User Recovery and Reversal in Interactive Systems

JAMES E. ARCHER, JR.
Rational Machines
and
RICHARD CONWAY and FRED B. SCHNEIDER
Cornell University

Interactive systems, such as editors and program development environments, should explicitly support facilities that permit a user to reverse the effects of past actions and to restore an object to a prior state. A model for interactive systems that allows such recovery facilities to be defined precisely and user and system responsibilities to be delineated is presented. Various techniques for implementing recovery are described. Application of a general recovery facility to support reverse execution is discussed. A program development system (called COPE) with extensive recovery facilities, including reverse execution, is described.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*programmers workbench; user interface;* D.2.3 [**Software Engineering**]: Coding—*program editors;* D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; D.2.6 [**Software Engineering**]: Programming Environments; D.4.7 [**Operating Systems**]: Organization and Design—*interactive systems*; H.1.2 [**Models and Principles**]: User/Machine System—*human factors*

General Terms: Design, Human Factors, Languages

Additional Keywords and Phrases: Recovery, reverse execution, undo, checkpoint, editor

1. INTRODUCTION

Interactive systems, such as editors and program development environments, allow a user to construct and modify data objects (e.g., documents and programs) in real time. Since users make mistakes and change their minds, an important aspect of the design of such systems is support for facilities that permit a user to reverse the effects of past actions and to restore an object to a prior state. This capability has always been present in systems that create and then modify a temporary copy of an object. However, in such systems, the user must anticipate

This work was supported in part by The Defense Advanced Research Projects Agency under grant 903-80-C-0102 at Stanford and National Science Foundation grants MCS 80-03304 and MCS 81-03605 at Cornell.
Authors' addresses: J. E. Archer, Jr., Rational Machines Inc., 1500 Salado Drive, Mountainview, CA 94043; R. Conway and F. B. Schneider, Department of Computer Science, Cornell University, Ithaca, NY 14853.

recovery needs and deliberately save versions of the object to which recovery may be desired. Our concern here is with general recovery facilities that are both automatic and convenient.

There is increasing interest in such facilities. The INTERLISP system includes pioneering work on recovery [16], and some form of **undo** command is not uncommon in recent interactive systems [2, 5, 12, 13]. While it is probably useful to add facilities for recovery to existing interactive systems, a system designed from the outset with recovery capabilities could also do other things differently. Both the system and its users could be bolder in their actions. The system could take more initiative in performing actions on behalf of a user, and a user would be less hesitant to try powerful and perhaps unfamiliar commands.

Recovery has long been important in database management systems [6, 7, 11, 17]. Recovery in database systems is motivated by the possibility of system failures. Since failures are infrequent events, the corresponding recovery facilities can be expensive both in time and space and need not be especially easy to use. We are concerned with a user's recovery from his own prior actions, which we expect to be a frequent event, so recovery must be convenient and relatively inexpensive. Nevertheless, some of the techniques we describe are derived from approaches first developed for use in database management systems.

This paper is organized as follows. Section 2 presents a model for interactive systems that allows recovery to be defined precisely and user and system responsibilities to be delineated. Section 3 enumerates various useful restrictions on the types of recovery a user can request. Section 4 describes several implementation techniques for supporting recovery. An application of a general recovery facility that provides support for reverse execution of programs is the subject of Section 5. Sections 6 and 7 describe recovery facilities in two implemented systems. Section 8 explores an interesting generalization of the execution phase of our model. Conclusions are drawn in Section 9.

2. INTERACTIVE COMPUTER SYSTEMS

Below, we define a model of an interactive computer system. The recovery problem is then described in terms of that model. While this is not the most general model imaginable, it is simple and instructive.

2.1 Objects and Scripts

Interactive computer systems are used to create and modify information structures, which we call *objects*. The *state* of an object at some time is defined by the values of its components at that time, possibly including the position of one or more *cursors*. In order to view or change the state of an object, a user issues *commands*. The *execution of a command* causes the display of some portion of an object and/or a transformation of the object state. The effects of execution are assumed to depend only on the state existing when the command is executed, not on the manner in which that state was established.

The user's role in the interactive process is to construct a sequence of commands called a *script*.[1] The script specifies the transformation of the object from

[1] BRAVO [12] also employs a script (called a transcript). However, a script in BRAVO is intended solely as a way to recover the results of an editing session after a failure.

its initial state Q_0 to some other desired state Q. The system performs this transformation by *executing the script*, which involves executing each of its constituent commands in the order in which they appear in that script.

A script is constructed by using *metacommands*. These allow individual commands to be created, modified, reordered, entered into the script, and removed from the script. The execution of metacommands may involve interaction between the user and the system (prompts, error messages, etc.). Only the results of these interactions are stored in the script, not the interactions themselves. Note that the script is itself an object (text file), so metacommands are merely commands for editing this particular object. (One could go on to describe meta-scripts and meta-metacommands, but that does not serve our present needs.)

2.2 The Interactive Cycle

From time to time the user suspends the construction of a script and offers the system an opportunity to perform some execution. Later, the user regains control and resumes editing the script. Thus, the basic interactive cycle has two logical phases:

(1) *Edit*: user edits the script;
    submission terminates the edit phase.
(2) *Execute*: system performs some execution;
    control *returns* to the user when the execution phase terminates.

This cycle is repeated until the user is satisfied that the script will, upon execution, produce the desired state Q from the initial object state Q_0.

In principle, the user could complete the script in a single edit phase and submit it for execution. This is what occurs in a classical "batch" system. In an interactive system, since this cycle is repeated, the user can receive feedback from execution that could guide in further modifications to the script. New commands can be added to the end of the script and, if recovery facilities are present, other portions of the script can be changed.

2.3 Execution and Recovery

Consider any two consecutive cycles in the interactive process (see Fig. 1). In the editing phase of the first cycle the user constructs script S consisting of a sequence of n commands:

$$c_1; c_2; \ldots; c_n$$

Then, S is submitted for execution, during which it is partitioned into two sequences: E and P. E is the prefix of S containing commands that have been *executed*, and P is the remainder of S—those commands whose execution is still *pending*. An execution policy is called *complete* if after the execution phase P is empty and $E = S$; otherwise the execution policy is said to be *partial*.

Let S' be a script consisting of m commands that is produced in the next editing phase:

$$c'_1; c'_2; \ldots; c'_m$$

S' can be viewed as partitioned into two sequences U' and M', where U' is the longest prefix of S' that is also a prefix of S, and M' is the balance of S'. Thus,

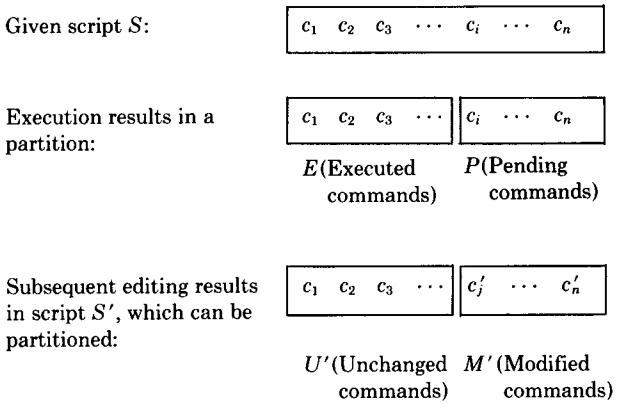

Fig. 1. Script model.

U' contains the prefix of S that is *unchanged* in S', and M' contains the first and other commands that were *modified* during the previous edit phase. If S' can be formed only by appending commands to the end of S, then the script modification policy is called *incremental*, and U' is identical to S. A prefix of S is said to be *committed* if the user is prohibited from subsequently changing any of its commands. Note that a committed prefix of S is necessarily a prefix of U'.

Any cycle in which E is not a prefix of U' leaves the system in an *inconsistent state*: the user has modified some command that has already been executed. Before the system can proceed, consistency must be reestablished. This process is called *recovery*. During recovery the object is transformed to a state that would have been produced by execution of E' on initial state Q_0, where E' is some prefix of E and of U'. Note that recovery is never necessary in any cycle where script modification has been incremental. Committing E by definition precludes the possibility of recovery. On the other hand, if script modification is not incremental, recovery may be required.

It is instructive to contrast our script model with the traditional view of the interactive process. There, each command is considered separately; the user first constructs a command and then submits it for immediate execution. In the script model, the user repeatedly submits versions of a script, where each version can differ arbitrarily from its predecessor. Moreover, in the script model any execution phase may require recovery and may involve execution of more than one command. The traditional view is a special case of the script model in which script modification is incremental and execution is complete. Note also that the script model cleanly separates responsibility for script modification from responsibility for execution and recovery: the *system determines* E during execution, the *user establishes* U' during script modification, and the *system determined* E' during recovery.

2.3.1 *Side Effects of Execution.* The execution of certain commands can have effects that are not reflected in the state of the object. We call these the *side effects* of execution. Since side effects can have consequences that are beyond the

control of the system, a system may not be able to recover entirely from the execution of a command with side effects.

The most common example of a side effect is communication of information beyond the boundary of the system. Once communicated, information cannot be "uncommunicated". A message directing one to "forget" information previously received does not result in a state of affairs equivalent to the original message never having been sent. Receipt of information from outside the system is also a side effect—although the system can forget information it has received, the sender may not be able to forget having sent it. Communication with the script's author might be exempt from concern in this regard, because as an active participant in the interactive process the author should understand that the validity of previous communications may be affected by changes made to the script. However, communication with other parties can cause more difficulty, since they probably will be unaware that the script has been modified and may have taken actions based on prior communication.

The fundamental side effect of execution is the passage of real time. Execution of a command takes time, and recovery takes additional time; so no recovery facility can really restore the universe to a state that existed earlier. Science fiction authors enjoy considering the effect of reversing time [8, 18]; we must be content with the more modest goal of restoring an object to an earlier state.

Although side effects can simply be ignored, this shifts responsibility to the user, who must either avoid submission of commands with significant side effects until such time as their execution will never be subject to recovery, or contrive to undo these side effects manually when recovery is necessary. Other strategies include

(1) *Commitment by the User.* A user could commit a prefix of the script containing commands with side effects, thereby relinquishing the privilege of subsequently modifying that portion of the script. Execution of the committed prefix would be safe, since recovery would never be required.

(2) *Commitment by the System.* The system could always commit a prefix of S that includes all commands with side effects, thereby precluding the possibility of recovery.

(3) *Buffer the Side Effects.* Side effects could be uncoupled from the object-transforming effects of execution by delaying their delivery. Pending side effects would be part of the object state, and therefore accessible and reversible. Their actual delivery would only take place in response to special commands, which of course could not be undone.

None of these approaches is always satisfactory. The side effect question appears to be quite difficult and merits more careful attention.

2.3.2 *Session Boundaries and Execution.* A *session* is an interval of more or less continuous interactive activity—from **logon** to **logoff.** Traditionally, the session has been an important epoch. Each session is self-contained—for the first cycle the script is empty; at **logoff** the entire script is committed, and hence at that time the system can execute commands without concern for side effects.

However, in several recent systems [2, 15] **logoff** merely signals a temporary interruption. No commitment or completion of execution is implied by **logoff,**

and at **logon** the user is presented with precisely the same state of affairs that existed at the time of the last **logoff.** Note that this does not deny the existence of significant epochs during the evolution of a program; rather, it decouples such epochs from sessions and gives the user explicit control over them.

These different views of a session affect recovery. The traditional session view provides a convenient point (session end) for the system to tidy up matters—commit, execute, and erase the script, deliver communications, etc. Alternatively, when sessions are not significant epochs, the system must contend with ever-increasing script lengths and be prepared to cope with modifications arbitrarily early in the script.

3. MODIFICATION OF A SCRIPT

The user's freedom to modify a script might be restricted in various ways. In Section 3.1 we describe a taxonomy of forms of modification that could be applied to a script. This is the basis for our subsequent discussion of implementation strategies, but it does not necessarily represent the set of primitive actions that should be exposed in the interface to the user. Then, in Section 3.2 we consider various recovery commands that might be offered to the user.

3.1 Types of Script Modification

For a given script $S = c_1; c_2; \ldots; c_n$ recall

(a) *Incremental modification*

$$S' = c_1; c_2; \ldots; c_n; c'_{n+1}$$

That is, S' is formed by appending a new command c'_{n+1} to S. This means that $U' = S$, M' consists of the single new command c'_{n+1}, and E is necessarily a prefix of U'. Hence, incremental modification corresponds to the traditional view of an interactive system and no recovery capability is required.

At the other extreme, the user can be allowed complete freedom in modifying the script:

(b) *Unrestricted modification.* To form S', S can be modified in any way:

(i) new commands can be inserted at any point;
(ii) existing commands can be deleted, changed, or reordered.

Unrestricted modification means that M' and S' could be identical—for example, S could be discarded and S' constructed from scratch. Since there is no restriction on the relationship between the commands in M' and those in E, recovery may be necessary in order to execute S'. We say that a system whose recovery facilities are powerful enough to permit unrestricted modification of S has *complete recovery* capability.

Other possibilities exist between these extremes. Some interesting ones are discussed below.

(1) *Single-truncate.* Two types of modification are allowed:

append:
$S' = c_1; c_2; \ldots; c_{n-1}; c_n; c'_{n+1}$
truncate:
$S' = c_1; c_2; \ldots; c_{n-1}$

However, **truncate** cannot be used on two consecutive cycles.

Thus, if **truncate** is used to construct S', then only **append** is allowed in the formation of the next version of the script, but **truncate** would be allowed for the version after that. This means that only the most recently appended command can be deleted. Moreover, since U' always contains $c_1; c_2; \ldots; c_{n-1}$ as a prefix, c_n is the only element of E that might not also be a command in U'. Therefore, recovery *might be* required when c_n is deleted from S in forming S'.

(2) *repeated-truncate* (*truncate**). Two types of modification are allowed:

append:
$S' = c_1; c_2; \ldots; c_{n-1}; c_n; c'_{n+1}$
truncate:
$S' = c_1; c_2; \ldots; c_{n-1}$

Unlike single-truncate, here **truncate** can be repeated as long as S is not empty. Truncate* is, therefore, substantially more powerful than single-truncate, in the sense that by performing a sufficient number of **truncate**s followed by **append**s, the user can accomplish any desired modification of S. Truncate* has the same power as unrestricted modification, although considerable work is required to exercise that power. For example, to make a change in command c_{n-2}, first c_n, c_{n-1}, and c_{n-2} must be deleted, and then c'_{n-2}, c_{n-1}, and c_n must be reinserted. Nevertheless, arbitrary modification is possible, so the system must provide complete recovery capability.

If truncate* were to be implemented, there would likely be a limit on the number of consecutive **truncate**s the system could honor. A particular implementation could be called a truncate^k system, for some integer k, if the length of U' must be at least $n - k$ commands. Single-truncate is a special case in which $k = 1$.

(3) *Truncate/reappend.* An auxiliary script R is introduced:

$$R = c_{r1}; c_{r2}; \ldots; c_{rp}$$

Only S' (and not R) is submitted for execution; unrestricted modification of R is allowed.

Three types of modification to S are allowed:

append:
$S' = c_1; c_2; \ldots; c_{n-1}; c_n; c'_{n+1}$
truncate: move the rightmost command of S to the left end of R.
$S' = c_1; c_2; \ldots; c_{n-1}$
$R = c_n; c_{r1}; c_{r2}; \ldots; c_{rp}$
reappend: move the leftmost command of R to the right end of S.
$S' = c_1; c_2; \ldots; c_{n-1}; c_n; c_{r1}$
$R = c_{r2}; \ldots; c_{rp}$

Truncate/reappend is similar to truncate* but includes provision to save (in R) the text of commands that have been truncated from S. This is just a matter of convenience. Truncate/reappend has the same power as unrestricted modification, but it is easier to use than truncate*. For example, although a change to command c_{n-2} still requires six script modifications, the last three are simple **reappend**s, rather than reentry of of c'_{n-2} , c_{n-1}, and c_n. Truncate/reappend is, in effect, a manual simulation of unrestricted modification. n **truncate** cycles

move the entire script S into R where it can be arbitrarily modified. m **reappend** cycles are then required in order to submit the modified script S'. This is in contrast to unrestricted modification where these **truncate**s and **reappend**s are unnecessary.

3.2 User Commands to Control Script Modification

The simplest interface to recovery facilities is to provide the single (meta) command **undo.** This command would have the same semantics as **truncate** in single-truncate modification described above. Several recent systems [5, 12] provide this type of interface, and it may well become fairly common.

Another option would be to give **undo** the semantics of **truncate** in truncate* script modification, but we know of no system that does this. However, some systems [13] provide a related facility that might be called *block truncate.* The user anticipates recovery needs by means of a **checkpoint** (meta)command, which causes a mark to be placed in the script. Performing an **undo** in such a system truncates the script through the last such mark. Thus, the effect is similar to truncate*, except for the necessity of anticipating the position of subsequent script modifications.

If the user interface includes a **redo** (meta)command as well as **undo**, then truncate/reappend script modification can be provided. The COPE system, which is described in Section 6, is an example of this. Although such an interface supports complete recovery, a designer might elect to provide increased convenience to the user by including commands such as

(a) **undo** back to c_i
(b) **undo** back to c_i and **append** commands ...
(c) **undo** back to c_i then **redo** c_j
(d) **undo** back to c_i then **redo** commands ...
(e) **undo** back to c_i then modify commands ... in R and then **redo** through c_j.

Although each of these modifications is achievable in a truncate/reappend system with an **undo/redo** interface, they would be unquestionably easier to accomplish with these higher level commands.

A user interface might also have a **commit** command to specify that a prefix of the script can be committed. Such a command would allow the user to relinquish the privilege of subsequently modifying that portion of the script.

4. RECOVERY STRATEGIES

We now turn to methods for supporting recovery. Except for incremental modification, implementation of any of the schemes described above requires recovery facilities. Because recovery strategies are largely independent of the form of script modification allowed, this aspect of the problem can be discussed in isolation.

4.1 The Complete Rerun Strategy

Conceptually, the simplest recovery strategy is first to restore the object to its initial state Q_0, thereby undoing the effects of all execution and making E' a null

sequence, and then to execute S'. Of course, this presumes that a copy of Q_0 is available. It also assumes that neither efficiency nor response time is important, since execution times can become quite large when this approach is used. When recovery is required, response time is a minimum of $O(n)$, where n is the length of the script, compared to constant response time without recovery. And if the number of recoveries is proportional to the length of the script, total running time is $O(n^2)$, compared to $O(n)$ without recovery.

The significance of the complete rerun strategy is that it establishes that recovery is possible. Only a concern for improved performance motivates consideration of other strategies.

4.2 Full Checkpoint Strategies

During the course of executing S, the system can periodically save the object state in anticipation of recovery needs. Such a copy of an intermediate state is called a *full checkpoint*; each full checkpoint Q_i is identified with E_i, the sequence of commands that was executed to transform the object from Q_0 to Q_i. Recovery then involves (1) finding some sequence E_i (preferably the longest) that is a prefix of U' for which there is a checkpoint and (2) restoring the object to that state.

An extreme version of this strategy would be to make a checkpoint after executing each command. Regardless of the change the user had made, there would always be a "perfect" checkpoint. That is, there would always be a checkpoint such that no command in U' need be rerun to execute S'. However, an enormous amount of time and space would be required to produce and store these checkpoints.

More practical strategies involve less frequent checkpoints. The determination of a good checkpoint interval is essentially a matter of minimizing the expected cost in terms of the size of a checkpoint, the expected reexecution time, and the expected frequency of recovery.

Note that in the strategies described above, multiple checkpoints must be saved—a subsequent checkpoint does not replace a previous one. However, once a checkpoint corresponding to a particular sequence E_j has been used in recovery, the checkpoints associated with any sequences of which E_j is a prefix can be discarded[2]; checkpoints associated with all sequences that are themselves prefixes of E_j must be retained since they might be needed for future recoveries. Note also that when script modification is limited to single-truncate, at most one checkpoint need be retained, because only the most recently appended command can be deleted. Thus, a single checkpoint taken prior to execution of c_n will always be sufficient. Undoubtedly, this is why single-truncate facilities are more popular among implementors than more powerful schemes.

An interesting variation of the checkpoint strategy involves storing only a small number of checkpoints. Each time a new checkpoint is saved, some prior checkpoint is discarded. The problem is to optimally and dynamically schedule the checkpoints.

[2] Although checkpoints corresponding to supersequences of E_j are not needed for recovery, they may be useful to avoid the reexecution of commands. Hence they may be worth saving.

4.3 Inverse Command Strategies

Both the full checkpoint and complete rerun strategies rely on the existence of a copy of some prior object state. To recover, the current state is discarded and replaced by a prior state, after which some commands in U' may need to be rerun. Alternatively, recovery could be accomplished by starting from the current state and executing "inverses" of the commands in E. Although this might intuitively seem like the most natural way to undo commands, there are difficulties with it.

Many commands do not have inverses, because they do not implement one-to-one mappings. For example, assignment ordinarily overwrites one value with a new one. Hence, there may be no single context-independent command that performs the inverse of an assignment. Commands comparable to "$X := Y$" are the rule; commands comparable to "$N := N + 1$" are fortuitous exceptions. Consequently, to execute the inverses of such commands, past states must be preserved. While this not impossible, it is effectively equivalent to the strategy described in the next section.

4.4 Partial Checkpoint Strategies

Execution of an individual command typically tranforms the state of only a few components of an object. Consequently, in order to be able to later undo the effects of a command, one need only save the states of those components that will be changed by execution of that command. This approach allows the *effect* of a command to be reversed, without determining the inverse of the command (although a partial checkpoint could be considered an encoding of a command inverse). Moreover, it represents a common mechanism capable of reversing any type of command.

There is a continuum of partial checkpointing strategies, ranging from full checkpointing through saving just the components that are changed. Under some circumstances it may be simpler to save larger fractions of the object state than is theoretically necessary. For example, when an object is naturally divided into sections (such as pages or fixed blocks for a file system) it may be more efficient to save entire sections in which components have been changed than to isolate and save only the individual components. An example of such an implementation is described in Section 6.

Recovery using partial checkpoints requires that consecutive checkpoints be restored in the opposite order of their creation. It is conceivable that the aggregate cost of performing a long sequence of such restorations could exceed that of full checkpointing, in either space or time. However, typically the depth of recovery will be a small portion of the length of the full script, and the ratio of size of a full checkpoint to a partial one will be very large, so partial checkpointing will be much more efficient than full checkpointing.

5. REVERSE EXECUTION OF PROGRAMS: AN APPLICATION OF RECOVERY

The possibility of running a program backwards has long been intriguing. When testing a program, if trouble is encountered it often would be useful to be able to reverse the direction of execution and search for the cause. True reverse execution

requires the ability to construct and execute the inverse of individual statements (assignment, for example) as well as the ability to retrace the forward flow of control. Although not yet commonplace, this capability has been demonstrated in several systems [3, 10, 15, 19]. Below, we describe how the same result can be achieved using a general recovery facility.

In an integrated program development environment it is usually possible to *execute the object* (a program) being manipulated. A script then contains various forms of **execute** commands calling for execution (rather than modification) of the object. Since the program is itself a form of script, the effect of its execution is to cause transformations on some set of files (objects). If we view these files simply as components of the basic object, then **execute** commands need not have special status, and our previous discussions of modification and recovery apply to execution as well as editing of a program.

In particular, in such systems there is usually a **single step** form of **execute** for executing only the next statement of the program. Undoing a **single step** command is equivalent to reversing the execution of a single statement. This means the general recovery facility provides reverse execution—and does so without extra machinery for its implementation or extra facilities for the user to master. As a practical matter, it is easier for the user (and the system, as well) to control execution in increments larger than a single statement. Hence, some form of **step**(j) command is often provided, allowing at most j statements to be executed. (Execution might terminate or pause in fewer than j statements for any of a number of reasons.) The user varies j during testing. Undoing **step**(j) reverses execution in the same increments as it advanced.

This form of reverse execution is available in the COPE system described below.

6. THE COPE SYSTEM: AN IMPLEMENTATION OF TRUNCATE/REAPPEND

COPE [1, 2] is an integrated program development environment consisting of a syntax-cognizant editor, an interactive execution supervisor, and a file system. It offers what is probably the most extensive recovery capability of any current development environment. In the terms defined above, COPE allows truncate/reappend script modification, and implements recovery by partial checkpointing. Command execution is complete: after each cycle $E = S$.

6.1 The User Interface in COPE

The COPE user interface involves a small number of commands, each of which is assigned to a dedicated special-function key. Two of these comands are **undo** and **redo**; these provide truncate/reappend script modification.

A system file called the *log* contains the concatenation of the S and R scripts, and there is a *log cursor* that points to the last command in S, c_n. The log is accessible to the user. Like any other file, it can be displayed; only the R portion is editable, however. (The S portion is protected from direct change by the user.) In addition, a reserved area of the display screen, called the *undo window* and labeled "Previous command:", always displays c_n. This makes the most recently submitted command visible without the necessity of displaying the log.

Undo recovers from the effect of the command displayed in the undo window,

and moves the log-cursor back one command. The new command constituting c_n is then displayed in the undo window.

Redo causes the last command that was **undo**ne (c_{r1}) to be resubmitted. The log-cursor is moved forward, making this command the new c_n, and the new c_n is displayed in the undo window.

The normal process of introducing a new command inserts that command immediately after the log-cursor in the log. The log-cursor is then moved forward and the undo window is updated accordingly. The R script is unchanged.

COPE departs from our truncate/reappend script modification model in two ways. First, all commands are not entered in the script. Some commands are paired so that one is a direct inverse of the other—for example, cursor-motion commands and the **condense** and **expand** commands that control the format of text display. It was considered unnecessary to use the general recovery facility for such commands because specific inverses exist.

Second, since COPE employs a full-screen editor, there is an interesting question as to what constitutes a command during editing. In COPE, the editing process is viewed as consisting of alternate **enter** and **submit** commands. **Enter** comprises all the changes made to the program text between successive **submit**s. **Submit** invokes the COPE parser to process the program text as it appears on the screen. (In fact, the parser is incremental and considers only the changes made during the prior entry.) Both **enter** and **submit** are invoked implicitly—there is no key for either. The **enter** command is implied by any character insertion, deletion or replacement; the **submit** command is implied by typing the "return" key and by other explicit commands such as **execute** and **resume**. Nevertheless, **enter** and **submit** are *bona fide* commands. Recovery from **submit** undoes the parser's response to modified text, restoring the program text to precisely the state the user attained at the end of the entry cycle. Recovery from **enter** undoes the user's modifications since the last system response.

This process is harder to describe than to use. Three fonts are used in the display of program text, as follows. Current user modifications are shown in reverse video; changes resulting from the previous parser response are shown as brightened text; and text unchanged in the most recent **enter/submit** cycle is shown in the normal font. The font used for each character of the display is considered part of the state of the system. Consequently, recovery reestablishes the fonts as well as the contents of the screen.

COPE permits execution of the program under development. By **undo**ing the **execute** or **resume** command, reverse execution is provided in the sense described in Section 5.

A typical session involves frequent switches between editing and executing the program under development. A program that is still incomplete can be executed, and a program that is being executed can be interrupted, modified, and then **resume**d from the point of interruption. Use of a single command script (the log) for both editing and execution commands allows recovery to retrace a user's path between editing and execution.

The recovery facility also permits COPE to view the user's activity as essentially continuous, with occasional dormant periods (between **logoff** and a subsequent **logon**). The **quit** command returns control to the operating system. The

next invocation of the system automatically undoes the effect of **quit**, leaving COPE in precisely the state that existed before the termination of the last session. For example, if a program was paused in execution at that point, it can now be **resume**d from the point of interruption. This is clearly the most reasonable way for any system to behave with respect to user sessions; it should not be the user's responsibility to save and restore the system state over the **logoff/logon** interval.

6.2 Implementation of Recovery in COPE

Implementation of the COPE recovery capability was remarkably straightforward; it uses partial checkpointing within the file system. COPE has a single file system, which serves both the user and the implementation. The user explicitly generates files for procedures, input data, and results. The system implicitly generates files for data supplied interactively during execution and for screen output produced during execution. In addition, all of the tables, stacks, etc., used by the editor, the parser, and the execution supervisor are implemented as files; even output to the screen is directed to a file, from which certain window segments are selected for display by a screen manager. *Everything* that goes on in COPE is reflected (only) in changes to some file.

Each file consists of a sequence of fixed-size blocks. Whenever any block is changed, a complete new copy of the block is saved, without overwriting the previous version. (Both primary and secondary storage are viewed as a single-level store, and blocks are treated the same way whether they happen to be in main memory or only in the backing store.) This means that the total effect of executing a command is reflected by a sequence of changes to blocks, which can be represented as a sequence of pairs of block identifiers naming the old and new versions of each block changed. This information is stored in the log, interleaved with the text of commands whose execution is thus described. The block list is never displayed to the user, however.

Recovery is then easily accomplished. Each **undo** causes the blocks changed by execution of c_n to be restored to their previous states and this portion of the block list to be deleted from the log.

Initially, all the space allocated to a COPE file system is on an internal free-list (in blocks). New blocks are obtained from this free-list as files require additional space and as blocks are changed. The log grows as commands and the corresponding block changes are recorded. When the free-list is exhausted, further demands for space are met by reclaiming the old versions of blocks changed by the earliest command in the log. This results in commitment of that command, so its text and block list are deleted from the log. In effect, during normal steady-state operation the recovery mechanism uses all the space available to COPE that is not used by other files. (Thus, the log serves as an internal free-list; to the outside world there is no apparent free space.) Consequently, the depth to which the system is capable of recovering depends entirely on the amount of space available for outdated copies of blocks and for the log.

During editing the user should seldom perform more than a few consecutive **undo**s, so it is unlikely that recovery will be thwarted by encountering the current limit of the log. However, in reverse execution the user may backtrack over a

long sequence of **execute** commands, and in so doing might reach the limit. There also tend to be more block changes per command during execution than during editing, which has the effect of reducing the recovery capacity during execution.

This implementation has the additional virtue that the entire mechanism is concentrated in the single module that implements the file system. Since all system activity is reflected in the file system, the recovery mechanism is capable of undoing anything the system does, without intruding on any other module of the system.

6.3 Automatic Error Repair in COPE

The recovery facilities in COPE have an interesting interaction with another major aspect of the system. In addition to providing a vehicle for exploring recovery facilities, COPE was intended to allow exploration of automatic error repair in an interactive environment. In the past, automatic repair has been primarily associated with batch systems [4]. It has been argued that such a facility would be unnecessary in an interactive system, where the user is immediately available and can be called upon to make his own repairs. The counterarguments to this are that (a) repair in this context can be viewed as a suggestion—one the user could easily reject if inappropriate, (b) a syntactically valid and complete program segment is often more informative than an error message, and (c) if the system *can* make a plausible repair, it is counterproductive to require the user to do so. Just because a user is accessible does not necessarily mean a system should be less helpful than it might be. This position has been explored in PL/CT [14] and INTERLISP [16], but is carried much further in COPE. Repair in COPE is automatic, inescapable, applicable throughout the system, and quite ambitious in the repairs undertaken.

While COPE is not yet in productive service, initial experience with a prototype implementation suggests there is important interaction between the recovery and repair facilities. The availability of simple and safe recovery encourages users to be bolder in exploiting the repair capability. Users deliberately abbreviate entries without always being sure what the system's response will be, secure in the knowledge that they can readily **undo** the result, expand their entry, and try again. It seems that the repair capability is much more frequently exercised on such deliberate "errors" of abbreviation than on inadvertent mistakes. The result is a truly interactive and cooperative program generation process—quite different from ordinary text entry. The interaction of extensive repair and simple recovery appears to fundamentally change the way this program development environment is used.

The recovery capability also had an important effect on the choice of algorithms for automatic error repair in COPE. Confident that user recovery from an inappropriate repair is simple and safe, COPE attempts more ambitious repairs than would otherwise be prudent. This repair facility is described in detail in [1].

7. RECOVERY AND REPAIR IN INTERLISP

The pioneering implementation of user recovery facilities is part of INTERLISP [16], a program development environment for the LISP language. INTERLISP

maintains a *history list* of commands as part of the *programmer's assistant.* Although the objective is the same, there is a significant difference between this history list and our script concept. A script represents a modifiable specification of a transformation; a history list is a literal record of the commands that were submitted. Therefore, although an INTERLISP user can undo the effects of commands that have been executed, the history of command submission and execution cannot be changed.

undo and **redo** are the basis of recovery in both COPE and INTERLISP, but their semantics are quite different in the two systems. In INTERLISP **undo** is a command rather than a metacommand so it is itself appended to the history list. Its execution causes the reversal of the effect of the execution of some other command(s). Not only can the INTERLISP **undo** be used like the COPE **undo**—to undo the execution of the previous command—but a user can specify that *any subsequence* of the history list be undone. If a suffix of the history list is specified, the effect is like a sequence of COPE **undo**s. However, when an internal subsequence is undone, it can be difficult to anticipate or even understand the object state that will be produced. Moreover, since **undo**s themselves can be present in the history list, a sequence to be undone can include one or more **undo**s. This can be extremely confusing.

The **redo**s in the two systems are also quite different. The COPE **redo** applies to commands that have previously been **undo**ne, while the INTERLISP **redo** is completely independent of **undo**. It is effectively just a "repeat" or "copy" command, which appends to the history list a copy of commands that are already present at an earlier point.

Appending commands to the history list corresponds to incremental modification in the script model. However, treating **undo** and **redo** as commands with arbitrary subsequences of the history list as arguments gives INTERLISP more power than unrestricted script modification. Undoing an internal subsequence of the history list represents a form of recovery that cannot be described in terms of our script model. INTERLISP can simulate the COPE forms of **undo** and **redo**, while COPE cannot simulate the INTERLISP commands.

The immutable history list and the modifiable script are distinctly different models. The history list model requires greater sophistication of its users and offers greater flexibility, but unfortunately it also provides ample opportunity to get in trouble. However, it is hardly surprising that facilities in a development system for LISP would be designed for a sophisticated user community.

It is probably more than coincidental that INTERLISP also offers automatic error repair. Although to contrast INTERLISP's DWIM facility with repair in COPE is beyond the scope of this paper, the interaction between repair and recovery in INTERLISP is notable and reinforces the effect observed in COPE.

8. GENERALIZATION OF THE EXECUTION PHASE

The separation of script modification from execution and recovery allows consideration of alternative models for execution without perturbing the user's view of the process. Up to this point, we have considered only execution that is complete ($E = S$) and *sequential* (commands are executed in the order they appear in the script). Relaxation of these restrictions could yield performance improvements by reducing the frequency or severity of recovery effort.

The recovery problem is reduced by deferring the execution of as much of S as possible for as long as possible, since commands that have not been executed never have to be undone. For example, consider partial sequential execution. In each cycle, E is a prefix of S. If E is also a prefix of U', then no command changed has been executed—hence no recovery will be required. The difficulty is that deferring execution of commands in S might deprive the user of useful side effects—feedback from execution of the script—thereby negating the principal virtue of the interactive process. Consequently, care must be taken to ensure that all commands that have useful side effects are executed. Note that commands that can raise an error condition must be considered as potentially having side effects. Hence, the number of commands that are assuredly free of all side effects may be small.

A more suitable policy is partial nonsequential execution. Under such a policy the system is allowed to execute commands in a different order from that in which they appear in S. This makes it possible to defer execution of "difficult" commands, yet still provide complete execution feedback to the user. Of course, in any nonsequential execution, the semantics of the script must be preserved. That is, the system must be limited to execution schemes for which the final state of the object is identical to the state that would be produced by complete sequential execution of the script. In general, it can be difficult to characterize allowable executions, so the effort to manage nonsequential execution may be greater than the recovery effort it seeks to avoid. Nevertheless, there are special cases that are easily identified and managed, so the question warrants further investigation. These implementation issues seem to be closely related to lazy evaluation [9].

Another approach to partial execution is to partially execute individual commands. For example, consider a command that transforms many components of the object, only a few of which will be immediately displayed to the user. The system might postpone transformation of other components until they are to be displayed, or until their state affects the transformation of some component that will be displayed. Again, the user would have the illusion of complete execution, while the system would have the advantage of reducing potential recovery effort by deferring execution.

9. CONCLUSIONS

General recovery is a facility whose time has come. It is a practical way to capitalize on the increasing economy of computer cycles relative to the cost of user time. Users are fallible and will continue to be so no matter how well engineered computer interfaces may be, so the necessity of recovery is inescapable. Ad hoc recovery by the user is slow, expensive, and unreliable; automatic recovery by the system can be convenient, reasonably economical, and reliable. Moreover, user fallibility is not just a characteristic of neophytes that is overcome with experience. The nature of errors may change with experience, but their occurrence does not. Therefore, automatic recovery should be seriously considered in any interactive system, regardless of its purpose or the expected sophistication of its users.

Recovery capability should not be viewed as an isolated characteristic. If

automatic recovery is integral to a system design from the outset, its designers can be bolder in other regards. Its users, too, can be more courageous in their actions, secure in the knowledge that safe and convenient recovery is always possible. For example, recovery allows systems to accept and execute commands that are less than complete and correct—in effect, to propose repairs that the user can either accept or reject.

A general recovery facility can apply to execution of objects, as well as to their construction. This can provide support for reverse execution of programs, which is a powerful diagnostic tool. There are substantial advantages to both user and implementor in providing a single mechanism capable of recovering from editing errors and of executing programs in reverse order.

The feasibility of general recovery has certainly been demonstrated, especially in INTERLISP and COPE, but also in other contemporary systems. It is difficult to give quantitative estimates of the performance penalty incurred by support for recovery facilities, since this depends on many factors. However, we can report that with a hospitable file system such as the one in COPE the cost is negligible—both for implementation and execution. Our experience suggests that the partial checkpoint strategy is dominatingly attractive and should (at least) be considered for any implementation.

COPE also demonstrates the feasibility of isolating a general recovery facility for an arbitrary interactive system by restricting it to a supporting file system. Such an interactive system would store all of its state in the file system. At first, this might seem a drawback; however, our experience is that this should be a simpler and more reliable method of implementation than incorporating the recovery facility directly in the interactive system.

Most recent interactive systems are "full-screen" oriented. It may not be obvious what user actions constitute commands when employing the script model for such a system. While in principle each insertion or deletion of a single character could be considered a command, as a practical matter this fine granularity is excessive. The definition of **enter/submit** commands, as employed in COPE, seems generally applicable to such full-screen systems, for purposes of recovery.

Our script modification model appears to be a convenient vehicle with which to describe recovery facilities. It seems preferable to the approach in INTERLISP where the history list is immutable, and **undo** and **redo** are commands that are themselves appended to the list. It allows questions of the semantics of undoing an **undo** or redoing a **redo** to be avoided, since metacommands do not appear in the script. Even so, the script model may be an unnecessarily flexible way to describe such facilities to a user. Similarly, unrestricted modification is a standard against which to judge particular facilities, but its seems too powerful and potentially confusing for general use. A user could too easily make a change early in a script, without fully comprehending how the execution of later commands would be affected. On the other hand, a truncate/reappend system provides effectively the same power, in a form that is easily understood and used. While a single-truncate system may suffice in many cases, the implementation of truncate/reappend is not much harder than single-truncate so it seems pointless to compromise.

The execution phase of our model can be generalized to admit the possibility of nonsequential and partial execution of commands. The separation of script modification from execution means that neither of these would affect the user's view of modification or recovery. Conceivably, such generalizations of execution could reduce the number of situations in which recovery is required, or the difficulty of performing recovery when it is necessary.

The recovery facilities in COPE and other recent systems are immediately and overwhelmingly attractive to users. These systems demonstrate that implementation of recovery facilities is not prohibitive, so it would be reasonable to expect many future systems to have comparable facilities. We are currently investigating the feasibility of including an extensive recovery facility in a document preparation system.

ACKNOWLEDGMENTS

We gratefully acknowledge the many helpful suggestions provided by our colleagues Mimi Bussan, Alan Demers, David Gries, Dean Krafft, Tim Teitelbaum, and Steve Worona. We also appreciate the many useful comments from the referees.

REFERENCES

1. ARCHER, J.E., JR. The design and implementation of a cooperative program development environment. PhD dissertation, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., 1981.
2. ARCHER, J.E., JR., AND CONWAY, R. COPE: A cooperative programming environment. Tech. Rep. TR81-459, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., June 1981.
3. BALZER, R.M. EXDAMS-EXtendable debugging and monitoring system. In *Proceedings AFIPS Spring Joint Computer Conference* (Boston, Mass., May 14–16), vol. 34. AFIPS Press, Arlington, Va., pp. 567–580.
4. CONWAY, R., AND WILCOX, T. Design and implementation of a diagnostic compiler for PL/I. *Commun. ACM, 16*, 3 (Mar. 1973), 169–179.
5. GOOD, M. Etude and the folklore of user interface design. SIGPLAN/SIGOA Symposium on Text Manipulation, *SIGPLAN Not., 16*, 6 (June 1981), 34–43.
6. GRAY, J., GRAHAM, R.M., AND SEEGMÜLLER, G. Notes on data base operating systems. In *Operating Systems An Advanced Course*, Bayer *et al.* (Ed). Springer Verlag, New York, 1979, pp. 393–481.
7. GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, I. The recovery manager of the system R database manager. *Comput. Surv. 13*, 2 (June 1981), 223–242.
8. HEINLEIN, R. *A Door into Summer*, New American Library, New York, N.Y., 1979.
9. HENDERSON, P., AND MORRIS, J. H., JR. A lazy evaluator. In *Proceedings 3rd ACM Symposium on Principles of Programming Languages* (Atlanta, Ga., Jan. 19–21), ACM, New York, 1976. pp. 95–103.
10. HODGSON, L., AND PORTER, M. BIDOPS: A bi-directional programming system. Tech. Rep., Dept. of Computer Science, Univ. of New England, Armidale, N.S.W., Australia, 1980.
11. KOHLER, W. H. A survey of techniques for synchronization and recovery in decentralized computer systems. *Comput. Surv. 13*, 2 (June 1981), 149–183.
12. LAMPSON, B. W. *BRAVO Manual, Alto User's Handbook*, Xerox PARC, Palo Alto, Calif., 1978.
13. MEDINA-MORA, R., AND FEILER, P. An incremental programming environment. *IEEE Trans. Softw. Eng., SE-7* (Sept. 1981), 472–481.
14. MOORE, C., WORONA, S., AND CONWAY, R. PL/CT—A terminal version of PL/C. Tech. Rep. TR 75-230, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Feb. 1975.

15. TEITELBAUM, T., AND REPS, T. The Cornell program synthesizer: A syntax-directed programming environment. *Commun. ACM, 24*, 9 (Sept. 1981), 563–573.
16. TEITELMAN, W. *INTERLISP Reference Manual.* Xerox PARC, Palo Alto, Calif. Dec. 1975.
17. VERHOFSTAD, J. S. M. Recovery techniques for Database Systems. *Comput. Surv. 10*, 2 (June 1978), 167–195.
18. WELLS, H. *The time machine.* In *Seven Famous Novels.* Knoph, New York, 1934.
19. ZELKOWITZ, M. Reversible execution as a diagnostic tool. PhD dissertation, Dept. of Computer Science, Cornell Univ., Ithaca, N. Y., 1971.

Received November 1981; revised July 1982; accepted April 1983