A Recovery Mechanism for Shells

IAN HOLYER AND HÜSEYIN PEHLIVAN

Department of Computer Science, University of Bristol, BS8 1UB, UK
Email: ian.holyer@bris.ac.uk

An undo facility is an essential component of most interactive applications. In current operating system shells, whether textual or graphical, such facilities are typically very poor. Algorithms are presented for adding a recovery mechanism to a shell which allows previous commands to be selectively undone and redone, and previous versions of files to be recovered.
The recovery mechanism involves making the shell control resources in a more intelligent way. Programs are run under greater control, with the shell monitoring and analysing their resource requests. This provides better high-level information to the shell and, for example, provides techniques to prevent foreign or untrustworthy programs from doing any damage, and to reduce problems with conflicting resource requests from concurrent programs.
A prototype implementation called `brush` has been constructed to investigate the convenience and natural feel of these facilities.

Received November 13, 1998; revised December 7, 1999

1. INTRODUCTION

A recovery ability is a crucial feature that many interactive single-user applications provide to allow the user to reverse the effects of previous commands. This capability of applications enables the user to recover from unintentional commands and repair any resulting damage at any point in the interaction. An undo facility, for instance in an editor, encourages a user to act more freely, without the fear of losing useful information.

A *shell* is a program which provides a user interface to an operating system. It may be a text-based command line interpreter as in Unix, or it may be a graphics-based file and process manager. Either way, the ability to repair damage to permanent resources such as files is an important one. The facilities which are typically provided at present are rather primitive, consisting of a ‘waste bin’ directory where old versions of files are stored when explicitly deleted, together with various ad hoc backup mechanisms provided by individual applications. It is ironic that one can always undo the deletion of a single character in an editor, but not the deletion of a permanent file in a shell.

The aim of this paper is to describe a way of designing a more intelligent shell which keeps track of versions of files on behalf of the user, together with information about how they were created or manipulated. This enables it to provide a more uniform and consistent mechanism for undoing the effects of commands, recovering old versions of files, repairing accidental damage, and otherwise managing a user’s most permanent and valuable resources in a safe and convenient way.

The ideas presented in this paper emerged from the desire to design an operating system shell using a purely functional language such as Haskell. This involves redesigning the shell to remove as much non-determinism as possible. However, the ideas potentially have a wider application, so they are presented here without reference to functional programming. The aim is to achieve a shell with the following properties:

- Intelligent management of files and programs.
- The ability to undo commands and recover old versions of files.
- The ability to kill rogue programs cleanly.
- The ability to run foreign programs with minimal risk.
- The ability to run programs concurrently with clean sharing of resources.

Many of these points depend on the ability to undo and redo commands, and this is the aspect which we concentrate on in this paper.

The model for recovery which we present is concerned with users’ recovery from their own prior commands. We are not concerned with data loss through hardware failure, for example, nor with the recovery of old versions of files from overnight file system dumps, nor with the backups which some application programs keep to guard against system crashes.

Conventionally, a shell runs programs which are allowed to access resources directly using system library procedures. In order for a shell to be able to act in a more intelligent way, and provide an undo mechanism, it needs to be able to keep track of the changes which each command or program makes to permanent resources such as files.

The remainder of this paper is organized as follows. In Section 2, previous work on extending operating system functionality is discussed briefly, and the literature on undo support is reviewed. Section 3 deals with analysing the effects of commands, determining to what extent they are reversible, and handling untrustworthy programs. In Section 4 a specification for a proposed undo and redo mechanism is presented, as it might look from the user's point of view. Section 5 discusses the algorithms needed to implement the proposed recovery mechanism. Section 6 deals with the issues which arise when concurrent programs are supported by the shell. The results are summarized and future work is outlined in Section 7.

2. PREVIOUS WORK

In this section, we review techniques for extending operating systems, and we also review previous work on undo facilities.

To be able to undo the effects of commands, we need to monitor and control the permanent changes that commands make. For example, if a command overwrites a file, we must detect this and take a copy of the old version first, so that we can restore it later if necessary. In addition, in order to execute foreign programs safely, it is desirable to apply suitable security policies to the changes which a program attempts to make, beyond what can be done with conventional user permissions.

A shell causes programs to run, and those programs access operating system primitives directly, rather than via the shell. What we need to do is to extend and adapt the operating system primitives so that they cooperate with the shell in managing resources, allowing the shell to monitor and control what is going on. Thus, in effect, we need to extend the operating system functionality, changing the way in which its primitives work.

Various methods for extending operating system functionality have been proposed, and are discussed and compared for example by Alexandrov *et al.* [1], who use the extended functionality for adapting filing systems. Jones [2] uses extended functionality for implementing interposition agents, and Goldberg *et al.* [3] use it for the secure execution of untrusted programs. All these example applications are relevant to our work.

One approach to extending the operating system is to change the kernel itself. The main problems with this are that it requires superuser privileges to make the changes, and that the risk of compromising the integrity or security of the system as a whole is very high.

A second approach is to change device drivers, libraries or managers associated with the operating system. For example, it is often easy to add new communication-based file system managers, as with NFS, and one of these could represent an alternative method of access to an existing file system. The main problems are that programs may need recompiling and/or relinking, and that the approach is unlikely to be sufficiently general.

The most successful approach appears to be to intercept system calls. In most operating systems, commands and programs use system calls as the lowest level interface to operating system facilities. The operating system may provide a way of intercepting the system calls made by programs. For example, many Unix-based operating systems provide a device called `/proc`, described by Faulkner and Gomes [4], which allows one process to gain control over another at the system call level for debugging, tracing or other purposes. This facility can be used without needing special privileges, and is used in Unix commands such as `ps`, `truss` and `strace`.

This approach is very suitable for implementing undo, because system calls are the only way in which commands or programs can make permanent changes to the system. Moreover, no high-level knowledge about the intentions of any particular program is needed. This approach is used in the `brush` prototype, described in Section 7.

Issues concerning undo support have drawn the attention of many authors for some time. Some have examined the relation between undo support and the interface of a computer system and focused on the reasons why undo support facilities are important, while others have described a number of different undo models and implemented them as part of various systems.

Undo support is a capability which is directly concerned with maintaining the integrity of a user's work [5] and should allow easy reversal of actions as long ago in the history as possible [6]. The psychological behaviour of a user may make interactions with the system frustrating. For example a command may give an unexpected response and, therefore, a system should provide recovery from unwanted actions conveniently and easily [7]. However, frustration can be caused by irreversible actions such as the accidental deletion of important data. As it is difficult to cope with irreversible actions, interface actions should be made as reversible as possible [8]. Also, the provision of undo support may help users in minimizing the time spent in correcting errors. To avoid such a waste of time, error-recovery methods should be designed in terms of learnability and efficiency [9].

Many programming environments, and application programs such as editors and word processors, support a *single undo* facility. The most recent command can be undone and redone, but commands previous to that cannot be reached. Examples are the Smalltalk system [10] and the Vi editor [11].

More sophisticated environments and applications support a *multiple undo* facility in which the most recent commands can be undone consecutively, back to some previous commit point such as the beginning of the current session. A general model for this has been presented by Archer *et al.* [12] and examples include Microsoft products, COPE [13], Interlisp [14], PECAN [15] and Sam [16].

Most systems with multiple undo also support *redo* facilities. The most important property of a redo facility is recoverability, as described by Gordon *et al.* [17]. It should be possible to revert to any previous state that the system has been in, including all states which were abandoned by undoing some commands and then issuing alternative commands. A number of approaches to the provision of completely recoverable undo facilities have been investigated by providing various undo, redo, skip and rotate commands and defining how these act on each other, for example by Vitter [18] and Yang [19]. The Emacs editor [20] achieves a completely recoverable undo facility using only a single special undo command. Consecutive undo commands provide a multiple undo facility, recovering from any number of previous commands. However, any command other than an undo command breaks the sequence of undo commands and, at this point, any previous undo commands are treated as ordinary changes that can themselves be undone.

Beyond this, it is possible to provide *selective undo and redo* facilities in which an arbitrary previous command can be chosen and its effect undone or redone. General models are described by Berlage [21, 22] and by Prakash and Knister [23], who include issues to do with multi-user interaction, and an example system is GINA [24]. Desirable features of selective systems are that the current state should correspond to the sequence of non-undone commands which have preceded it, and that a command which is redone should be executed in a state which agrees with the state in which it was originally executed. Unfortunately, conflicts can arise in which some commands cannot be undone or redone without violating these desirable features. We will discuss these conflicts later.

3. EFFECTS OF COMMANDS

To be able to reverse the effects of a command, it is necessary to be able to detect and analyse those effects. Often, the permanent effects of a command consist of the files which it creates or alters. Using the mechanisms in the previous section, these can be monitored, copies can be taken of old versions of files before they are altered, and the effects of a command on the file system can be reversed by reverting to those old versions. However, commands can have a wide variety of different effects, other than their effects on files, some of which are difficult or impossible to reverse.

First, there can be other local state changes such as changing permissions or other attributes on files, or changing the directory structure. These state changes can generally be dealt with in the same way as with files. In the case of directories, we will see in Section 5 that it is not necessary to keep old versions of directories.

Second, there are changes to the state of the shell itself, e.g. changing the current directory, altering the values of environment variables, keeping track of command aliases and so on. Similarly, some shell commands are intended to be used in scripts as programming tools, e.g. for controlling loops. These are all necessarily commands which are built into the shell. As they are under the direct control of the shell, they cause few problems; indeed, the shell state can be stored in files and the mechanisms for keeping track of versions of files can be used for these too.

Third, programs may have external effects, such as interacting with the user, printing a document, sending an email, communicating interactively with another user or changing remotely stored files. Tracking these effects can be difficult [25]. Although the relevant system calls can be trapped, it may be difficult to determine what they do or how to undo them. In general, these external effects cannot be undone, and we regard these issues as being beyond the scope of our shell.

Finally, some shell commands deal with concurrent process control and interactions between multiple users. This includes compound commands which run several programs together, possibly communicating with each other via pipes. It also includes direct process control commands which create long-running processes which execute concurrently with the shell and each other. Undoing the effects of a complete process group may be relatively easy, but undoing the effects of individual processes within the group is more difficult. Concurrency issues are discussed further in Section 6.

In addition to tracking the effects of commands in order to undo them, it is also desirable to control the effects of potentially untrustworthy foreign programs. It is conventional for World Wide Web applets to be run in extremely restricted environments which effectively prevent them from making any local changes or accessing any local information. In addition, Goldberg *et al.* [3] discuss ways to restrict helper applications which get run as a result of downloading files from the Web. However, it is also common to obtain complete application programs directly from the Web or other untrustworthy sources and, at present, such programs are usually run with no restrictions at all.

To some extent, providing an undo facility in itself provides some protection from such programs; if a program deletes files it shouldn't, they can be recovered easily. However, stricter knowledge and control of what a program is doing is desirable. It is possible to use a restricted shell to prevent a user from doing damage by forbidding access to unsafe programs. However, our aim is rather different. We want to allow a user to run any desired program, but to prevent the program from doing damage by running it in a restricted way. We want the program to run in its normal environment, allowing it to do useful things such as creating files. However, since all file accesses are monitored, a security policy can be applied, and any unauthorized access is forbidden or referred to the user for confirmation.

One further issue which needs to be addressed is the question of aliases. If there are two names for the same file and a command writes to the file using one name, it also changes the contents visible via the other name. The shell needs to know about this to keep track of the effects of commands. In a concurrent setting, if two commands attempt to access the same file via different names, the resulting contention problem needs to be recognized and addressed by the shell.

If all commands which create or manipulate aliases are issued by the proposed shell, it has complete control and can keep track of them. Otherwise, alias problems have to be detected as and when they occur. Detection methods differ from one operating system to another, and detection of all cases of aliasing can be difficult. However, reasonably good solutions are usually possible. For example, given a pathname on Unix, a unique device identity number and file identity number (inode number) can be obtained and used to record where the file is physically held.

From now on, we will assume that mechanisms like this are in place so that the shell knows exactly what each command is doing to the system.

4. A PROPOSAL FOR UNDOING

The next few sections form a proposal for a simple version of a shell with undo and redo facilities. In this section we describe how the facilities look to the user. The description concentrates on changes to individual files, but applies to other permanent resources such as directories. Concurrency issues are deferred to Section 6.

To allow the user to undo previous commands, the shell provides a history list, i.e. a list of the commands which have been issued recently, either up to some fixed number of commands, or back to some previous commit point such as a logout. Many text-based shells already provide such a history list, where each entry records the text that was typed. In the case of graphical interfaces, a textual description or other visual representation of the recently issued commands needs to be provided.

With text-based shells which provide a history list, it is common to provide a *resubmit* facility. This allows the text of a previously issued command to be copied, edited if desired, and then submitted as a new command. This is a cut-and-paste facility which saves time by reducing the amount of typing the user needs to do. The undo and redo facilities described here can be added without affecting the resubmit feature.

A selective undo facility is provided where the user can select one of the previously submitted commands and ask for its effects to be reversed. Suppose that four commands are issued, followed by a request to undo the second command, which involved updating a file `fileA`:

```
write fileA
(edit fileA)
move fileB fileC
copy fileC fileD
```

We have used brackets here to indicate that the appearance of the `edit` command in the history list has changed, perhaps by being greyed out or having a different colour. This change of appearance acts as a record of the undo command, which does not appear explicitly in the history list. The undo command causes the new version of the file `fileA` to be removed from view and saved, and the old version of the file `fileA` (which was saved when the `edit` command was issued) to be reinstated.

From now on, we refer to the commands in the history list as *active* or *inactive* according to their current status and hence appearance. A redo command is provided to reverse the effects of undoing. An inactive command such as the `edit` command above is selected, and the original effects of the command are reinstated. The appearance of the command in the history list is also reinstated. Undo can only be applied to an active command, and redo can only be applied to an inactive command, so a single command name or mouse button or keystroke can be used for both.

The redo feature is very different from the resubmit facility. What happens is that the changes to file versions carried out by the undo command are reversed. The new version of `fileA`, which was saved when the undo was issued, is reinstated. This contrasts with a resubmit where the editor is re-executed.

The intention of these facilities is that when a command is undone, the state of the filing system is exactly the same as if the command had never been issued. When it is redone, the state is exactly the same as if the undo had never been requested. This is an important principle which makes it easy to understand the meaning of undo and redo. It can be stated as an invariance condition:

The current state should be completely determined by the initial state and the active commands in the history list.

This provides a very simple mental model for the user of what undo and redo mean. However, it follows that undo and redo are not always possible because of dependencies between commands, as described by Prakash and Knister [23]. For example, the `move` command above cannot be undone because `fileC` would not be available for the following `copy` command. This brings into question a second important principle:

It should be possible to return to any previous state.

In fact this is possible with the facilities described so far, though it is not necessarily very convenient. It is always possible to undo the last active command in the history list, and so to undo all the active commands sequentially, from the last one backwards. After that, the active commands in the desired previous state can be redone in a forward direction. Of course, it may be possible to achieve the desired result more efficiently in practice.

A feature of the system which would make undoing more convenient would be that when an undo is requested for which there are later dependent commands, the system could offer to undo those later commands at the same time as the requested command. Similarly, a request to redo may result in an offer to undo or redo other commands as well, to make the requested one possible. The issue of what constitutes a dependency will be explored further later.

A further principle which needs to be addressed is whether desired versions of files can always be recovered. Specifically:

It should be possible to recover any desired collection of versions of files, or other resources.

This is difficult if we want to recover two different versions of the same file, or versions of two files which do not correspond with each other. A simple example of this is the following:

```
write fileA
edit fileA
```

The first command creates one version of `fileA` and the second replaces it with a new version. Suppose it is now discovered that important information was deleted by accident during the editing session. Important new information was also added during the editing session, so both versions are needed in order to resolve the situation.

A further feature of the design which takes care of this sort of problem is the ability to insert a command into the history at an arbitrary point. Inserted commands, as with undo and redo, have to be checked for dependencies before being allowed. In the simple example above, the recovery procedure would be first to undo the `edit` command, then insert a copy command before it, and then redo the `edit` command, to give:

```
write fileA
copy fileA fileB
edit fileA
```

Now both versions of the file are available as `fileA` and `fileB`.

It is important to make sure that the proposed shell behaves correctly when commands fail. For example, suppose the user types `move fileA fileB` at a time when `fileA` does not exist. The state of the system should be made the same as if the command had never been issued. The command is added to the history list as an inactive command, to allow editing of its text and re-submission.

5. IMPLEMENTATION

In this section, we describe the algorithms which are necessary to implement the desired shell features. We assume that system calls are monitored in such a way as to record all the operating system transactions which a command or program carries out. For simplicity, we will initially concentrate on the effects of commands on a collection of files in a single directory.

In order to deal with different versions of the same file, the shell distinguishes them according to their time of creation. The shell numbers all the transactions which occur, and uses these sequence numbers to represent time. The time of creation of a file version is the sequence number of the transaction which produces it. The shell simply needs to keep track of the 'current time', i.e. the sequence number to be issued when a transaction is next executed.

The shell keeps all old versions of files in a subdirectory called `save`, say, using their times of creation as file names. For example, suppose that the command sequence:

```
write fileA
write fileB
copy fileB fileA
edit fileB
```

results in the creation of the first version of `fileA` at time 1, the first version of `fileB` at time 2, the second version of `fileA` at time 5 and the second version of `fileB` at time 7. The files existing after these commands are:

```
fileA
fileB
save/1
save/2
```

If `fileA` and `fileB` are changed again at a later date, the current versions are saved by moving them as:

```
fileA  ->  save/5
fileB  ->  save/7
```

Thus the directory is seen in a correct state by programs outside the influence of the shell, except for the extra `save` directory.

The shell ensures that when a command is run, it does not result in any file versions being deleted. File system requests from the command are converted into actions which create new file versions or move them as appropriate.

The shell stores the history list, with the time at which each command was issued, and whether or not the command has been undone. For example:

```
1> write fileA
2> write fileB
3> copy fileB fileA
6> edit fileB
```

The time at which a command is issued does not need to be visible to the user, provided that there is some way for the user to indicate which command to undo or redo as necessary.

In addition, the shell stores the sequence of transactions which occurs as commands are executed. For simple files, we need only consider the creation of a file version, reading from a file version, and deleting a file version. Other file system requests can be handled as combinations; for example, a request to open a file for appending can be treated as a `Read` followed by a `Create`. For each transaction, the file name and version affected are stored. For example, for the above command sequence, the transactions are:

```
1> Create fileA 1
2> Create fileB 2
3> Read fileB 2
4> Delete fileA 1
5> Create fileA 5
6> Read fileB 2
7> Create fileB 7
```

The history list and the transaction sequence allow the shell to work out which versions of files are current, and which versions were current at any particular time in the past. To do this, the shell keeps track of deletions of file versions, which are treated as separate transactions. If a command overwrites a file, this is treated as two transactions; one to delete the old file version, and one to create the new version. If we take time *t* to mean the time just before the transaction tagged with sequence number *t* occurs, the file versions current at time *t* are those with active creation times less than *t* and active deletion times (if any) greater than or equal to *t*.

In addition, the shell keeps track of which transactions are active or not, i.e. which transactions belong to active commands. (In fact, the information about which transactions are active, and the version numbers of the files, are redundant and could be re-created from the remaining information and the history list.)

This information about transactions does not normally need to be visible to the user. For built-in commands, the information can be generated from a knowledge of what the commands do. For other commands, the information can be gathered while the command runs by monitoring its system calls, as discussed in Section 2.

It is possible to allow the user to see the transaction sequence for a command, and to selectively undo or redo individual transactions. If not, the fact that all the transactions belonging to a particular command are undone or redone together allows the transaction sequence to be thinned out, if desired. For example, the transactions belonging to one command need only contain at most one `Read` and one `Create` or `Delete` for each file. For a file version that existed before the command was issued, a `Read` is stored if the contents of that file version may have been read by the command, and a `Delete` if the version does not survive after the command. For a new file version that the command produces, a `Create` is stored. Nothing needs to be stored for temporary files, i.e. ones which are created and deleted during the course of the command.

How does the shell implement undo? First, it has to detect whether a request to undo a command is valid. If a file was created or altered by the command, and the new version of the file is mentioned in the transactions of any subsequent active command in the history list, then the undo is invalid because it would lead to a version inconsistency. If a file was deleted by the command, and there is a subsequent command which creates a new version, then again the undo is invalid (because the behaviour of the command may depend on the non-existence of the file, as with the use of the `O_EXCL` flag in Unix). There are no other restrictions; an undo request is otherwise always valid. Next, the shell can use the command's transactions to determine how to change the current state of the filestore, effectively undoing the transactions one by one in the reverse order.

How does the shell implement redo? Again, as with undo, files created, altered or deleted by the original command must not be mentioned in any subsequent active transactions. However, there are additional restrictions to ensure that the redone command accesses the same versions of files as when it was originally issued. Suppose that the original command was issued at time *t*. If the command created a file without first deleting an older version, and an older version of the file is now current at time *t*, then the redo is invalid. Also, for any file which was read or deleted by the command, the version which was originally affected must match the version which currently exists at time *t*. For example, suppose we have the following situation:

```
1> write fileA
2> (edit fileA)
4> (copy fileA fileB)
```

with stored transactions:

```
1> Create fileA 1
2> (Delete fileA 1)
3> (Create fileA 3)
4> (Read fileA 3)
5> (Create fileB 5)
```

A request to redo the `copy` command is not valid, because the version of `fileA` (version 3) which the `copy` command was supposed to act on does not currently exist at time 4. This restriction ensures that a redo can always be implemented by manipulating file versions rather than by re-executing the command itself. In this example, the `edit` command needs to be redone before the `copy` command can be redone, perhaps inserting a command to save the old version of `fileA`, if desired.

The insertion of a command into the history list other than at the end is handled in a similar way to a redo request, except that the versions of the files affected can be taken to be the ones existing at the time of insertion. The shell may not discover that the inserted command is invalid until it is part-way through execution, but the failure can be reported and the partial effects of the command can be undone.

In the above, we have prevented undo or redo whenever a file is involved which is mentioned later. It is possible to be less restrictive. If two successive changes to a file involve disjoint portions of the file, the changes may be regarded as independent. This is the approach taken in systems like CVS [26] where multiple developers may work on the same program source file, the changes in the text are stored using RCS and treated as independent whenever they don't overlap. In our setting, this would allow two non-overlapping changes to a file to be undone or redone independently of each other, and would in general reduce the amount of space used for saved versions of files.

So far, we have only dealt with simple files. Other local state changes are dealt with in similar ways. Consider attributes of files, such as permissions and alias information, for example. A change of attributes can be handled correctly with the mechanisms already described by creating a new version of the file with the new attributes attached. However, the storage of multiple versions of the file contents can be avoided by adding an extra transaction type which records a change of attributes, and which can be undone by reversing the change. Also, some undo and redo operations can be allowed, which would otherwise have been invalid because of the change of file version, provided care is taken to ensure that the attribute changes do not affect the relevant commands. Directories are handled by adding transaction types which record changes to the directory structure. At first sight, it appears that multiple versions of directories need to be stored, as their contents change with time. However, this is not necessary, as the state of a directory at some previous time *t* can be inferred by working out the file versions which were current in it at time *t*, as described earlier.

For the most part, external effects such as printing or communicating with remote facilities are simply declared as being outside the scope of the shell. Although these effects are detectable in principle by monitoring the system calls which access the relevant devices, in practice the effects are not reversible, or the intentions implicit in the device accesses are impossible to interpret. The user must be made aware that an undo command only undoes the local effects of a command, and other external effects are left to the user to deal with.

In addition, some local files may not be suitable for handling by the shell. For example, some files might carry with them special security or system integrity implications. To cover such cases, the shell provides a way of making specific files or directories exempt from the usual version handling, so that changes to them are treated in the same way as external effects which the user must handle. Database systems also often pose a problem. A database may be held on a separate disk or disk partition outside the control of the normal file system, or it may be held in a large file for which multiple versions might take up too much space. On the other hand, database systems often provide their own recovery facilities, so it makes sense to exempt them from the usual version handling too.

6. CONCURRENCY

In this section, we describe a number of complications introduced by concurrency and non-determinism, and propose some solutions. Concurrency arises from shell commands which create and control processes. Most of the problems involve concurrent access to shared data, which is discussed at length in the database management literature. The solution in databases is based on the idea of transactions and the ACID properties of transactions, as described by Date [27]. In our case, we can use the mechanism for monitoring and adapting system calls described in Section 2. The system call interface to operating system facilities which it uses is effectively transaction based, and many of the same principles can be applied.

The first issue we discuss is how to deal with single commands which involve several processes. For example, in Unix, a compound command `p1 | p2` creates two processes `p1` and `p2` which communicate via a pipe. It is easy to support undo and redo on the group as a whole, treating it as a single command, but it is difficult to undo or redo individual processes within the group. In the case of `p1 | p2`, the two processes can be dealt with separately if the information sent along the pipe is stored in a file, `tmp` say. Then the compound command can be treated in the same way as two commands `p1 > tmp` and `p2 < tmp` and the parts can be undone individually.

A second type of problem arises when commands share resources unnecessarily with the shell. For example, if a program uses the shell window for standard input and output, this leads to problems of arbitrary and confusing interleaving of text. To avoid these problems in the proposed shell, it makes sense to treat all commands, other than built-in shell commands, as processes which run concurrently. Each program should be run with a separate window being provided for it, where needed, for standard input and output. No special convention (such as the & symbol in Unix) is needed. As each program is started up, the shell immediately prompts the user for further commands.

There is then a problem if a program does not work properly and gets stuck in an infinite loop, for example. Conventionally, there is no clean and deterministic way to shut it down. Signals sent to the program (e.g. triggered by Control/C) may be ignored by the program. If a signal is used which cannot be ignored, the filestore may be left in an inconsistent state. However, there is a clean way to kill a rogue program using the undo facility. If the user undoes a previous program which is still running, the shell destroys the process or processes associated with the program, removes any new file versions created by the program, whether complete or partial, and restores everything to a state which is as if the program has never been run. If redo is subsequently used on the program, it must actually be re-executed from scratch, in contrast to our previous description of redo.

Long-running programs which run concurrently with the shell are likely to request resources dynamically. For example, a user may want to insert a file into a document while using a word processor. Problems arise if two programs request the same resource; which one succeeds, and what happens to the other? In conventional systems, this sharing of resources leads to unpredictable behaviour. For example, consider what happens if one program writes to a file `f` and a second program reads from it. The second program may ‘see’ the old or new version of the file (or on some systems, something in between) in a timing-dependent way. Where file locking is provided, this can add to the unpredictability because the second access may succeed or fail, depending on the relative speeds of the two programs.

To solve this, we want to ensure that all dynamic accesses are correctly sequenced by the shell. In the example above, the two relevant shell commands will appear in the shell’s history in one order or the other, as determined by the user’s sequence of interactions. If the read request appears first, the reading program sees the old version of the file. If the read request appears second, the reading program sees the new version.

One way to achieve this is to treat a dynamic request as a double action. A command is given to the shell to provide the requested resource, and a command is given to the running program to access it. In a graphical setting, such a double action is quite natural. For example, the user may select a file icon using a file manager window belonging to the shell, and use a drag-and-drop operation to give the file to the program. If all dynamic file accesses are mediated by the user and involve an interaction between the user and the shell, then the shell knows what sequence these requests occur in, and any conflicts between programs accessing the same resources can be resolved. It may seem restrictive that all resource requests should involve the user, but we have already seen that when running foreign programs, it is desirable to ensure that every dynamic access is interactively sanctioned. It is not unreasonable to apply this to all programs, not just foreign ones, though some mechanism for allowing a program to have silent access to files which ‘belong’ to it may be desirable.

Although a drag-and-drop interface is probably the most natural way of presenting this to the user, a textual version is also described here. For a request to read from a file `f` say, the shell can provide a command `give f n` where `n` is the time at which a previous program was started up, or some other identifier for the program. This makes file `f` visible to the program. An attempt by the program to read the file then succeeds. For a request to write a file `f`, a similar `take f n` can be provided. The original command to start up the program, and all the subsequent associated `give` and `take` commands have to be treated as a single group which are all undone or redone together.

We have been assuming up to now that an operating system transaction such as writing a file is indivisible. In practice, a program makes several system calls to write a file; one to open the file, several to write data into it, and one to close it. This raises the question of what happens if another program requests access to the same file while it is open. The answer is that the second program must be suspended and made to wait until the requested file becomes available, so `brush` effectively implements a locking mechanism. This raises the question of how we know whether it is the same file that is being accessed. As mentioned in Section 3, we need to find unique identifiers for files to get around problems with aliases.

For some long-running programs, particularly search programs, it is appropriate to stop them before they terminate when the results produced so far are judged sufficient. In this case, the user would want to save the partial results. This can be accomplished by a commit operation, also implemented as a double cooperative action between program and shell, which effectively splits the program run into two separate program runs.

A further question that arises is what happens if there are multiple users, or if a single user runs several shells. In order to avoid losing the deterministic properties of transaction sequencing, we propose the use of directory locking. When `brush` accesses a directory for updating, the directory is locked, which prevents two users from updating the same file at the same time. Also, when a shell attaches itself to a directory to start reading files from it, we want it to see a consistent view of the directory for the duration of the attachment. This can be achieved reasonably easily, because the old versions of files which are kept allow a shell to ‘see’ a directory at any particular time in the past, instead of the current state. A shell can thus keep a consistent view of a directory, corresponding to the time at which it attached. In the case of a directory which is being updated, a shell which attaches for reading sees the directory as it was before the updates started. In this way, a series of updates within a directory is treated as a single transaction from the point of view of other shells, so that the set of files as a whole is always seen in a consistent state. A commit command, equivalent to detaching and reattaching, can be added to allow finer-grained control over events.

7. CONCLUSION AND FUTURE WORK

The ability to undo and redo commands is an indispensable facility of interactive systems which increases confidence and productivity. We have proposed such a facility for shells, whether textual or graphical, providing a mechanism for recovery from accidental loss of files through unintentional user commands. The facility is convenient, and has a clear meaning for the user in terms of a visible command history. The ability to select arbitrary commands to undo or redo, and to insert commands at arbitrary points in the history, makes it possible to recover any previous state or any collection of desired files.

A prototype called `brush` has been constructed to demonstrate the algorithms described. It is a textual shell implemented under Unix, and it uses the Unix `/proc` mechanism to intercept the system calls made by commands and programs. Alexandrov *et al.* [1] use the same mechanism to implement direct access to remote file stores by intercepting file system calls and altering their arguments. In our case it is not necessary to alter the arguments to a system call before it goes ahead, merely to insert some extra processing. Also, where file handling is concerned, we need only intercept `open` and `close` calls, and not `read` or `write` calls. In practice, with typical use of the shell, the time overhead involved in managing file versions is acceptable.

The prototype currently keeps all versions of all files, without compression. This has an obvious space overhead, and the efficient storage of multiple versions of files needs to be addressed. Also, methods are needed for determining exactly which versions need to be kept and for how long, for example by detecting which files are source files and which files are generated files so that only source files need be kept. In the long run, a shell which has much more high-level information may be able to cooperate with the user in managing a user’s file space better than at present.

The only system calls which are intercepted are file system calls, and process control calls in order to monitor all the processes created by a program. No attempt is currently made to monitor external effects or communications or signals. The prototype allows for concurrency in a limited way, using the techniques described in Section 6. Aliases are tracked using the Unix device number and inode number.

Further work needs to be done on monitoring and handling programs with external effects, including programs which cooperate with remote filestores or other services, and on security issues. Work is being carried out on adding further facilities based on the ability to monitor commands centrally. For example, it is possible to add an automatic `make` facility, without the need for the usual explicit `Makefile` dependency file.

Overall, the facilities presented make a significant contribution to an increase of intelligence in the management of files and programs, reducing the risks of loss and unpredictability.

REFERENCES

[1] Alexandrov, A. D., Ibel, M., Schauser, K. E. and Scheiman, C. J. (1998) Ufo: A personal global file system based on user-level extensions to the operating system. *ACM Trans. Comput. Syst.*, **16**, 207–233.

[2] Jones, M. B. (1993) Interposition agents: transparently interposing user code at the system interface. *ACM SIGOPS Oper. Syst. Rev.*, **27**, 80–93.

[3] Goldberg, I., Wagner, D., Thomas, R. and Brewer, E. A. (1996) A secure environment for untrusted helper applications—confining the wily hacker. *Proc. 1996 USENIX Security Symp.*, USENIX Assoc., Berkeley, CA.

[4] Faulkner, R. and Gomes, R. (1991) The process file system and process model in UNIX system V. *Proc. 1991 Winter USENIX Conf.*, USENIX Assoc., Berkeley, CA.

[5] Meyrowitz, N. and Van Dam, A. (1982) Interactive editing systems: part 1. *Comput. Surveys*, **14**, 321–352.

[6] Shneiderman, B. (1987) *Designing the User Interface: Strategies for Effective Human–Computer Interaction.* Addison-Wesley, Los Angeles.

[7] Foley, J. D. and Wallace, V. L. (1974) The art of natural graphic man–machine conversation. *Proc. IEEE*, **62**, 462–471.

[8] Norman, D. A. (1983) Design rules based on analyses of human error. *Commun. ACM*, **26**, 254–258.

[9] Card, S. K., Moran, T. P. and Newell, A. (1983) *The Psychology of Human–Computer Interaction.* L. Erlbaum Associates, Hillsdate, NJ.

[10] Goldberg, A. (1984) *Smalltalk80: the Interactive Programming Environment.* Addison-Wesley, New York.

[11] Bourne, S. R. (1983) *The Unix System.* Addison-Wesley, New York.

[12] Archer, J. E. Jr, Conway, R. and Schneider, F. B. (1984) User recovery and reversal in interactive systems. *ACM Trans. Program. Languages Syst.*, **6**, 1–19.

[13] Archer, J. E. Jr and Conway, R. (1981) *COPE: A Cooperative Programming Environment*. Technical Report TR81-459, Department of Computer Science, Cornell University, Ithaca, NY.

[14] Teitelman, W. and Masinter, L. (1981) The Interlisp programming environment. *Computer*, **14**, 25–33.

[15] Reiss, S. P. (1985) PECAN: program development systems that support multiple views. *IEEE Trans. Software Eng.*, **11**, 276–285.

[16] Pike, R. (1987) The text editor Sam. *Software—Practice and Experience*, **17**, 813–845.

[17] Gordon, R. F., Leeman, G. B. and Lewis, C. H. (1985) Concepts and implications of undo for interactive recovery. *Proc. 1985 ACM Ann. Conf.*, pp. 150–157.

[18] Vitter, J. S. (1984) US&R: a new framework for redoing. *IEEE Software*, **1**, 39–52.

[19] Yang, Y. (1988) Undo support models. *Int. J. Man–Machine Studies*, **28**, 457–481.

[20] Stallman, R. (1986) *GNU Emacs Manual*, version 17. Free Software Foundation. Inc., Boston, MA.

[21] Berlage, T. (1993) Recovery in graphical user interfaces using command objects. Arbeitspapiere der GMD, Sankt Augustin, Germany.

[22] Berlage, T. (1994) A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. Human Comput. Inter.*, **1**, 269–294.

[23] Prakash, A. and Knister, M. J. (1992) Undoing actions in collaborative work. *Proc. 4th ACM Conf. on Computer-Supported Cooperative Work*, Toronto, Canada, Oct 31–Nov 4, pp. 273–280.

[24] Spenke, M. and Beilken, C. (1990) An overview of GINA—the generic interactive application. In Duce D. A. *et al.* (eds), *User Interface Management and Design*, pp. 283–303. Springer-Verlag, Berlin.

[25] Thimbleby, H. (1990) *User Interface Design.* Addison-Wesley, New York.

[26] Loukides, M. and Oram, A. (1996) *Programming with GNU Software.* O'Reilly and Associates Inc., Farnham, UK.

[27] Date, C. J. (1995) *An Introduction to Database Systems.* Addison-Wesley, New York.