An Editor for Revision Control

CHRISTOPHER W. FRASER and EUGENE W. MYERS
University of Arizona

Programming environments support revision control in several guises. Explicitly, revision control software manages the trees of revisions that grow as software is modified. Implicitly, editors retain past versions by automatically saving backup copies and by allowing users to undo commands. This paper describes an editor that offers a uniform solution to these problems by never destroying the old version of the file being edited. It represents files using a generalization of AVL trees called "AVL dags," which makes it affordable to automatically retain past versions of files. Automatic retention makes revision maintenance transparent to users. The editor also uses the same command language to edit both text and revision trees.

Categories and Subject Descriptors: D.1.1. [**Programming Techniques**]: Applicative (Functional) Programming; D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution and Maintenance—*version control*; E.1 [**Data**]: Data Structures—*lists*; I.7.1 [**Text Processing**]: Text Editing

General Terms: Algorithms, Design, Languages, Management

Additional Key Words and Phrases: Editor, persistent data type, revision control, undo command

1. INTRODUCTION

Revision control systems [9, 13] help maintain the revisions of files that proliferate as software goes through its normal life cycle.[1] Revisions naturally form a tree; each new one is a descendant of the older one from which it was derived. Revision control systems generally save space by storing some revisions in their entirety and storing editor scripts to reconstruct the rest. Revision control systems may be thought of as editors for revision trees, for they allow users to examine, insert, and delete revisions in trees. The tree editor rarely shares much with the system text editor or editors.

[1] Following Tichy [13], this paper prefers the term "revision" to the term "version." A *revision* is a text file that arises through manual editing. A *version* need not be text nor involve human preparation. For example, an object module may be said to be a compiled version of a source file, but it is not a "revision" of anything. This paper uses the term "version" only in reference to past versions, because the term "revision" suggests newness and thus sounds awkward when juxtaposed with "past."

This work was supported in part by the National Science Foundation under grants MCS-7802545, MCS-8102298, MCS-8210096, DCR-8320257, and DCR-8511455.
Authors' current addresses: E. Myers, Department of Computer Science, University of Arizona, Tucson, AZ 85721; C. W. Fraser, AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974.

Two features related to revision control appear in some text editors. Some editors allow users to undo editor commands and return to the version of the text as it appeared one or more commands before [6]. Also, some editors automatically save backup copies of files to allow users to return to the version of the text as it appeared one or more sessions before [6]. The number of commands or sessions that can be reversed is normally limited. These limitations can be avoided by using a revision control system, though this defeats the purpose of undos and backups: recovering from situations that cannot be foreseen.

Revision control systems, undo commands, and automatic backup features all involve restoring previous versions of text files, but traditionally they have been implemented using distinct mechanisms. Distinct implementations allow problem-specific optimizations of programs that can be expensive in time (by requiring the reexecution of editor scripts) or space (by saving whole files), but they waste programming effort and force users to learn different models for tasks with at least some similarities.

This paper describes the editor EH, which presents a unified solution to the problems described above. EH never destroys the old version of the file being edited. Historically, this would have been prohibitively expensive, but EH represents files using a generalization of AVL trees called "AVL dags" [7]. This representation makes it affordable to retain past versions automatically.

EH must do more than merely save the revisions. It must also automatically organize them so that users can easily identify the desired version from the past when necessary. Transparency of operation is important so that users do not evade backup and thus lose the opportunities for backup afforded by AVL dags. To this end, EH automatically builds revision trees and uses the same editor command language to edit both text and revision trees. EH runs under UNIX.

2. AVL DAGS

EH represents each revision as a list of lines. A revision control system must either not change old files, or it must be able to reconstruct them. EH's implementation of lists efficiently accesses any line in any revision and also creates each new revision without either changing or copying the old version.

Possible implementations either retain revisions as constructive procedures or simply data. For example, RCS [13] stores one revision, but models others with procedural edit scripts to reconstruct them from the stored revision. Such systems are compact because edit scripts are usually short, but they can be slow when past revisions must be reconstructed. On the other hand, editors with simple backup store past revisions in their entirety, but they need more space and thus limit backup.

The data-oriented approach can be made more space-efficient by copying just that data affected by the operation. COPE's "partial checkpoint" scheme [1] exploits this idea in a straightforward way. It divides each list into blocks and copies only affected blocks.

The AVL dag scheme used here refines this mechanism to the point of guaranteeing worst-case efficient space and time performance. Each list of lines is modeled by an AVL tree. Each vertex contains a line, and inorder traversal of the tree delivers the list. The tree is height-balanced—that is, the heights of the

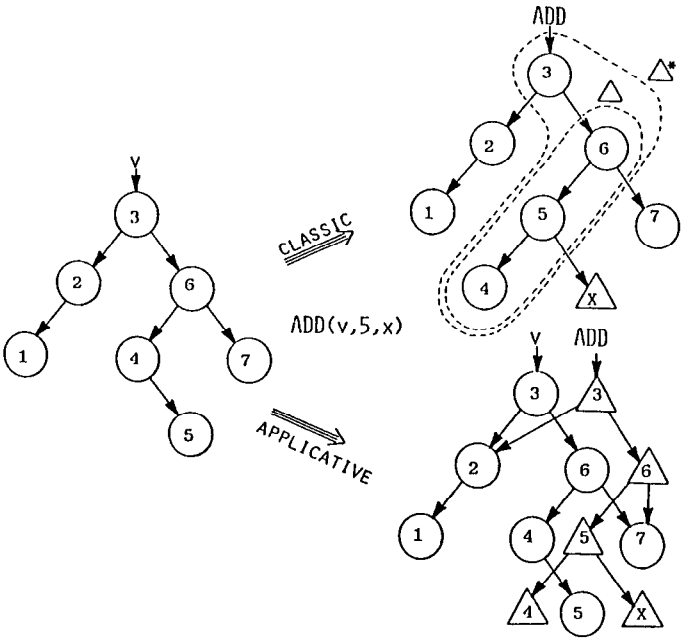

Fig. 1. The central idea.

left and right children of every interior vertex differ by at most one—so the height of the tree is $\Theta(\lg N)$ where N is the length of the underlying list.

Lists encoded as AVL trees can support a full range of list operations. Lists can be concatenated, sublists selected, and arbitrary elements accessed, inserted, and deleted all in $O(\lg N)$ time per operation [5]. However, the standard algorithms are not applicative. For example, concatenation consumes no space but destroys its operands through rebalancing and join operations. This choice may have been based on the assumption that preserving input trees would require copying them at an intolerable cost of $O(N)$ time and space.

However, not all parts of the operand trees are modified, so they can be preserved with much fewer than $O(N)$ vertices. Consider the AVL tree at the left in Figure 1. Suppose that a vertex labeled X is to be added as the right child of vertex 5. The procedure-oriented AVL algorithm modifies vertices 4, 5, and 6 to produce the result shown at the upper right. The list represented by their ancestor, 3 is also indirectly affected, because its right sublist is directly affected.

The set of vertices Δ^* modified by an AVL operation is the set of vertices Δ directly modified by the operation, plus their ancestors. Thus the result of the operation can be represented and the original tree preserved by copying just those vertices in Δ^*. In the example, this leads to the dag shown at the lower right of Figure 1. In general, Δ^* depends on the balances of search path vertices and their children and grandchildren. Nonetheless, for every AVL algorithm, Δ is $O(\lg N)$ because each takes $O(\lg N)$ time and Δ^* is also $O(\lg N)$ because each algorithm modifies a tree along at most two distinct paths.

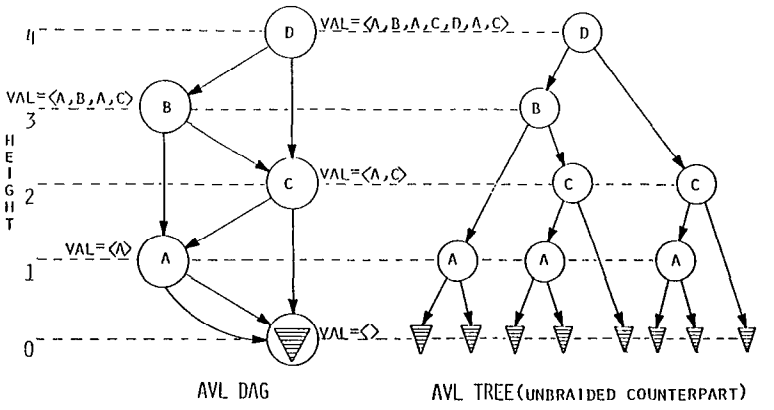

Fig. 2. An AVL dag example.

To preserve operands efficiently, applicative AVL operators build results that share subtrees with their operands. Thus, a set of lists is represented not as a forest of AVL trees, but as a dag that would yield such a forest if it were "unbraided" by replicating every vertex with in-degree greater than one. This structure is called an AVL dag and is illustrated in Figure 2. An AVL dag with N vertices may represent an AVL tree with as many as 2^N vertices. However, this observation does not affect the complexity of the algorithms because they never unbraid the dags, and they take time proportional to the *height* of the dag. An AVL dag algorithm is obtained by simply reformulating its procedural AVL tree counterpart so that it modifies a copy of every vertex it would normally modify and copies every ancestor of such a vertex. The resulting applicative operator has the same time efficiency as its procedural counterpart and requires only Δ^* or $O(\lg N)$ space for updates. Further details are given in [7].

EH models each revision as a pointer to a vertex in an AVL dag that encodes the current set of revisions. Figure 3 gives an example. Initially the revision tree consists of a single revision $V1$ of ten lines. Next, revision $V2$ is derived from $V1$ by an update that inserts line 0 before line 1. EH uses an applicative list primitive to perform the update and modify the AVL dag as shown. Finally, descendant $V3$ is derived from $V1$ by deleting line 8, resulting in a 17-vertex AVL dag modeling the 30 lines of $V1$, $V2$, and $V3$. The high degree of sharing guarantees space efficiency, while the balanced structure guarantees $O(\lg N)$ queries into *all* revisions. EH uses the following repertoire of applicative line-oriented operators to query, update, and remove revisions.

(1) FIND(T, a): Line
    Return the a^{th} line of revision T.
(2) SCROLL(T, a, Φ)
    Starting with the a^{th} line, pass successive lines of T to the "handler" procedure Φ until it returns a halt signal.
(3) REPLACE(T, a, s): Revision
    Replace the a^{th} line of T with line s.
(4) DELETE(T, a, b): Revision
    Delete lines a through b of T.

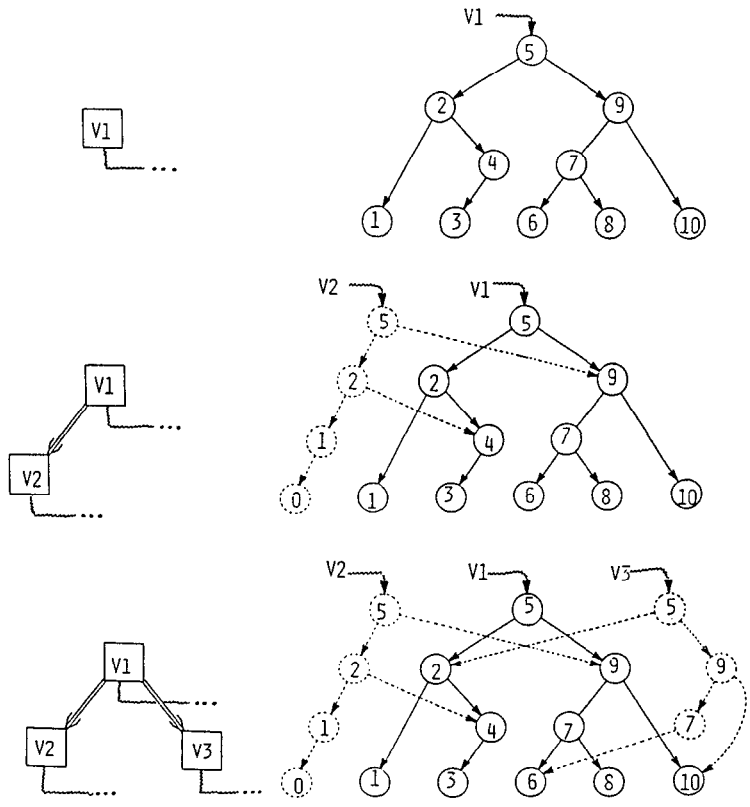

Fig. 3. AVL dags and version trees.

(5) INSERT(T, c, Φ): Revision
Insert the sequence of lines returned by the procedure Φ immediately after line c of T.
(6) MOVE(T, a, b, c): Revision
Move lines a through b of T immediately after line c.
(7) TRANSFER(T, a, b, U, c): Revision
Place a copy of lines a through b of T immediately after line c of revision U.
(8) REMOVE(T)
Remove (delete) T.

REMOVE(T) uses reference counts to garbage-collect every vertex modeling a line of T, but not of any other revision. For example, at the bottom of Figure 3, REMOVE($V1$) collects just the vertex for line 5.

Each update operator (3–7 above) can be expressed as a composition of the basic AVL dag operations: concatenation and sublist selection. For example, DELETE(T, a, b) could select the sublist from 1 to $a - 1$ and the sublist from $b + 1$ to the end of T, and then concatenate these two pieces together. Closer examination of the underlying mechanism reveals an approach that uses half the space and thus doubles the revisions that can be maintained.

At the center of AVL tree algorithms for concatenation and sublist selection is a primitive, JOIN(l, x, r). It combines AVL trees l and r and *juncture value* x

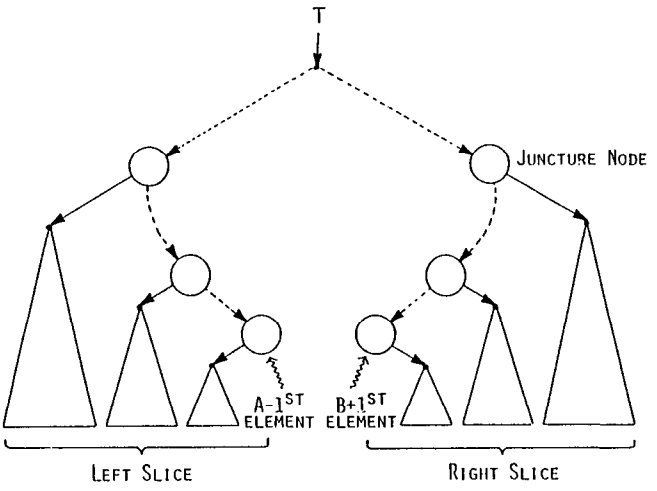

Fig. 4. Delete (T, A, B) slices.

into an AVL tree that models the concatenation of l, x, and r. The applicative version of JOIN is given in [7]. The important property of JOIN is that it adds a maximum of $|h(l) - h(r)| + 1$ vertices to the AVL dag where $h(v)$ is the height of the AVL tree corresponding to the vertex v.

Consider the operator DELETE. The portion of the tree to be retained is represented by the height-monotone *slices* depicted in Figure 4. The desired result is obtained by successively joining the subtrees of the slices together with their interspersed juncture values (the circles in Figure 4). When formulated as above, DELETE first joins the subtrees of the left slice together in height-increasing order. DELETE then joins the subtrees of the right slice. A final application of JOIN on the results of the preceding two steps produces the desired list. The sum of the space bounds of each application of JOIN telescopes to give a total worst-case bound of $2H + V$ vertices, where H is the height of the highest subtree in either slice and V is the number of subtrees in both slices.

Since concatenation is associative, any order of joins is correct. The improvement is obtained by using a better "join order" than the one above. Specifically, the subtrees in both slices are simultaneously joined in order of increasing height. For example, if the left slice is $\langle L_1, x_1, L_2, x_2 \rangle$ and the right slice is $\langle y_2, R_2, y_1, R_1 \rangle$, where $h(L_1) > h(R_1) > h(R_2) > h(L_2)$, then the optimal join order is JOIN(L_1, x_1, JOIN(JOIN(JOIN(L_2, x_2, NIL), y_2, R_2), y_1, R_1)). This merged joining is analogous to the sorting technique of merging two ordered lists. With this order, only $H + V$ vertices are added in the worst case.

For DELETE, the *height profile* of the slices forms a "V". For other operations the height profiles are more complex and merging opposing slices is more subtle. For example, the operator TRANSFER requires joining the four slices depicted in Figure 5, which form a "W". The optimal join procedure is as follows: (1) in a merged fashion join the slice $T_{\text{lca},b}$ and the portion of slice $U_{c+1,\infty}$ whose subtrees have height less than h(lca); (2) as in (1), join the lower portion of $U_{0,c}$ and all of $T_{a,\text{lca}}$; (3) join the results of (1) and (2) with the juncture value in vertex lca;

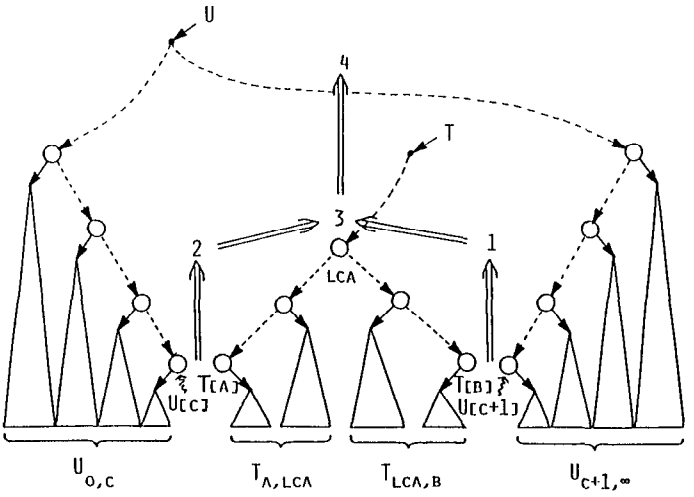

Fig. 5. Transfer (T, A, B, U, C) slices.

and (4), in a merged fashion join the upper portions of $U_{0,c}$ and $U_{c+1,\infty}$ using the result of (3) to "seed" the merge. With this join order, at most $H + L + V$ vertices are used where V is the number of subtrees in all the slices, H is the height of the highest subtree, and L is h(lca). The naive join order would have used $2H + 2L + V$ vertices.

With this order, it has been shown empirically that space performance is improved by 45 percent in the expected case. Table I lists the asymptotic performance bounds for the applicative editing operators and shows the expected-case space usage of both optimized and unoptimized operators. The expected space usage formulas were obtained by running linear regressions on the average value of Δ^* for a series of experiments involving geometrically dispersed choices of the parameters N, K, and so on. Depending on the operator, the regressions were over 500 to 1,200 data points. Each data point (an average value of Δ^* for some choice of parameters) was obtained by running 100 experiments on randomly constructed AVL trees of the appropriate size.

The formulas in Table I may be used to bound accurately EH's expected space usage when combined with a few additional implementation details. Character strings representing lines of text are maintained in a heap with an overhead of four bytes per string. The AVL dag is in a list space of 20-byte vertex records. Each record holds the height and size of the represented AVL tree, two pointers to subtrees, a pointer to a string, and a reference count. The height and reference count share one 4-byte word; the other fields take one word each. However, when stored in a file between sessions, the height, size, and reference counts are stripped, reducing nodes to 12 bytes each.

Reference [12] reports that revisions average 250 lines, and lines average 33 characters. Thus the AVL dag for a typical initial revision requires 250 vertices and heap strings. Reference [12] also reports that revision trees average 5 revisions, and that a typical edit session (or "delta") changes 22 lines in 4 blocks. A typical delta from Reference [12] is: "Insert 5 lines, Delete 5 lines, Insert 6

Table I.

Operator	Time	Δ^*: Order	Δ^*: Expected (unoptimized)	Δ^*: Expected (optimized)
FIND(T, a)	$O(\lg N)$	0	0	0
SCROLL(T, a, Φ)	$O(\lg N + K)$	0	0	0
REPLACE(T, a, s)	$\Theta(\lg N)$	$\Theta(\lg N)$	$.99 \lg N + .28$	$.95 \lg N - .28$
DELETE(T, a, b)	$\Theta(\lg N)$	$\Theta(\lg(N - R))$	$1.89 \lg(N - R) - .99$	$1.04 \lg(N - R) + .28$
INSERT(T, c, Φ)	$\Theta(\lg N + K)$	$\Theta(\lg N + K)$	$1.85 \lg N + K - .89$	$1.00 \lg N + K + .23$
MOVE(T, a, b, c)	$\Theta(\lg N)$	$\Theta(\lg N)$	$1.95(\lg N + \lg RS) - 1.19$	$1.13(\lg N + \lg RS) + .41$
TRANSFER(T, a, b, U, c)	$\Theta(\lg N + \lg M)$	$\Theta(\lg N + \lg R)$	$1.83(\lg N + \lg R) - .47$	$1.07(\lg N + \lg R) + 1.11$

where N = number of lines in revision T. M = number of lines in revision U. $R = |a-b|$. $S = |c-b|$. K = number of lines inserted or scrolled by Φ.

lines, Delete 6 lines." Table I implies that such a delta would add 43 vertices to the AVL dag and 11 strings to the heap, so EH models an average revision tree with 422 vertices and 294 heap strings for a total of 15,942 bytes.

Thus, for a typical revision tree, EH requires 1.93 times the storage needed to store a single revision as a text file, where RCS requires only 1.35 times the storage by using edit scripts to encode the deltas between revisions. Essentially, RCS deltas hold the lines that differ between revisions, and EH deltas hold these lines plus the dag vertices that differ between revisions. Thus EH consumes more space, though the fraction of space devoted to the extra dag vertices drops as text files grow because of the logarithmic performance of AVL dags. For example, if revisions are 1,000 lines, and deltas change 88 lines in 4 blocks, then EH uses only 1.80 times the storage for a single revision while RCS's relative consumption remains unchanged.

Furthermore, the space usage formulas for AVL dags overestimate consumption when intermediate revisions are removed. Creating K revisions and retaining all of them requires $O(K \lg N)$ new vertices. Some of these vertices are, however, required by only some intermediate revisions, so freeing intermediate revisions frees some vertices. Suppose only the first and last revisions are retained after a series of K updates. That is, consider the space performance of a delta that composes K basic operations. Certainly, the delta adds at most N vertices to the dag because, at worst, it replaces all N lines. At one extreme, $K = 1$ implies $O(\lg N)$ vertices are added. At the other extreme, $K = N$ implies $O(N)$ vertices are added. In general, the delta adds $O(K \lg(N/K) + K)$ vertices to the AVL dag when $K < N$.[2] Moreover, if the K updates are localized, then the degree of sharing increases and space consumption further improves. In the analysis above, the space consumption of the average delta was only an upper bound as the three intermediate states were not retained. The formulas predicted that the average delta would add 43 vertices; empirically, the average delta added only 35 vertices, for an overhead factor of 1.89, not 1.93.

EH uses more space than RCS, but it has immediate access to *all* revisions. RCS must reconstruct them by executing edit scripts. It is the combination of immediate revision access and reasonable space performance that permits EH uniformly to solve the problems of revision control, backup, and undos.

3. COMMAND LANGUAGE

EH was made to resemble a common editor, ED [4]. This decision was taken to avoid gratuitous proliferation of command languages, to show how revision control can be offered by a few natural changes to the command language of a well known editor, and to take advantage of an existing display-based front-end to ED [2, 3], which thus may also be used with EH. Because the screen-editing front-end is independent of the back-end used, it is not particularly relevant to

[2] Suppose the delta affects paths $p_1, \ldots, p_P$, and let $l(p_i)$ denote the length of path p_i. There are at most K vertices in the top lg K levels of the tree. Retaining all intermediate revisions would make $\sum_{i=1}^{P} \min(l(p_i), \lg K)$ copies of these K vertices, but the final result needs only one copy of each. Thus of the $\sum_{i=1}^{P} l(p_i)$ vertices used for the K intermediate revisions, at least $[\sum_{i=1}^{P} \min(l(p_i), \lg K)] - K$ will be freed, leaving at most $K + \sum_{i=1}^{P} (\lg N - \lg K)$ new vertices. But this is $O(K \lg(N/K) + K)$ because P is $O(K)$.

this presentation. To concentrate on the important issues, EH is presented here in its simplest form: as a line editor. The rest of this section presents the few ED commands needed to present EH. Readers familiar with ED may skip to the next section.

ED edits an internal buffer that represents a list of lines of text. Each ED command is a single line of text, holding line numbers, a command character, and possibly arguments, in this order. The line numbers delimit the lines on which the command operates. For example, the p command prints text, so 3p prints the third line, and 1,5p prints the first five lines.

Line number arithmetic is provided, "$" stands for the number of lines in the buffer, and "." stands for the number of the last line accessed (called the "current line"). Thus, for example, the command .+1,$p prints all lines following the current one.

The list below summarizes the commands used in this paper, including the default line numbers assumed when none are given.

(.)	a	add lines after given line.
(.,.)	d	delete given lines.
	e *file*	clear buffer and read in *file*.
(.)	n	print the screenful after the given line.
(.,.)	p	print given lines.
	q	terminate execution.
(.)	r *file*	read in *file* after given line.
	u *n*	undo the last *n* changes (see below).
(1,$)	w *file*	write given lines to *file*.

Commands that add text to the buffer (like a) are followed by the new text and a line holding a lone period. For example,

```
a
Here comes
another one.
```

inserts two new lines after the current one.

Files are written only in response to an explicit w command, and there is no implicit connection between writing a file and ending the session. The w command writes a file but does not automatically exit the editor; the q command exits the editor but does not automatically write a file.

ED has many other commands and features, which are not described here because EH can be presented without them. Readers interested in more detail on ED should see [4].

4. CREATING THE REVISION TREE

EH represents ED's buffer with an AVL dag that represents the list of lines currently in the buffer. When it gets a command to, say, delete a line, it uses the algorithms described in the second section and deletes the line from the current dag. It then records that the resulting dag is the next "current dag". The original dag is not lost.

This organization makes EH's undo command easy to implement. After each command (including the undo command), EH stacks the current dag pointer if it has changed. The undo command u *n* merely sets the current dag pointer to the *n*th element below the top of the stack; this dag pointer differs from the old one, so it is immediately stacked. The stack is never popped so that u commands may themselves be undone. For example, the command u 2 undoes the last two commands. If this is found to have undone too much, another u 2 undoes the first u command plus only the second of the two commands originally reversed. The stack is cleared when EH is entered or switched to a new file because it was thought confusing to undo commands from past sessions.

EH also maintains revision trees. Excepting the links recording the tree structure, nodes in revision trees have only two required fields. One records the revision's creation time, and the other holds a pointer to the dag that represents the list of lines of text forming the revision. To be of any use, the revision tree and the dags it references must be saved in a file between sessions, so EH does this automatically. It initializes new tree/dag files to hold a tree with exactly one node. This node's dag pointer references the empty list, and its creation time represents the earliest possible date. Everything in the revision tree descends from this original empty list. EH records a "current ancestor" that identifies where to enter new revisions in its revision tree. The determination of the current ancestor is described below.

EH was designed so that users may ignore its revision control features until they are needed. Thus EH builds its revision tree automatically, without explicit help from users. The w command is its cue to enter a node in its revision tree. Thus EH, like ED, uses w commands to separate short-lived revisions from long-lived ones, but EH increases the life spans of both. With ED, intermediate revisions between writes last only until the next ED command that modifies the buffer; with EH, these revisions are stacked and last the rest of the session or until a new file is edited. With ED, written revisions last until the file is rewritten; with EH, these revisions are entered in the tree and last until explicitly discarded (see the next section).

For transparency, EH's w command continues to create a conventional UNIX text file, at least in the default case. Thus, when EH receives a w command, it first writes a standard UNIX file exactly as ED would. Next it asks UNIX for the creation time of this new file. Then it creates a tree node holding this creation time and the current dag pointer, and it enters this node in the revision tree below the current ancestor. Finally, it makes this new revision the current ancestor. With each w command, the user may optionally enter a comment describing the changes made. This text is saved in the revision tree with the revision's creation date. The file name from the command is automatically prefixed to the comment.

The w command creates this structure for use by the commands that read revisions back in (e and r). When the user asks EH to read a file, EH first asks UNIX for the file's creation time. EH then searches its revision tree for a node with that creation time. If the search succeeds, then EH gets its text from the dag pointer in the tree node just found; no file I/O is done. If the search fails, EH reads the UNIX file and inserts its text into the current dag, using the algorithm

sketched in the second section. In this case, the read command acts like an a command where the text comes from a file instead of the command stream.

The revision tree may also be searched using the comment as the key. If the file name in a read command is a question mark, EH searches the revision tree for the latest revision with a comment that contains the text following the question mark. For example, the command

```
e ? x.c: Release 2
```

might retrieve the revision of file x.c that had been given the comment Release 2.9. Thus comments may include revision numbers.

The read commands must also identify the ancestor of any new revision that is created from the text just read. This determination necessarily depends on whether the editor's text buffer was empty and whether the read's tree search succeeded. There are three cases:

Buffer empty, search succeeded. In this case, the tree node found during the search represents the appropriate ancestor of any new revision created. EH records this node as the current ancestor.

Buffer empty, search failed. Here, the file just read had not been known to the revision control tree, so its ancestor is unknown. The new revision can only be a descendant of the primeval empty list, so EH records that the current ancestor is this empty list. This is how conventional files are first incorporated into EH.

Buffer nonempty. In this case, text is being read and inserted in a buffer that already contains some text. (The command must be r, not e, because e clears the buffer before reading.) Any new revision could be viewed as descending from *two* ancestors, the one just read and the one already in the buffer. Attaching any new revision below the one for the text already in the buffer seems preferable in the more common case where a small file is read into a large buffer, so EH implements this choice. Thus, in this case, EH does not change the current ancestor.

Writing text to files is redundant because the text is also saved in a dag between sessions. To save space, users may optionally request that write commands write no text. EH still writes an empty file because EH and its users still need a timestamp to identify the revision. A simple command script uses EH's read command to retrieve the "text" of such empty files for compilers, formatters, and so on. Given a (presumably empty) target file plus a file holding a revision tree and its dags, this script searches the revision tree for a node with the target file's creation date and outputs the text from the corresponding dag. The same script naturally retrieves revisions by comments (and thus revision numbers) as well.

5. EDITING THE REVISION TREE

The mechanisms of the previous sections, though internally sophisticated, are transparent to users. So far, EH appears to be ED with an undo command,[3] except to users who have deliberately chosen to comment revisions or write empty files.

[3] Actually, to save implementation time, a few of ED's features were omitted from EH, mainly those that involve regular expressions. The omitted features are unrelated to EH's relevant features. Adding them should be straightforward.

As with conventional editors, users can also back up and fork off new revisions from past ones *if* they have written each distinct revision to a distinct file. Overwriting these files discards the corresponding creation dates and, as in conventional editors, prevents the recall of past revisions, at least through the mechanisms presented thus far. To fully exploit *automatic* revision retention, users must be able to back up to old versions for which no external copy remains, and they must be able to fork off new revisions from there.

EH offers these new features by allowing users to edit revision trees. Again, to avoid gratuitous proliferation of command languages, ED's command language is used to edit revision trees.[4] Because ED's buffer is an unnested list, it cannot directly represent a full revision tree, which is a nested list. Thus at any point in time, the "lines" in EH's buffer represent either the ancestors or descendants of some revision. (The identification of the current revision is included with the ancestors.) The identification is by creation date plus any comments entered when the revision was created. Three new commands switch the buffer between text, ancestors, and descendants:

(.) A edit the ancestors of the given revision.
(.) D edit the descendants of the given revision.
(.) T edit the text of the given revision.

If the buffer currently represents text, the leading line number is not used because the line represents text, not a revision. Instead, the ancestors or descendants of the current revision are edited.

For example, when editing a simple buffer of the source code for a revision of EH, the command A switches to a buffer in which the "lines" identify the ancestors of this revision. At this point, the command 1n might print

```
10:00:00  Sun 8/19/84  The primeval empty list.
12:00:00  Sun 8/19/84  eh.c: Read in eh.
12:10:00  Sun 8/19/84  eh.c: Fixed undo stack underflow bug.
```

if EH were incorporated in the revision tree shown in Figure 6. Now the command 2T switches the buffer to the text of the second revision, EH's source code. This command thus backs up to a revision of which no external copy may exist. If a new revision of this code is written, EH will naturally create a new fork in the revision tree for it. When editing EH's source code, the command A edits the ancestors of this initial version, so a 1n now prints just

```
10:00:00  Sun 8/19/84  The primeval empty list.
12:00:00  Sun 8/19/84  eh.c.: Read in eh.
```

because this version of code has fewer ancestors than the one whose ancestors were displayed above.

The D command is like the A command except that it looks down the tree instead of up. For example, when editing the primeval empty list, D edits the descendants of this list, namely, the originals of all files incorporated into EH. At this point, the command 1n might print

```
11:00:00  Sun 8/19/84  ed.c:  Read in ed.
12:00:00  Sun 8/19/84  eh.c:  Read in eh.
```

[4] The last section discusses alternatives to this choice.

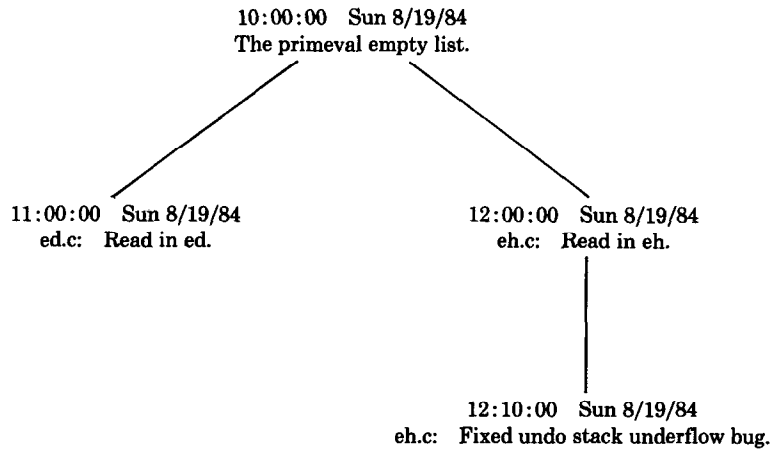

Figure 6

Now the command 1D edits any descendants or revisions of ED's original source code.

Like some ED commands, A, D, and T commands can be followed by a p or n suffix to print one line or one screenful of the buffer after executing the command. Thus users can simultaneously switch to a new buffer and orient themselves. For completeness, the A, D, and T commands are all legal whether EH is editing text, ancestors, or descendants. When editing text, T commands are naturally no-ops, A commands edit the ancestors of the buffer, and D commands edit the descendants of the buffer, which can exist only if the buffer holds some previously established revision, since a buffer that has not yet been written out cannot yet have descendants. When editing ancestors or descendants, T commands edit the text of the revision identified by the given line, and A and D commands edit the ancestors and descendants of the revision identified by the given line. After the switch, the "current line" is the first line to make the n suffix useful.

When editing ancestors or descendants, the standard d command naturally prunes the revision tree. When editing descendants, the revisions corresponding to the given lines are deleted, as are their descendants, recursively. When editing ancestors, the situation is slightly more complex. For example, the ancestors of revision 2d in Figure 7 form a list of length 5, including the primeval empty list and revision 2d itself.

For example, when editing this list, the command 3,4d must delete the revisions identified by lines 3 and 4 and, naturally, all of their descendants recursively, *except* the revision subtree identified by line 5. This subtree must be reattached somewhere, and the spot just under the first undeleted line (line 2) is natural. Figure 8 shows the result. EH refuses to delete the primeval empty list so that it will always have a place to attach new revisions.

Internally EH implements editing text, ancestors, and descendants as three modes, but the user sees a modeless editor [10, 11], for the commands used to edit a revision's past or future are the same as those used to edit text. Instead of

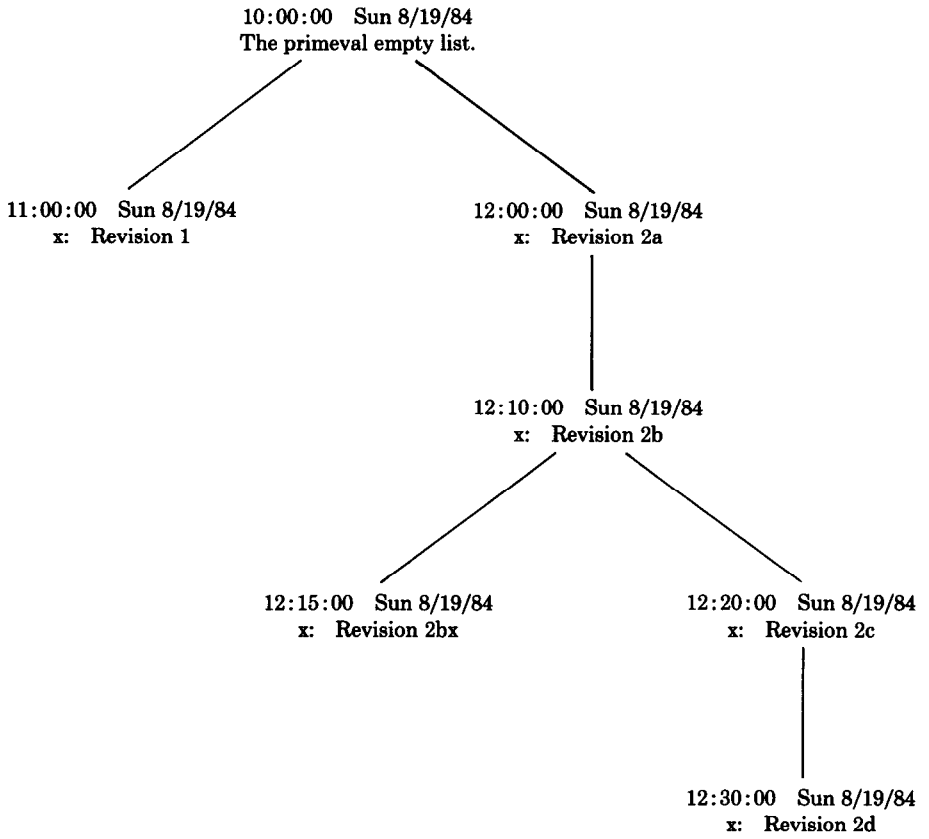

Figure 7

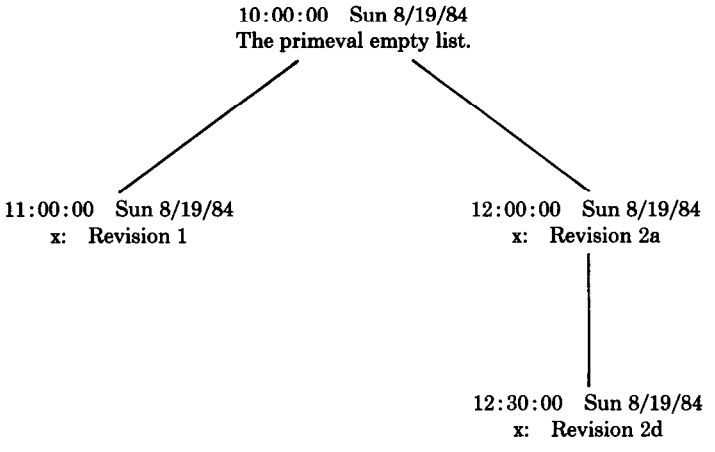

Figure 8

seeing three modes, the user sees a single command set that edits three different data structures. All commands that add new text to the buffer are disabled when editing lists of ancestors or descendants, but this is appropriate because the revision tree is augmented only implicitly, by w commands issued when editing text.

6. DISCUSSION

During EH's development, alternatives to some design decisions were explored and, in some cases, implemented. Some of the results may be instructive.

6.1 Identifying Revisions

The users of most revision control systems identify the revision that they wish to edit by giving a combination of file name and revision number. If the revision number is omitted, the latest revision of the file is generally retrieved. EH users identify the revision to edit by selecting one from a list of creation times and optional comments. Again, the default revision is the latest one.

EH does not number revisions automatically because some users may want EH merely for simple automatic file back up. They may not need revision numbers, so they should not be required to learn about them. This design decision is easily reversed by automatically prefixing revision numbers to the comments entered with w commands. Even now, users may include revision numbers in such comments, and this process may be partly automated by using a simple command script that freezes a release by invoking EH on a series of source files and issuing a w command for each file with a comment like Release 2.0. Since retrieval by both users and command files can be based on any substring of the comment, the comment is, in effect, a very general symbolic revision number.

An early version of EH attempted to record the development of a file by associating with each file a list of pointers to the dags that represented the file and its ancestors. To avoid interfering with other programs like compilers or formatters that would process the file, the pointers were stored in a separate but parallel file. For example, the dag pointers for the file x were written to the file x.hist. Dag nodes hold reference counts so that they may be freed when they are no longer referenced; writing dag pointers to files like x.hist naturally required incrementing these reference counts. Unfortunately, conventional file system utilities could be used to copy or delete x.hist, thus changing the number of references to some dags without informing EH of the need to adjust the reference counts, which were stored with the dag in another file.

Using creation times and comments to identify revisions avoids the problems mentioned above. Operating systems automatically associate creation times with files, so it is not necessary to store a separate revision identification for a file to tell whether it has been incorporated into a tree/dag file. The text files are normal UNIX text files, and tree/dag files hold all existing dag pointers.

Creation times and comments complement each other as means to identify revisions. For recent revisions, creation times are enough. For example, users often wish to see the version of some text as it existed a few sessions or days ago, and a simple creation time is enough to identify such recent revisions. Creation times are not enough to distinguish more distant revisions, but comments entered

for important events in the development of software easily identify these revisions.

Indeed, it might be desirable to use comments to guide semiautomatic tree pruning. Consider the following protocol for adding a new feature to source code under the control of EH. The programmer edits the code and describes the change with the first W command. If subsequent testing reveals bugs, the programmer fixes them. Since these are more likely to be viewed as intermediate developments, they are less likely to be commented. Once the bugs are fixed, a simple utility could move the last comment down to the corrected leaf where it really belongs and mark the truly intermediate revisions for deletion once their age passes a certain limit.

Assuming sufficient accuracy, creation times uniquely identify revisions. EH runs under a version of UNIX that maintains creation times to the nearest second. This granularity makes it virtually impossible for even fast typists to create two distinct revisions with the same timestamp, but stored command scripts can do so. For this reason, EH pauses for one second before creating a new revision if another revision was created in the same second. This "fail-safe" mechanism is seldom activated.

6.2 Editing Trees

The other central decision that emerged only gradually was the decision to edit revision trees by editing unnested lists of ancestors or descendants.

One early version of EH presented revision trees whole, using indentation to display nesting. Unfortunately, EH's revision trees are deep but not bushy, because every write command adds depth, but only forking a parallel branch adds breadth. Deep trees gave unwieldy indentations so this approach was abandoned.

Like many decisions regarding user interfaces, different tastes and different technologies might yield different tree editors. EH was designed around an editor for common alphanumeric terminals, but large graphics terminals might display and edit trees pictorially. The canonical pictorial represention of trees would waste display space because the trees are so much deeper than they are broad. Vertical display space could be conserved by omitting explicit down-going arcs when a revision has only one descendant (as is often the case). The arc would be represented implicitly by placing the creation time and comment for the two adjacent revisions on adjacent lines. Explicit arcs would be used only at forks in the tree, and the text inside each "node" would show the historical development of the revisions between forks.

6.3 Access Control

Typical revision control systems support access control to help prevent cooperating programmers from overwriting one another's work. For example, RCS [13] offers commands to check files out and back in again. If a file is to be modified, it is "locked" on check-out, and it remains locked until it is checked back in. A file may have at most one lock at any time.

Access control must preclude simultaneous changes to the same revision, but still allow simultaneous changes to different revisions. The latter requirement complicates the addition of access control to EH because EH now reads a complete

dag, which represents many revisions. Distinct invocations of EH may have distinct copies of one dag, which makes it hard to keep the copies consistent and hard for the invocations to know which revisions are being edited by other invocations. One solution is to split EH into two processes: a front-end command interpreter and a back-end "dag server," which would store the dag and respond to messages that correspond to the primitives that make individual changes to it. Two-process editors can be implemented efficiently [2, 8].

Multiple users editing distinct revisions in the same dag would have distinct front-end processes but would share a common dag server. That is, the first invocation of EH would create a dag server for its tree/dag file and then replace that file with some identification of the dag server. Subsequent invocations of EH looking in that tree/dag file would take that identification as their cue that their dag server already exists. The dag server should keep a count of the number of front ends that it is servicing, so that it can tell when it is no longer needed and when to write the tree/dag file back out. EH would need to read and write tree/dag files in a critical section; typical revision control systems have similar requirements.

While explicit commands to check files out and back in again could be implemented if desired, they would not be strictly necessary. Messages to the dag server would naturally name the revision tree node corresponding to the "current ancestor" described in Section 4. The dag server would allow multiple front ends to make read requests involving one revision tree node, but it could take modification requests as its cue to lock the revision tree node named in the message. If that node had already been locked, the back end would report an error back to the front end.

That is, simply invoking EH on a revision "checks it out" implicitly, so no explicit check-out command is needed to examine a file. Sending a modification command to such an invocation of EH could implicitly lock the revision if possible, so no explicit locking command would be needed. Finally, sending a W command to such an invocation of EH would implicitly check the revision back in and unlock the revision if necessary. In this way, EH's transparency of operation could be extended to include transparent access control.

ACKNOWLEDGMENTS

The authors thank Dave Hanson and the referees for numerous suggestions that improved both EH and this presentation.

REFERENCES

1. ARCHER, J. E., CONWAY, R., AND SCHNEIDER, F. B. User recovery and reversal in interactive systems. *ACM Trans. Program. Lang. Syst. 6*, 1 (Jan. 1984), 1–19.
2. FRASER, C. W. A compact, portable CRT-based text editor. *Softw. Pract. Exper. 9* (Feb. 1979), 121–125.
3. FRASER, C. W. A generalized text editor. *Commun. ACM 23*, 3 (Mar. 1980), 154–158.
4. KERNIGHAN, B. W. A tutorial introduction to the UNIX text editor. In the *Unix Programmer's Manual, Seventh Edition, Volume 2A*, Bell Labs., Murray Hill, N.J., Jan. 1979.
5. KNUTH, D. E. *The Art of Computer Programming; Volume 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973.

6. MEYROWITZ, N., AND VAN DAM, A. Interactive editing systems: Part I. *ACM Comput. Surv. 14*, 3 (Sept. 1982), 321–352.
7. MYERS, E. W. Efficient applicative data types. In *Conference Record of the 11th ACM Symposium on Principles of Programming Languages* (Jan. 1984), ACM, New York, 1984, 66–75.
8. PIKE, R. The Blit: A multiplexed graphics terminal. *AT&T Bell Lab. Tech. J. 63*, 8 (Oct. 1984), 1595–1631, Part 2.
9. ROCHKIND, M. J. The source code control system. *IEEE Trans. Softw. Eng. SE-1*, 4 (Dec. 1975), 364–370.
10. SMITH, D. C., IRBY, C., KIMBALL, R., AND VERPLANK, B. Designing the Star user interface. *Byte 7*, 4 (Apr. 1982), 242–282.
11. TESLER, L. The Smalltalk environment. *Byte 6*, 8 (Aug. 1981), 90–147.
12. TICHY, W. F. Design, implementation, and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering* (Tokyo, Sept. 1982), IEEE, New York, 58–67.
13. TICHY, W. F. RCS—A system for version control. *Softw. Pract. Exper. 15*, 7 (July 1985), 637–654.

Received November 1984; revised August 1986; accepted August 1986