the direction of treating a collection of files uniformly. In *dired* the user can edit his directory in a buffer where the contents of the directory are displayed. Files can be created, deleted, or edited. While *dired* does not completely replace the file manipulation capabilities of the shell (for example, there is no way to manipulate files in another directory without reading the entire contents of that directory into a buffer), it is a step in that direction.

We infer that we have to provide a direct interface to several packages. Each of the packages that we divest from the shell and provide a direct interface to, may converse with the shell without the user's knowledge. By separating the individual packages from the shell we provide a clear context to the user. The user is aware that his input is being sent to a particular application program and the output appearing in that window is the result of the command he issued to that application program. Such an interface is not possible in the traditional view of the mono-dimensional shell. Approaches that force the user to input his commands in one place and watch the results in some other area on the screen are potentially distracting.

While it might seem that divesting the shell of access to application programs may make it harder for the user to interact with the application programs, the contrary is true. The users can take advantage of the workstation and the resident software rather than be limited by the shell built for an ordinary glass terminal. In the next section we expand on our model of the shell.

6 Our model of a shell

Having pointed out the deficiencies in the existing notion of a shell in window systems, we now present an alternate scheme that attempts to answer most of the questions raised in the earlier sections. We also discuss actions for which there is a need to retain some of the traditional shell notions.

It should be possible for the users to access all the services of the distributed system via the same shell to ensure transparency. Our approach is to gather all the shell services that can be provided in the distributed sys-

tem and make them available to the user through a single shell. We do this by viewing the shell to be an application program in the interactive system. We gather all the valid commands accessible via the shell and construct a table of the commands. As several of the application programs have their own interface, the number of commands that remain to be accessed solely via the shell is reduced, though the number remains large. A completion table is formed from the list of valid shell commands, requiring the users to type only the least disambiguous string. The arguments, options, flags of the shell commands can be used to prompt the user. As all the valid shell commands have been gathered, a uniform front end can be provided to the commands. Such a view of the shell is consistent with our model of interaction, whereby we gather as much information about the application program and download it into the window system.[6]

Providing menus for the commands has its drawbacks. One is the large number of commands that remain valid in the shell, in spite of divesting the application programs from it. If the menus are of the pop-up variety, their lifetime is usually a single mouse click as they appear (say) on the down transition of a mouse button and disappear on the up transition. If the menus are permanent, they take up valuable screen real-estate. We can get around this problem in two ways. Several window systems already provide a way to issue commands without displaying them on the menus. The technique is called *hotspots*. The hotspots are disguised commands. For example, on the Macintosh window system, there are hotspots on the corners of the windows, that permit the user to alter the size of the window, to delete or move a window. Hotspots replace the shell level commands such as *remove* and *move*. Scroll bars obviate the need for paginator programs that are usually provided with shells to browse through files. By combining the display of a file in a window with the availability of scroll bars, we have done away with file display commands in the shell. The advantage of displaying the file in a window is that the display is not transient, as the user can move around the displayed text at any time.

For an implementation of our alternate approach we divide the commands accessible via the shell based on the nature of output of the commands. Some commands have no output, some have a small amount, while others have varying amounts of large output. Commands that have no output are handled via the hotspots technique described above. Some commands may force the user to confirm and the confirmation is done via

another mouse click or a keystroke. Commands that have small amounts of output could have the output displayed in the same place as the command themselves, as in existing shells. Commands that have large and varying amounts of output have windows of their own. For example, the command that shows the load average on a cluster of machines can have a window of its own and constantly redisplay the status. The classification is not meant to constrain the commands, but to understand the nature of commands and provide an appropriate execution and display environment for them.

As our target environment includes graphics capability, a bit-mapped display, and pointing devices such as mice, we have to design a shell that will take these factors into account. In comparing the ordinary terminal world and the world of workstations, we notice that the granularity of the arguments to commands are different. In the latter we have *regions* of characters and contents of windows as potential arguments, apart from files or directories. Also there is multiple continuous display possibility, ability to dedicate windows to hold output and history of past commands. The display itself can be partitioned based on individual commands and their output, all in their own windows. The output of previous commands or a part of it can be used as input to further commands. Windows permit the commands and output to be displayed contiguously and provide the same continuity that a single window shell provides. Operations like sort and spell can be run on smaller granularity objects such as regions. In some sense we are integrating traditional shell services in the interactive system environment.

We wanted to base our design of the interface to the shell on the individual user's usage pattern. We contend that only a small percentage of over 300 commands available in *C-shell* are used frequently. This fact has been borne out by previous studies [4,9]. In [4] the full paths of the commands executed on the user's behalf were not studied. In neither of the studies were the raw command lines used, primarily because the information was extracted from data already being gathered by the system (e.g. via the system accounting program *sa*). As we were interested in exactly what the user typed to the shell, we altered the shell to log the command lines. We also logged the full path names of the commands executed. The details of the modification to the shell and the data gathering are examined in the next section.

7 Implementation

As the heaviest used shell at Purdue is Bill Joy's *C-shell*, we decided to obtain information about it's usage. Previous information gathering attempts about the usage of a command interpreter have been limited either to the nature of the commands issued [9] or the existing information in the system itself. For example [4], uses the information that UNIX gathers from the system accounting program *sa*. Instead, we decided to modify the *C-shell* command interpreter slightly to log the raw command lines as well as actual command usage dynamically. We ensured that the modifications did not affect the users of the command interpreter in any way.

The logging of the modified command interpreter usage was done on different environments. The modified command interpreter was implemented and data gathered on machines across several administrative domains within the Purdue University campus, as well as in an industrial site, Gould Corporation in Champaign, Illinois. On the Purdue campus, the users included faculty, graduate and undergraduate students, researchers in the various departments, systems administrative staff, and secretaries. The industrial site mainly consisted of systems programmers and other software developers. In the rest of this section we will look at the precise changes made to the *C-shell* and explain how the data was gathered and analyzed.

7.1 Modifications to the shell

Before we embarked on modifying the shell, a survey of existing information gathering tools were made and found to be inadequate for our purposes. *C-shell*'s built-in history mechanism was a potential place to get information on exactly what the user typed. A trivial way to log the commands typed by the user would have been to use the built in history-logging facility of *C-shell* via the *savehist* mechanism. However, when the user issues aliases for commands, the aliases, rather than the alias-expanded commands are stored. Commands typed to the shell within control structures such as *foreach* are not placed in the history list. Also, history events, used for redoing a previously issued command with potential modifications, are not logged in the history list. Instead, the new command line formed as a result of the change is logged. Thus, there is no way to find out how many times

a previous command is being reissued or how many times a command was typed incorrectly. Our changes to the shell rectified all these problems, as we logged the raw command line typed by the user, including lines typed as part of control structures such as *while*, *switch*, and *foreach*.

Apart from the raw command lines, we were also interested in the precise *full path* of the commands executed based on the user's path. Commands that are executable from the shell follow a notion of *path* by virtue of which users could specify the order in which directories should be searched to find the right command. Information obtained from the system accounting program *sa* did not have the full paths of the commands that were executed on the user's behalf. We could not find any reference to work previously done on logging complete path names in the literature. Full path name logging ability is not built in to *C-shell* and no system accounting program maintains this information. Our modifications enabled keeping track of the full path of the commands executed on the user's behalf. The *C-shell* uses a bit vector scheme to hash the commands in each of the directories in the user's path. When a command is issued, *C-shell*, to find the appropriate binary to execute, uses a more expensive hashing function to see if the command *might* be in the ith component of the path vector. After this, a cheaper hashing function is invoked, once for each path component checked. As the command could hash to an incorrect path name, it was possible that an incorrect string would be executed. When the execution failed because of the lack of a corresponding binary in that directory, the process of searching through the path would continue. Executing a full path directly is cheaper, as the searches (and potential incorrect executions) are obviated.

Logging of full paths was complicated because of *C-shell*'s hashing mechanism. The full path command had to be logged just before invoking the *execv* system call as there is no return from *execv*. Success or failure of the system call is not known *a priori*. If the call failed because of the hashing function causing an attempt to execute a non-existent binary, the incorrect log entry had to be removed. A combination of two system calls *fstat*, which stored the location in the file where the new log entry was to be made, and *ftruncate*, which truncated the file to the previously marked position, were used to get around this problem. Apart from the *writes* to the log files for logging the commands, these were all the additions made to *C-shell*.

The actual modifications that were done was small—104 lines were added to the original source of 12762 lines of code—less than 1% addi-

tion. The modified shell had to be functionally equivalent to the regular *C-shell*. There was no noticeable difference in the functioning of the modified shell. The two extra atomic *write* system calls made by the shell caused negligible overhead compared to the amount of work the *C-shell* has to do to execute a command. The modifications were transparent enough that even regular and heavy users of the shell (the author included, though there might be reason to doubt his unbiasedness) could not detect any difference in the performance. The only real resource being consumed was disk space to store the logging information. We will look at the actual data gathering in the following sections.

7.2 Machine environment

The *C-shell* runs on practically all versions of the UNIX operating system and on a variety of hardware. Fortunately, Purdue University has a large number of machines running UNIX and thus we were able to install our shell on a variety of machines across several administrative domains. The machines ranged from single user workstations to large time sharing systems. The various administrative domains were chosen to ensure gathering of data from diverse user groups. Within the Purdue campus, the machines were chosen from those available in the Computer Science department, the university's computing center, and the Physics department.

The machines in the Computer Science department ranged from a VAX-8600 (Arthur) primarily used by the faculty in the Computer Science department, a VAX-11/785 (Gwen) used by students in the operating system and networks course, a VAX-11/780 (Ector) used by secretaries in the department, 2 VAX-11/780s (Blays and Merlin) used by students in the systems research groups, and another VAX-11/780 (Mordred) used by a large number of generic graduate students. All these machines ran 4.3 BSD version of the UNIX operating system. The shell was also installed on two file servers running Sun OS 3.0 and 3.2, serving several individual sun workstations (Sun-2/120, Sun-3/50, Sun 3/75), a Sun 3/75 (Surya) with a local disk. 113 users used the shell in the various machines belonging to the computer science department.

Beyond the Computer Science department, the shell was installed on a VAX-11/750 (Newton) also running 4.3 BSD UNIX, used by 90 researchers, graduate students, and faculty in the Physics department. The shell was

installed on a large Sequent Balance 21000 multiprocessor (S) running the Dynix operating system, with 735 users. The users were primarily undergraduate and graduate students in various computer science courses, though several other departments were also represented. Another instructional machine being used for an introductory computer science course complete with labs, (N) was a dual VAX 11/780 with 53 users. The shell was also installed on four other VAX-11/780s (H,I,J,K), all belonging to the computing center. While H, I, and K were used primarily by graduate and undergraduate students for coursework, J was a front end to the Cyber-205 Supercomputer. J is also used for systems programming. Another VAX-11/780 (L), used by researchers and graduate students in the Statistics department with 79 users was also part of our experiment.

Going beyond the academic environment we were successful in getting the cooperation of the Gould corporation in obtaining data from one of their development sites in Urbana-Champaign.† In Gould, the shell was installed on two Gould PowerNode dual 9080s (Mycroft and Fang) running UTX32 V2.0, each with 107 and 110 users respectively. A vast majority (about 85%) of the user community on the Gould machines were people involved in development software while fractions included support staff, technical writers, and other miscellaneous users.

In all 2761 users in 19 machines used our shell for a period of at least four weeks. The data gathered represented usage in a large campus and the large number of users lend strength to our conclusions. Table 1 describes the machines on which data was gathered. CS refers to the computer science domain and CC refers to the computing center which administered the corresponding machines. Table 2 displays the classification of users as well the number of users in each machine.

7.3 Gathering data

We first had to ensure that the changes to the shell were not critical as far as performance, upward compatibility, and functionality were concerned. We then had to address the issue of how much data had to be collected and the duration for which the data would be collected. Usage patterns of

†The author would like to thank Arden White's help in installing the author's modifications and the Gould corporation for permitting the data gathering.

Table 1: Machines, domain, model and Operating System

Machine	Domain	Model	Operating System
Arthur	CS	VAX-8600	4.3 BSD UNIX
Blays	CS	VAX-11/780	4.3 BSD UNIX
Ector	CS	VAX-11/780	4.3 BSD UNIX
Gwen	CS	VAX-11/785	4.3 BSD UNIX
Lionel	CS	SUN-3	SUN OS 3.2
Lucas	CS	SUN-2	SUN OS 3.0
Merlin	CS	VAX-11/780	4.3 BSD UNIX
Mordred	CS	VAX-11/780	4.3 BSD UNIX
Surya	CS	SUN-3/75	SUN OS 3.2
Newton	Physics	VAX-11/750	4.3 BSD UNIX
H	CC	VAX-11/780	4.3 BSD UNIX
I	CC	VAX-11/780	4.3 BSD UNIX
J	CC	VAX-11/780	4.3 BSD UNIX
K	CC	VAX-11/780	4.3 BSD UNIX
L	CC	VAX-11/780	4.3 BSD UNIX
N	CC	VAX 11/780(dual)	4.3 BSD UNIX
S	CC	Sequent 21000	DYNIX
Fang	Gould	PowerNode 6030	UTX32, V2.0
Mycroft	Gould	PowerNode 6030	UTX32, V2.0

Table 2: Number of users and classification

Machine	Users	Primary User Classification
Arthur	23	Faculty
Blays	8	Research
Ector	8	Secretary
Gwen	11	Student
Lionel	8	Research
Lucas	8	Research
Merlin	5	Research
Mordred	41	Student
Surya	1	Faculty
Newton	90	Faculty
H	482	Student
I	444	Student
J	115	Research
K	433	Student
L	79	Student
N	53	Student
S	735	Student
Fang	110	Industry
Mycroft	107	Industry

the machines vary with the user community and to ensure that the data gathered was a representative sample of the usage, we decided that data had to be gathered for a period of four weeks. We collected data continuously, ignoring peak activity. Logging was performed on login and interactive shells only, eliminating non-interactive and remote shell activities. The amount of data naturally varied with the machines and the number of users.

We gathered data for four weeks on each of the machines that the shell was run. As our shell was implemented on different administrative domains across the Purdue University campus, it was difficult to collect all the data simultaneously. However, the data was collected for the same duration. Rather than accumulating the data in several different places, we copied the data periodically to a single machine (Blays) and freed up the disk usage on all the other machines. At any given time no more than 3 Mbytes had accumulated on any of the other machines. Our experiment did not interfere with the users as the shell was fully compatible with the original *C-shell* and had no appreciable response difference.

The question of disk usage as far as logging to obtain better results with the shell is an open one. While it might be possible to do something similar to alias tracking of the Korn-shell [5], we came to the conclusion that the amount of floating disk usage because of the logging is small. If the workstations have a local disk attached, then the problem does not merit serious concerns at all.

7.4 Analysis of data

Over 40 megabytes of data was gathered in all and data compression techniques were used to store and analyze them. Our main purpose in gathering data was to verify the hypothesis that a small number of commands were being used frequently. This was proved beyond any doubt regardless of the class of users: secretaries, faculty members, researchers, and graduate students in other departments. The average number of commands that were executed over 85% of the time was around 15. (See Table 3). We combined the data within classifications such as faculty, researchers, students and looked at the most frequently used commands. Table 4 shows twenty of the most frequently used commands in each of our classification.

Table 3: Number of commands vs. percentage

Percentage	75	80	85	90
Arthur	14	17	21	27
Blays	14	16	20	25
Ector	11	13	16	20
Gwen	12	15	18	23
Lionel	8	9	12	15
Lucas	10	13	15	20
Merlin	11	13	17	21
Mordred	15	18	21	27
Surya	12	14	16	19
Newton	7	9	11	14
H	8	10	12	14
I	8	10	12	14
J	10	12	14	18
K	9	10	12	15
N	10	11	14	17
Fang	8	10	12	16
Mycroft	9	11	14	18

Table 4: Most frequently used commands in each classification

Faculty	Industry	Research	Secretary	Student
ls	set	ls	ls	ls
vi	ls	cd	cd	vi
cd	cd	vi	vi	cd
set	vi	jobs	xe	set
more	stty	set	Troff	tset
stty	dirs	stty	mail	biff
echo	hostname	next	troff	logout
pg	fg	cat	rm	cat
fg	more	fg	troffq	a.out
logout	jobs	rmm	pushd	more
mail	mail	biff	Pic	stty
ps	rm	mail	cp	rm
rm	pwd	mesg	lpq	z29init
pwd	grep	show	eqn	pix
date	if	tset	Eqn	f
tset	make	echo	lpr	mesg
jobs	logout	more	stty	fg
hostname	eval	u	page	echo
dirs	echo	rm	w	mail

7.4.1 Classification of commands

From the classification table (4) we could see that the most frequently executed commands could be classified as follows:

- directory and location related (*ls, cd*)
- terminal related (*mesg,stty,tset*)
- editor invocation (*vi, xe*)
- status commands (*dirs, f, lpq, jobs, ps, w*)
- display related (*cat, pg, more, lpr*)
- operations on files (*cp, grep, rm*)
- mail commands (*mail, next, rmm, show*)

7.4.2 Eliminating commands from window system shells

The aim of studying the usage of commands was to show that a few commands dominate the interaction process and better interfaces can be provided for them. As we noticed in the previous section a few commands constitute a large percentage of the total number of commands. Several commands can be replaced by window system equivalents that take advantage of the additional interface features. Replacements of shell services by window system is the first step to move away from the shell-per-window paradigm.

In this section we consider the commands that can be targeted for elimination from shells in window systems or replaced with alternate mechanisms. In the next section we show how interfaces can be built for the commands that are not eliminated.

The first classification consists of commands relating to directory and location manipulation. *ls* is the most popular command by far and a remarkable short cut was to build file name completion into *c-shell.* The user could type the name of the directory and follow it by the *list-completion* character. The list-completion mechanism being built into the shell opened the directory and read the contents and displayed the names of the files in the current directory. This was in contrast to running the *ls* command

which caused the shell to fork and execute the command. Such a solution is specific to the *ls* command and in fact later versions of the UNIX kernel (4.3 BSD) modified the *namei* routine used in *ls*, thus speeding up directory lookup. A specialized solution of this nature, while alleviating the *ls* problem is not the general solution we seek. Our suggestion is to display the files in the current directory at all times. A graphical display of the directory structure will permit the user to move around the directory using a mouse.

The second most frequently used set of commands are the terminal attribute setting commands. These are non functional in a window system unless we use the windows as terminal emulators. As we are trying to move away from the shell-per-window paradigm we can ignore this class of commands.

The third category is the editor invocation commands. In modern window systems where there are editing environments such as Emacs, the need to invoke editors with less capabilities like *vi* or *xe* is minimized. Besides, these editors are not capable of making use either of the advanced input devices such as mouse, or of the ability to display different parts of the file being edited in distinct regions. Editing environments are capable of providing advanced interaction mechanisms.

Status commands continue to remain useful in any environment. The user is still going to create processes, print files, and inquire about the status of the machines and users in his immediate environment. Some of the commands in this category can be however eliminated. For example, *dirs*, which displays the stack of directories pushed onto the user's environment can be displayed at all times obviating the need for the command to be ever executed. The stack of directories can be displayed in the banner area of the window. The display will include the current working directory and can be displayed in conjunction with the files in the current working directory.

Graphical interfaces to collections of files enable elimination of commands that operate on files. For example, in the Macintosh system, a file can be removed by dragging its icon to another icon representing a trash can. Searching within a file is an editing operation provided by the editing environment. However the advanced window system interface does not improve the ability of the user to perform operations on collections of files. Using the meta-character '*' a *csh* user can search for patterns in a collection of files. The advanced interface can be then used to display all the files

in which the pattern was found. We see the need for retaining some of the notions of the present shell here.

It is interesting to note that the next category of most frequently used commands is the set of commands relating to the mail subsystem. A mail specific interface takes care of the commands in this category. The mail subsystem can provide the user with some context, permit him to switch between this context and another. For example, he can begin to compose a message, move to another window, extract some text from there and insert it into the message. We advocate a separate context for related operations.

7.4.3 Interface for commands not eliminated

In the previous section we looked at the commands that can be eliminated from shells. In this section we see how a general interface can be provided for commands that have not been eliminated. We also see how specific interfaces can be provided for certain commands.

Once we have established the most frequent commands that cannot be eliminated, we can attempt to tailor the user's interface to these commands. For example, we can have a fixed menu of the most frequently used commands that is displayed at session startup time. Depending on the user's choice of style of interaction this can be made available in a pop-up-menu also. In Section 8 we will see how a suitable interface can be provided for the commands used frequently during a current session.

Commands in the status category included some that could not be eliminated. We suggest improving the interface for such commands. For example, the status of commands running could be displayed as icons as in Spice's Sapphire window system.[2] A command that periodically gets updates about the status of other users in the immediate environment eliminates the need for commands like *u, w*. In general for status commands that are going to be invoked repeatedly a separate window can be dedicated.

The question of handling inquiries about status of jobs sent to printers *(lpq)* or jobs waiting to be processed by document formatters is answered slightly differently. Commands dealing with the status of jobs printing and waiting to be processed by formatters almost always arise after the corresponding invocation of the commands that print files or submit files to be formatted. If we caused the commands that invoked the printing or formatting to automatically report on the status of the jobs we can

eliminate status commands to a reasonable extent.

7.4.4 Using raw command data

So far we have seen how to use command information data. Our data gathering involved not only logging of full path information but the raw command lines typed by the users as well. The full path information was logged so that aliases used by users would not prevent factual data to be compiled. From the raw command line data we can extract information about usage of parser metacharacters that are immune from aliasing. Over two million command lines were examined during the course of data gathering.

The parser metacharacters are used to redirect input and output, to run jobs in background, to pipe input between processes, to combine commands, and to invoke the history mechanism. We studied the percentage use of the various metacharacters in each machine and the combined percentage according to our classification. The results are presented in Table 5 and Table 6 respectively.

A small but significant percentage of the command lines were manipulations of the command history. An interface whereby the previous commands are available in a menu to be selected easily would be an improvement to textual subsitution or selection from a history list using command numbers as a handle. There is a direct correlation between the advanced programmers as well as users who repeated commands frequently and the running of jobs in background. For example, from Table 6 we see that secretaries had the highest pecentage use of the history mechanism.

A subset of the history mechanism usage is the redo ('!!') command. A high percentage of redo indicates that commands are frequently repeated. We see that is the case in both the industry and secretary classification. This fact is borne out by the data presented in Table 3 where the machines comprising the industry classifcation have fewer commands (13 on the average) that are executed 85 % of the time.

Jobs that are run in background can be automatically migrated to a separate window. The user would not have to inquire about the status of the job, as the window in which the job is running will indicate the status. As mentioned earlier a changing icon would work just as well in place of a window for status indication. Once again we notice that the

Table 5: Percentage usage of metacharacters

Host	!	!!	&	<	>	|	;	Total
arthur	6.5	1.3	0.9	0.4	1.0	1.3	0.5	78519
l	5.7	1.1	0.3	0.3	0.6	0.3	0.2	69214
newton	2.5	0.5	0.5	0.5	0.6	1.1	0.3	81754
surya	8.8	0.6	2.7	2.1	3.5	2.5	3.5	298
fang	5.6	1.9	0.6	0.6	0.6	2.5	0.3	117218
mycroft	4.7	1.0	0.4	0.1	0.8	1.8	0.8	157180
blays	5.8	2.0	0.7	0.5	0.6	0.8	0.7	8868
j	4.1	1.0	0.5	0.6	0.7	0.3	0.5	195795
lionel	4.9	2.0	0.3	0.0	0.1	1.2	5.2	3134
lucas	2.9	0.6	0.4	0.1	0.7	1.5	0.3	1920
merlin	5.1	1.2	0.7	0.2	0.2	0.6	0.7	1743
ector	6.5	0.6	3.4	0.2	1.1	3.5	0.0	22159
mordred	5.7	1.8	0.6	0.7	0.9	1.0	1.0	96599
gwen	6.2	2.4	0.2	0.0	0.4	0.6	1.5	24845
h	4.7	0.5	0.3	3.2	1.7	1.0	0.2	286137
i	4.4	0.6	0.3	3.0	1.6	0.9	0.2	280764
k	4.7	0.6	0.2	3.4	1.9	0.9	0.2	280147
n	5.2	0.9	0.6	2.9	1.4	1.0	0.2	66214
s	2.8	0.3	0.2	5.9	2.4	0.3	0.1	323529
Average	5.095	1.100	0.726	1.300	1.095	1.216	0.863	110317.737

! — History substitution
!! — Redo
& — Backgrounding
< — Input redirection
> — Output redirection
| — Pipe
; — Command concatenation
Total — Number of raw command lines

Table 6: Percentage usage of metacharacters in each classification

Class	!	!!	&	<	>	|	;
faculty	4.5	0.9	0.5	0.4	0.7	0.7	0.3
industry	5.1	1.5	0.5	0.3	0.7	2.1	0.5
research	4.2	1.1	0.5	0.5	0.7	0.5	0.8
secretary	6.5	0.6	3.4	0.2	1.1	3.5	0.0
student	4.1	0.5	0.2	4.0	1.9	0.7	0.2

secretary classification which has the highest percentage use of background ('&') mechanism also has two status commands in the top 10—*troffq* and *lpq*.

The input and output redirection percentages indicates the fact that students who tend to use data files and output files use the redirection mechanism more often. It is clear from the data in Table 4 that the most frequently used commands like *ls*, *vi*, *cd* etc. are unlikely to have input/output redirection. This fact implies the low percentage of command lines iwth input/output redirection. In a workstation environment we are likely to deal with regions of characters and other objects beyond files. The data shown in the above table is significant in the sense that at least a large group of users (students) would benefit from an improved interface for advanced input specification mechanisms. In our classification the number of students far outnumbered the rest of the classifications put together (Table 2 shows that approximately 85

The ability of sending output from a process as input to another process via the pipe mechanism is also studied. The table shows that secretaries who primarily issue word processing commands pipe them through filters. A quick cross check confirmed this. The most frequently piped-into commands for each classification is presented in Table 7.

In looking at command sequencing, the highest percentage was found in the research category and the lowest in the secretary category. The classification of semi-colon being the command sequencing category is not quite accurate as commands can be sequenced by the background character as well.

Table 7: Percentages of most frequently piped-into commands

Faculty		Industry		Research		Secretary		Student	
pg	18.6	grep	33.8	more	17.4	troff	19.4	cdc	26.8
grep	12.9	more	26.0	grep	8.8	Troff	19.2	more	19.5
more	11.7	m	5.5	less	7.2	eqn	18.1	lpr	12.2
lpr	11.4	bm	5.1	fgrep	7.2	ms	15.4	grep	6.6
m	3.2	wc	3.1	head	6.9	tf	13.8	a.out	4.8
troff	3.1	lpr	2.7	egrep	5.8	ntroff	3.0	wc	2.1
netlpr	2.8	tail	1.8	tail	5.4	tbl	2.5	cpp	1.5
sort	1.6	iroff	1.8	bm	3.6	Eqn	2.1	cat	1.4
pic	1.4	egrep	1.6	wc	3.2	mail	1.6	fgrep	0.9
splot	1.4	tperf	1.5	lpr	2.5	pic	1.2	pg	0.9

8 History mechanism

Our analysis of the data gathered from the modified shell showed that the number of distinct commands typed by the user are few. A history mechanism that kept track of these distinct commands can greatly reduce the need to retype these commands. Traditionally, history in *C-shell* had a textual interface owing to the limitations of the display hardware. The user could display his last n commands for any value of n between 1 and the number of commands he has issued thus far. In a command interpreter designed after *C-shell*, namely *Korn-shell*, the author David Korn provided the user with a one-line editable history window. The user could move up and down his history list and re-execute a command after editing it if he so desired.

The history mechanism is used for purposes of keeping track of a user's session, to re-issue a command, or to undo a command. Our belief is that the users tend to:

- Execute one of a small number of commands repeatedly.
- Re-execute a command after making possible modifications.
- Examine the output of the previous execution of a command.

- Filter the output from previous commands repeatedly.

We had to provide the facilities that are already provided in *C-shell* and *Korn-shell* and go beyond that. The interface to the history had to be in keeping with the visual nature of an overlapping window system. As part of the UNCLE [6] prototype window system built to explore abstraction mechanisms in the interface between user and application programs, we considered the shell as one of the application programs. Our novel history mechanism was part of UNCLE's interface to the shell. In UNCLE, the user can, with the transition of a mouse button, pop up a menu consisting of a stack of panes, one for each distinct prefix of a command line, with distinct commands sharing the same prefix appearing as separate items in the same pane. Thus, if a partial list of user's commands were:

finger

ls /tmp

finger bala

cat /etc/motd

finger @blays

cat foo

ls /usr/src

there would be three panes: labeled 'finger', 'ls', and 'cat'. Both the finger entries appear as items on the 'finger' pane. Similar commands are thus automatically grouped.(Figure 1)

Our history menu is *not* a log as only distinct command lines are stored. It can however be used to re-execute a command, or simply insert the command in the application program window to be edited before execution, or to display the output of the previous execution of the command. To conserve memory, we do not save the output of every command executed. Instead, we let the user explicitly specify the commands whose output he would like to be saved. Our belief is that the user will specify commands that he either expects to take a long time to complete or those he would

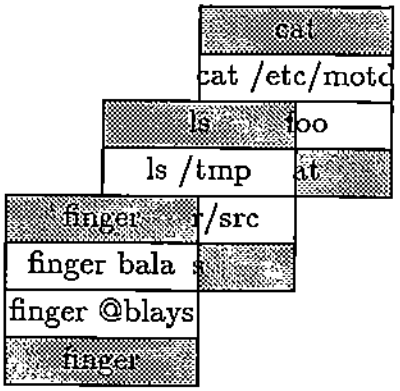

Figure 1: UNCLE history menu

use later, to be displayed in its own window. With our history mechanism he can come back at a later time and view the output of the command at leisure. Further, he could run the output through filters.

As we do not view interaction with the window system separately from interaction with the application programs, a history of the interaction between the user and the window system is automatically maintained. The mechanism to maintain the history of user–window system transactions is the same. By storing the changes made to the interactive environment separately, we can move back to a former style of interaction. This can function as a quick undo mechanism.

9 Conclusion

In this paper we considered the role of the command interpreter in an interactive system. It was our view that the shell-per-window paradigm whereby each physically distinct region had a shell running in it was a bottleneck in exploring superior interfaces to the commands of application programs accessible via the interactive system. To improve on the existing model we modified a popular command interpreter to obtain usage data. Analyzing the results brought out some interesting aspects of interaction. For example, regardless of the user classification, a small number of commands were used extensively. Some of them could be eliminated in window system shells, and others could either be replaced or provided with an improved

interface. Our approach of classifying the predominantly used commands and exploring mechanisms to improve the interface in each classification was vindicated to a reasonable extent.

References

[1] G. R. Andrews, R. D. Schlichting, R. Hayes, and T. D. M. Purdin. The design of the Saguaro distributed operating system. *IEEE Transactions On Software Engineering*, SE-13(1):104–118, January 1987.

[2] M. R. Barbacci. *The Spice Users' Manual.* Carnegie Mellon University, August 1984.

[3] S. Bourne. Unix time-sharing system: The UNIX Shell. *The Bell System Technical Journal*, 57(6):1971–1990, July-August 1978.

[4] S. J. Hanson, R. R. Kraut, and J. M. Farber. Interface design and multivariate analysis of UNIX command use. *ACM Transaction of Office Information Systems*, 2(1):42–57, March 1984.

[5] D. Korn. Ksh—A Shell Programming Language. In *Proceedings of the Summer USENIX Technical Conference*, pages 191–202, 1983.

[6] B. Krishnamurthy. *Partitioning the Process of Interaction: An Abstract View.* Technical Report CSD TR 705, Department of Computer Sciences, Purdue University, September 1987.

[7] J. Levitt. The Korn shell: an emerging standard. *UNIX/WORLD*, III(9):74–81, September 1986.

[8] M. T. Rose and J. L. Romine. *MH—Mail Handler.* The Rand Corporation, April 1986.

[9] A. B. Sheltzer and G. J. Popek. Internet Locus: Extending Transparency to an Internet Environment. *IEEE Transactions On Software Engineering*, SE-12(11):1067–1075, November 1986.

[10] D. C. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslem. Designing the star user interface. *Byte*, 7(4):242–282, April 1982.

[11] Sun Microsystems Incorporated. Sunview programmer's guide. February 1986.