

Molloy College DigitalCommons@Molloy

Faculty Works: Mathematics & Computer Studies

6-4-1984

CONCEPTS AND IMPLICATIONS OF INTERACTIVE RECOVERY


Robert F. Gordon Ph.D.

Molloy College, rfgordon@molloy.edu

George B. Leeman Jr

Clayton H. Lewis

Follow this and additional works at: https://digitalcommons.molloy.edu/mathcomp_fac

 Part of the [Graphics and Human Computer Interfaces Commons](#), [Other Computer Sciences Commons](#), and the [Partial Differential Equations Commons](#)
[DigitalCommons@Molloy Feedback](#)

Recommended Citation

Gordon, Robert F. Ph.D.; Leeman, George B. Jr; and Lewis, Clayton H., "CONCEPTS AND IMPLICATIONS OF INTERACTIVE RECOVERY" (1984). *Faculty Works: Mathematics & Computer Studies*. 20.
https://digitalcommons.molloy.edu/mathcomp_fac/20

This Research Report is brought to you for free and open access by DigitalCommons@Molloy. It has been accepted for inclusion in Faculty Works: Mathematics & Computer Studies by an authorized administrator of DigitalCommons@Molloy. For more information, please contact tochter@molloy.edu, thasin@molloy.edu.

RC 10562 (#47293) 6/4/84
Computer Science 46 pages

Research Report

CONCEPTS AND IMPLICATIONS OF INTERACTIVE RECOVERY

Robert F. Gordon
George B. Leeman, Jr.
Clayton H. Lewis

Office Applications Research, Department 511
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (i.e., payment of royalties).

RC 10562 (#47293) 6/4/84
Computer Science 46 pages

CONCEPTS AND IMPLICATIONS OF INTERACTIVE RECOVERY

Robert F. Gordon
George B. Leeman, Jr.
Clayton H. Lewis

Office Applications Research, Department 511
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

ABSTRACT: When working interactively on the computer, it is valuable to be able to undo a series of commands in order to return to a previous state. We identify contradictions and limitations in the basic concepts of undo. We introduce three types of undo functions with which we examine the characteristics of undo, explain these limitations, and determine the minimum requirements for a recovery facility. Then we discuss the implications of undo for user interfaces and suggest auxiliary functions to display and simplify the resulting history structure and to view and recover prior states.

CONCEPTS AND IMPLICATIONS OF INTERACTIVE RECOVERY

1. Introduction

Editing systems provide the capability to create and revise documents (text, programs, graphics) interactively on the computer. The user can issue one command that will change objects throughout the document, delete objects, insert new objects, or rearrange existing objects. After making a change, the user may find that he made an error or prefers the prior version. Therefore, when using an interactive system, it is valuable to be able to undo a series of commands and to return to a previous state of the document. With this recovery mechanism, the user can remove the effects of mistakenly-issued commands, and he has the freedom to experiment until satisfied with the final product. Most interactive systems provide limited recovery by requiring the user to save versions that he thinks he may need again before making changes. We will discuss an undo capability that allows the user to recover to a previous state without any prior actions.

There are experimental editing and formatting systems, such as Bravo [9], Etude [4], PEDIT [6], P-EDIT [8], and POLITE [14], that have undo capabilities, and the programming language extensions Interlisp [16] and Decilisp [7] support the undo facility. Medina-Mora and Feiler [11] discuss a program development environment which allows the user to restore a program to a predetermined checkpoint and re-execute it from that point. Wertz [18] presents an interactive programming environment which keeps enough information to allow the programmer to return to or modify any prior version. Archer, Conway and Schneider [2] describe a general undo facility using a script of commands that can be modified and re-executed.

They have implemented this recovery procedure in the program development system COPE. Vitter [17] introduces an interactive recovery system which displays the choices available to the user at each point in the recovery to assist him in selecting the desired state. He provides a direct way for the user to modify without restriction and re-execute the command history by introducing a "Skip" command to bypass selected previous commands. Leeman [10] defines and determines properties of four operators, which can be added to programming languages, representing two conflicting interpretations of the meaning of undo.

More systems are being planned to have undo capability, and there are commercial systems, such as Apple's Lisa [1,15], the Personal Editor for the IBM PC [5], and the software package Valdocs on the Epson QX-10 [13], with limited forms of undo.

The implementations of undo can be classified based on the effect of issuing two consecutive undo commands. This separation arises because there are contradictions and limitations in the fundamental concepts of undo. We will examine the characteristics of the undo command, explain these limitations, determine the resulting minimum requirements for an undo facility, and discuss the implications of undo for user interfaces.

In the next section, we examine some naive concepts of an undo command and show the basic contradictions inherent in these concepts. Section 3 describes three types of undo that salvage as much as possible from the naive concepts. Sections 4 and 5 examine and contrast the characteristics of these different types of undo. In section 6, we discuss the implications of undo for users and suggest facilities to enhance its use.

2. The Undo Command

In section 2.1, we define the basic property that an undo operation must satisfy. We then examine in section 2.2 other properties that one would like undo to possess. We show that some of these properties are contradictory, so that a single undo command cannot meet all of one's recovery expectations. In section 2.3, we analyze the recovery capabilities of undo commands. We shall discuss these notions within the context of a familiar task, namely, the editing of a document. However, it should be obvious that the concepts are application independent.

2.1 Basic Undo Property.

We first determine what is meant by applying undo to a document state. We begin with user-issued operators o belonging to a set O . Let K be the subset of O consisting of the editing commands k that operate on the states of the document; in addition to k , O will include other operators such as undo itself. We will use the letters f and g to distinguish different editing commands in K . Given the set S of all states of a document and the set K of editing commands with domain and range S , we would like to define an undo command u that recovers the prior state of the document:

$$uk(s) = s \quad \text{for any } s \text{ in } S \text{ and } k \text{ in } K. \quad (2.1)$$

(This notation lists the commands as issued from right to left.)

Unfortunately this simple idea won't work. In order to determine the prior state s , we need to know more than just the current state ks .

For example, let state s_a consist of the string 'abcd' and state s_b consist of the string 'bacd'. If f is the command to change all 'a' to 'b', then $f(s_a) = f(s_b) = 'bbcd'$. That is, there exist states s_a and s_b and commands f such that:

$$s_a \neq s_b \quad \text{and} \quad s_c = f(s_a) = f(s_b);$$

but then

$$u(s_c) = s_a \neq s_b = u(s_c).$$

Therefore, knowledge of the current state and the edit command that produced it is not sufficient to determine the prior state uniquely, because in general edit commands do not have unique inverses.

To define the result of an undo it is necessary to extend the states of the system by adding to the state of the document enough information about the commands that have been applied to allow them to be inverted. There are many ways to do this; we shall do it formally in Section 3 by creating a "history". Editing commands will be one-to-one when extended to histories, and undo will be well defined. For the present, we can consider any set E of extended states with a mapping $c: E \rightarrow S$ that gives the associated state of the document when applied to any extended state. The analysis we give here will apply to "histories" or any other way of constructing E , as long as each editing command k can be extended to E in such a way that if $k: E \rightarrow E$ is the extension of k , then

$$c(ke) = kc(e) \tag{2.2}$$

and k is one-to-one. We shall denote by K the set of all such extended editing commands.

We will use bold letters to denote user-issuable commands on the extended space. We shall denote by \mathbf{O} the set of all user-issuable commands on the extended space, and \mathbf{K} is a subset of \mathbf{O} . We assume that \mathbf{K} contains at least two different editing commands.

We now define the undo command $\mathbf{u}: E \rightarrow E$. Given an extended state e , we might propose to define the undo command \mathbf{u} on E by

$$\mathbf{u}k(e) = e, \text{ for any } k \text{ in } \mathbf{K}. \quad (2.3)$$

Such an undo returns the user to the state prior to the last command k , as if k had never been issued. However, this may be unnecessarily restrictive. What we care most about is the state of the document, not the entire extended state, so we can replace (2.3) by

$$\text{Basic Undo Property: } c(\mathbf{u}ke) = c(e) \text{ for any } k \text{ in } \mathbf{K}. \quad (2.4)$$

We will reject any definition of undo that does not satisfy (2.4), since (2.4) states the minimal requirement that undo remove the effects of editing commands on the document state.

2.2 Characteristics Desired of Undo.

There is a set of expectations usually associated with the concept of Undo. We define properties of undo to meet these expectations and find that this results in incompatibilities. We then classify the types of undo commands based on the set of properties they possess.

The Basic Undo Property (2.4) may not stipulate everything we desire of \mathbf{u} . Two extensions seem desirable.

First, consider how well or poorly uke mimics the original state e . By (2.4) we know that the document state is restored, and by (2.2) we have:

$$\begin{aligned}c(\mathbf{g}uke) &= gc(uke) \\ &= gc(e) \\ &= c(\mathbf{g}e).\end{aligned}$$

Therefore, editing commands applied to uke behave properly. But (2.4) does not specify the effect of undo on the restored state uke . If uke is a complete mimic of e , then for any $n \geq 1$ we should have

Thoroughness:

$$c(\mathbf{o}_n \dots \mathbf{o}_1 uke) = c(\mathbf{o}_n \dots \mathbf{o}_1 e) \text{ for any } \mathbf{o}_i \text{ defined on } E, \quad (2.5)$$

including \mathbf{u} .

Second, we may wish to provide that \mathbf{u} itself, like the editing commands, be invertible, in case we apply it in error. We can require that there exist an operator \mathbf{r} for "Redo", which might be \mathbf{u} itself, which inverts it.

Invertibility: There is an \mathbf{r} such that

$$c(\mathbf{r}ue) = c(e) \text{ for all } e \text{ in } E. \quad (2.6)$$

The case of Invertibility in which \mathbf{u} acts as its own inverse is of special interest:

$$\text{Self-applicability: } c(\mathbf{u}ue) = c(e), \text{ for all } e \text{ in } E. \quad (2.7)$$

Without this, the action of \mathbf{u} is restricted: it cannot undo all commands, but only commands other than itself.

If we make the assumption that K contains at least two edit commands f and g such that for some e belonging to E , $c(fe) \neq c(ge)$, then Invertibility and Thoroughness are incompatible: no u can be Thorough and Invertible. To see this, pick a state e and the editing commands f and g for which $fc(e) \neq gc(e)$, and consider $rufuge$, where r is an inverse of u in the sense of (2.6). By (2.6) and (2.4)

$$\begin{aligned} c(rufuge) &= c(fuge) \\ &= fc(uge) \\ &= fc(e). \end{aligned} \tag{2.8a}$$

But if we set $e' = uge$, then by Thoroughness (2.5)

$$\begin{aligned} c(rufuge) &= c(rufe') = c(re') = c(ruge) \\ &= c(ge) \text{ by (2.6)} \\ &= gc(e). \end{aligned} \tag{2.8b}$$

Note that the contradiction holds whether u is its own inverse or whether a separate 'redo' command, as in COPE [2] and POLITE [14], is provided: an Invertible undo cannot be Thorough.

A weakened form of Thoroughness is "Unstacking": if we wish to be able to recover from the effects of a series of editing commands, not just one, we can require

Unstacking:

$$c(u^n f_n \dots f_1 e) = c(e) \text{ for any } n \text{ and } f_i \text{ in } K. \tag{2.9}$$

We show that Thoroughness implies Unstacking by setting $e' = f_{n-1} \dots f_1 e$ and using (2.5) to arrive at

$$\begin{aligned}
c(u^n f_n \dots f_1 e) &= c(u^{n-1} u f_n e') = c(u^{n-1} e') \\
&= c(u^{n-1} f_{n-1} \dots f_1 e) = \dots = c(e).
\end{aligned}$$

However, Unstacking is possible without Thoroughness, since (2.5) is a requirement on any operators \circ in the system.

As it happens, an undo can be Invertible and Unstacking, as we show in Section 3. However, if we assume that there exist f and g belonging to K and e belonging to E such that $gfc(e) \neq c(e)$, then Self-applicability is not compatible with Unstacking. Pick an f, g, e for which $gfc(e) \neq c(e)$, and consider $uugfe$.

By Unstacking (2.9)

$$c(uugfe) = c(e), \tag{2.10a}$$

but by Self-applicability (2.7)

$$c(uugfe) = c(gfe) = gfc(e). \tag{2.10b}$$

Figure 1 shows in Venn-diagram form the relationships among these properties of undo.

The conflicting results of Invertibility and Thoroughness as seen in equations (2.8a) and (2.8b) and the conflicting results of Unstacking and Self-applicability as seen in equations (2.10a) and (2.10b) are the results of inconsistent, yet equally reasonable, expectations of undo.

In the first conflict, one would like to reverse the effect of erroneously issuing an undo, but then the restored state cannot behave the same with respect to all commands as when first created. The second conflict shows

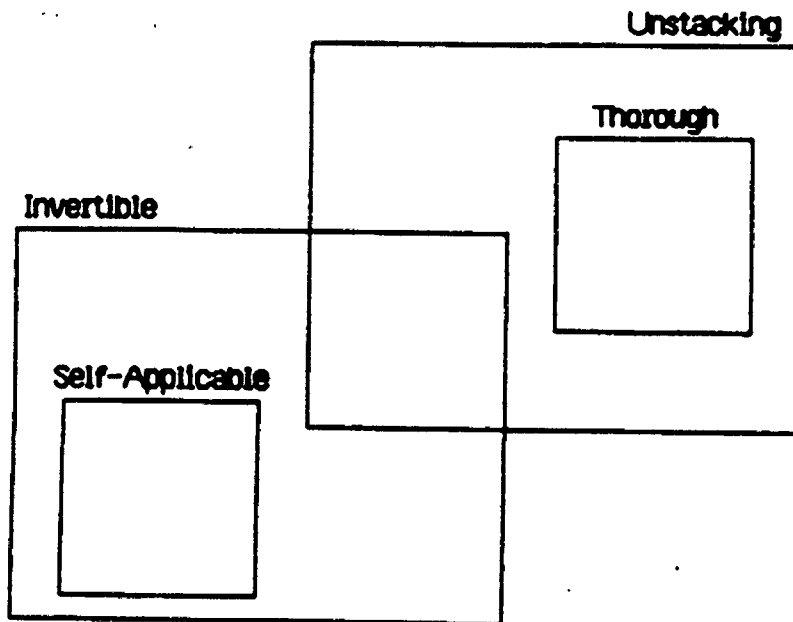


Fig. 1 Properties of Undo

the other limitation of the undo command: it cannot both undo a series of edit commands (unstack) and undo itself (self-applicable).

The classification of undo based on the result of $c(uue)$ (Self-applicability or Unstacking) was the basis for the work in [10]. However, we will examine its more general implications in the next subsection.

2.3 Recoverability.

In this section we define the concept of Recoverability and examine the undo command's capability to recover to a prior state.

Recoverability: The set O satisfies the Recoverability property if for any two extended states e' and e in E , such that $e' = o_m \dots o_1 e$ for some commands o_1, \dots, o_m in O , there exist (not necessarily distinct) commands w_1, \dots, w_n belonging to $O-K$, such that

$$c(w_n \dots w_1 e') = c(e). \quad (2.11)$$

We exclude editing commands in (2.11) to eliminate the process of reconstructing the document in order to recover to a prior state.

One may desire that a single undo command provide recoverability; that is, we restate Recoverability (2.11) for the case when the set $O-K$ consists of only the undo command u . Given the two extended states e' and e of (2.11), there is an $n \geq 0$ such that

$$c(u^n e') = c(e). \quad (2.12)$$

Within the context of (2.12) we examine Recoverability for Self-applicable undo commands and for Unstacking undo commands.

Whereas the definition of Self-applicable undo (2.7) determines the result of $c(u^n e)$ for all n , Unstacking (2.9) does not specify the result of applying u^n to a state which has less than n edit commands to unstack. To specify the result of an Unstacking undo for any n , we define an edit-less state e and introduce the Root Property. An edit-less state e is an extended state for which there is no k belonging to K such that $e = ke'$ for an e' in E . We define the Root Property.

Root Property:

$$ue = e, \text{ if } e \text{ is an edit-less state in } E. \quad (2.13)$$

Unstacking (2.9) and (2.13) determine the value of $c(u^n e)$ for all n and e in E , unstacking the edit commands for n less than or equal to the number m of edit commands composing e and producing the same edit-less state for n greater than m .

We assume that there exist commands f and g in K and state e in E such that $c(gfe) \neq c(e)$. In the case of Self-applicable undo commands we assume that the command f and the state e selected also satisfy the condition $c(fe) \neq c(e)$. In the case of Unstacking undo commands we assume that the commands f and g selected also satisfy the condition $c(gfe) \neq c(fe)$ for the edit-less state e .

First, we show that a Self-applicable undo cannot provide Recoverability. By (2.4) and (2.2)

$$c(ugfe) = c(fe) = fc(e), \quad (2.14)$$

and as shown above, (2.7) implies (2.10b):

$$c(uugfe) = c(gfe) = gfc(e).$$

Choose an extended state e and commands f and g for which $gfc(e) \neq c(e)$ and $fc(e) \neq c(e)$, and let $e' = gfc$ in (2.12). For this choice of e' and e , there is no n such that $c(u^n e') = c(e)$, since by (2.14) and (2.7)

$$\begin{aligned} c(u^n e') &= c(u^n gfc) = fc(e), && \text{if } n \text{ is odd;} \\ &= c(gfc) = gfc(e), && \text{if } n \text{ is even.} \end{aligned}$$

Second, if we impose the Root Property (2.13), an Unstacking undo cannot provide Recoverability. Choose a state e'' (to simplify the proof, let e'' be an edit-less state) and commands f and g for which $fc(e'') \neq gfc(e'')$ and $c(e'') \neq gfc(e'')$, and let $e' = ue$, where $e = gfe''$. For this choice of e' and e , there is no n such that $c(u^n e') = c(e)$, since by (2.9) and (2.13)

$$\begin{aligned} c(u^n e') &= c(u^{n+1} gfe'') = c(u^n fe'') = c(fe'') = fc(e''), & \text{if } n = 0; \\ &= c(u^{n-1} e'') = c(e''), & \text{if } n > 0. \end{aligned}$$

Therefore Self-applicable undo commands and Unstacking undo commands satisfying the Root Property cannot provide Recoverability.

We can give an explicit example of an unstacking undo which provides Recoverability but does not satisfy the Root Property. Let N be the set of nonnegative integers and S the set of all document states, including the empty state Ω . We denote by P the set of products in $S \times S \times \dots$, all but finitely many components of which are Ω ; consequently the length function L from P to N given by $L(s_0, s_1, \dots) = \min\{i \text{ in } N \mid s_j = \Omega \text{ for all } j \geq i\}$ is well defined.

To construct our example we let $E = P \times N$, and for (p, r) in E , $p = (s_0, s_1, \dots)$, we define $c: E \rightarrow S$ as $c(p, r) = s_r$ and extend editing commands k to $k: E \rightarrow E$ by setting $k(p, r) = (p', r+1)$, where $p' = (s'_0, s'_1, \dots)$ and

$$\begin{aligned} s'_m &= s_m, & \text{if } 0 \leq m \leq r; \\ &= k(s_r), & \text{if } m = r+1; \\ &= s_{m-1}, & \text{if } m > r+1. \end{aligned}$$

The undo function $u: E \rightarrow E$ is given by

$$\begin{aligned} u(p,r) &= (p,r-1), & \text{if } r > 0; \\ &= (p,L(p)), & \text{if } r = 0. \end{aligned}$$

We prove Recoverability as follows. Let $p'=(t_0,t_1,\dots)$ and $e' = (p',r_1) = o_m \dots o_1 e$ for some $e = (p,r_2)$, where $p=(s_0,s_1,\dots)$. We will show that there exists an n such that $c(u^n e')=c(e)=s_{r_2}$. It is easy to show that the operations o_i do not delete elements from p , and so all elements of p are elements of p' . Therefore pick r_3 with $s_{r_2} = t_{r_3}$.

If $r_1 \geq r_3$, let $n = r_1 - r_3$. Then $u^n(p',r_1) = (p',r_1-r_1+r_3) = (p',r_3)$,

and

$$c(p',r_3) = t_{r_3} = s_{r_2}.$$

Note that for this case if $r_1 = 0$, then $r_3 = 0$ and $n = 0$.

If $r_1 < r_3$, let $n = r_1 + L(p') + 1 - r_3$. Then

$$\begin{aligned} u^n(p',r_1) &= u^{L(p')+1-r_3}(p',0) \\ &= u^{L(p')-r_3}(p',L(p')) \\ &= (p',r_3), \end{aligned}$$

and

$$c(p',r_3) = t_{r_3} = s_{r_2}.$$

Consequently this u provides recoverability, as claimed.

Lastly we introduce the concept of "direct recoverability". Given a sequence of alternating edit and undo commands, $e' = u f_n \dots u f_2 u f_1 e$, by (2.4) the associated state after each undo is the same: $c(e)$. Therefore given the state $c(e)$, it may be desirable to be able to select any of the states

$f_i c(e)$ by one issuance of a command operating on e' . This direct recovery clearly requires a function with a parameter to specify the desired state, and this is required for any type of undo. We will discuss this further in sections 4 and 5.

In section 3, we will examine three types of undo that instantiate all the realizable combinations of the useful properties of undo. We define one undo that is both Invertible and Unstacking, another that is Thorough and a third that is Self-applicable. We will examine the functions needed for Recoverability for these operators in sections 4 and 5.

3. State History, Extended Edit and Undo Commands

In this section, we will formulate a history structure of document states and define edit and undo commands to manipulate that structure. We define three types of undo commands and associate them with two logical views of undo. We will show that one logical view gives rise to undo commands that are Unstacking, and we define one undo command that is both Unstacking and Invertible and a second undo command that is Thorough. The other view gives rise to undo commands that are Self-applicable, and we define a third undo command that is Self-applicable. We use these commands to skirt the undo inconsistencies shown in section 2.

First we will need to define more precisely the commands and their domain.

Definition 1. S is the set of all possible states of a document. K is the set of edit commands $k: S \rightarrow S$ that change the state of a document. We will use the letters f and g to denote specific edit commands in K .

As stated in section 2, to define the result of an undo it is necessary to extend the states of the system by keeping enough information to invert commands. We will keep a history for each document state. That history will consist of the unique sequence of states from the original document state (root) to the given state. We can represent the document history, consisting of all its states' histories, as an arborescence [12] whose vertices correspond to the states of the document. An arborescence is an acyclic, directed graph with the above property of exactly one immediate predecessor for each vertex, where we define the predecessor of the root vertex to be itself. This implies that for each vertex there is a unique path, which we will call a history path, originating at the root and terminating at the vertex. We will use an ordered arborescence for the document history, so that we can number the branches emanating from

any vertex based on when each branch was last reached, from oldest to most recent. We will define the space of document histories and extend the commands k to this space in such a way that the undo command is single-valued.

Definition 2. T is the set of history trees t whose vertices correspond to document states and whose arcs trace the history of the document beginning with the initial state at the root. A particular history tree t is the ordered arborescence $\{X, p, z, \Gamma\}$, where X is a collection of vertices, p is a function $p : X \rightarrow X$ that defines the sequencing of the vertices by selecting for each vertex its immediate predecessor (if x is the root, $p(x) = x$), z is the current vertex, and $\Gamma : X \times X \rightarrow N$ (where N is the set of nonnegative integers) orders the arcs to the immediate successor vertices of each vertex by numbering them from oldest to most recent. The numbering is based on when each immediate successor vertex was last designated as the current vertex of the history tree. We define the function $v : T \rightarrow X$ which identifies the current vertex: $v(t) = z$. Each vertex x belonging to X corresponds to some state of the document s belonging to S . We define the function $d : X \rightarrow S$ which identifies the document state associated with vertex x . A history path is an oriented path in t , whose origin is the root of the history tree and terminus is a given vertex.

By the definition of an arborescence, p is a single-valued function, and we have defined d to be single-valued. T is the history space corresponding to the set of extended states E in section 2, and $dv(t)$ maps the history tree to the current document state corresponding to $c(e)$ in section 2.

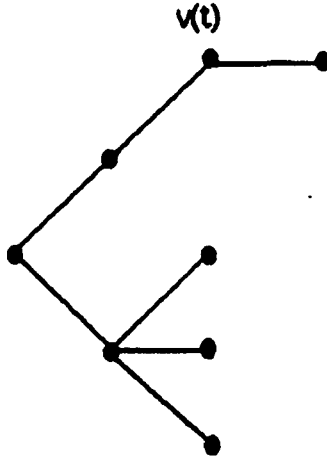


Fig 2 Tree t

Figure 2 is an example of a history tree. The ordering of the arcs at a vertex is from bottom to top, with the most recently-accessed arc at the top.

We shall use the following definitions of equivalence and equality of history trees.

Definition 3. Two history trees are equivalent if they have the same vertices, arcs, and ordering. Two history trees are equal if they are equivalent and have the same current vertex.

Definition 4. Given a history tree t with current vertex $v(t)$ and s in S , we define an append function a from $T \times S$ to T . The append function, $a(t,s) = t'$, forms tree t' by creating a vertex x corresponding to document state s (i.e., $d(x) = s$) and attaching x to $v(t)$, making this arc the highest order arc emanating from $v(t)$. In the resulting history tree

$t' = \{X \cup \{x\}, p, x, \Gamma\}$, p is extended to x , such that $p(x) = v(t)$, and we have $v(t') = x$.

In summary, if $x = va(t,s)$, then

$$d(x) = dva(t,s) = s,$$

$$p(x) = pva(t,s) = v(t).$$

Figures 3a and 3b display history trees t and t' .

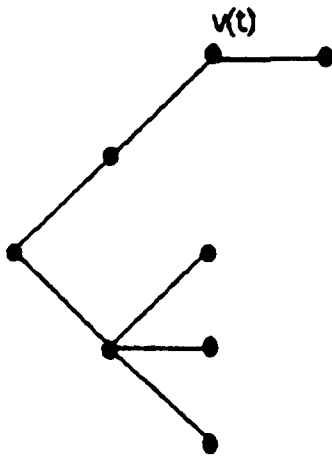


Fig. 3a Tree t

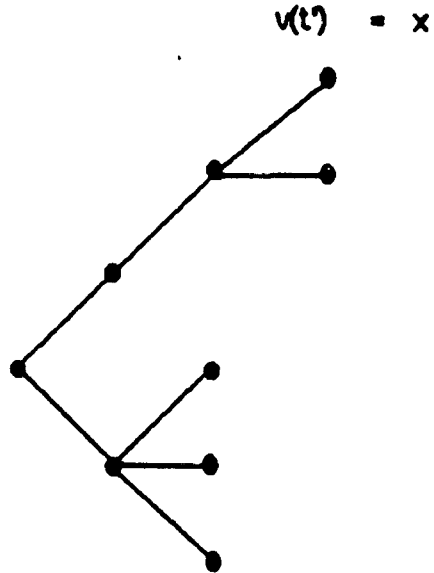


Fig. 3b Tree $t' = a(t,s)$

Note that the append function creates a new vertex corresponding to a given state s and attaches it to $v(t)$, even if there already exists a vertex in t corresponding to s .

We will show that the append function is one-to-one, since it will be the basis for the extended edit commands for which we want to have unique inverses. In this way we solve the problem discussed in Section 2 after (2.1). If $a(t,s) = a(t',s')$, we will show that $t = t'$ and $s = s'$. Given that $a(t,s) = a(t',s')$, their current vertices must be equal, $va(t,s) = va(t',s')$. Therefore, $s = dva(t,s) = dva(t',s') = s'$. The vertices, arcs and ordering of t are the same as those of $a(t,s)$ except that t does not contain $va(t,s)$. Similarly, the vertices, arcs and ordering of t' are the same as those of $a(t',s')$ except that t' does not contain the vertex $va(t',s') = va(t,s)$. Therefore, t is equivalent to t' . Tree $t = t'$, since $v(t) = pva(t,s) = pva(t',s') = v(t')$.

We start with a set of edit commands k , which operate on the current state s of the document to create a new state ks . We extend commands k to T by the following definition. As in section 2, we shall use bold letters for all user-issuable commands on the history space.

Definition 5. For each edit command k , we associate an extended command k with domain and range T . K is the set of extended edit commands k , such that for all k belonging to K ,

$$k(t) = a(t, kdv(t)).$$

The history tree $t' = k(t)$ has current vertex $v(t')$ with document state $dv(t') = kdv(t)$. Note, that corresponding to equation (2.2),

$$dvk(t) = dv(t') = kdv(t).$$

The extended edit commands belonging to K operate on the current state by adding a new state to the history and have no effect on other existing states. See figures 4a and 4b.

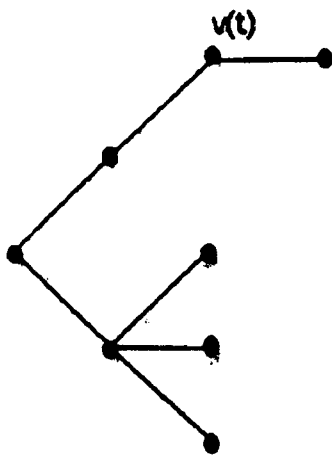


Fig 4a Tree t

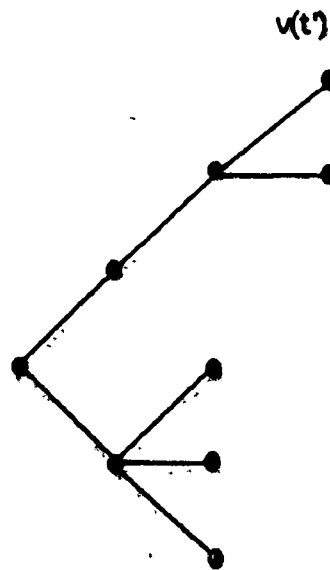


Fig 4b Tree $t' = K(t)$

The commands in K are one-to-one, since $k(t) = k(t')$ implies that $t = t'$: by Definition 5, $k(t) = a(t, kdv(t))$ and $k(t') = a(t', kdv(t'))$, and since the append function is one-to-one, $t = t'$.

We can now define undo commands on T . We will define the Travel Undo u_t , the Recall Undo u_r , and the Retract Undo u_R with domain and range T based on two logical views of undo.

One logical view is that undo moves the user back to the previous state in history.

Definition 6. The Travel Undo, u_t , is a command from T to T , such that if t is a history tree with current vertex $v(t)$, then

$$u_t(t) = t',$$

where the history tree t' is equivalent to t , and

$$v(t') = pv(t).$$

Note that vertex $v(t)$ remains a vertex in t' . The trees t and t' have the same vertices, arcs and ordering; only a different vertex is selected as the current vertex. See figures 5a and 5b.

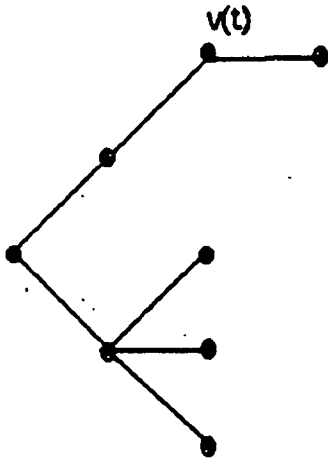


Fig. 5a Tree t

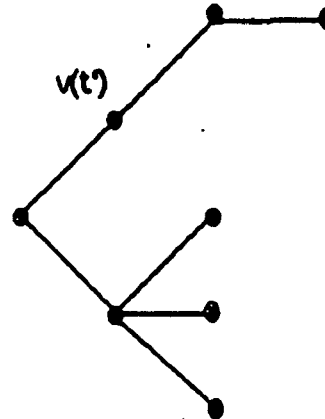


Fig. 5b Tree $t' = u_t(t)$

Since each vertex has a unique predecessor (the root being its own predecessor), u_t is single-valued.

From Definitions 4, 5 and 6, it is seen that for all k belonging to K

$$v(u_t k(t)) = v(t)$$

and

$$dv(u_t k(t)) = dv(t).$$

Therefore the Travel Undo satisfies the Basic Undo Property (2.4).

We show that the Travel Undo satisfies Unstacking (2.9) by applying Definition 6 repeatedly.

$$u_t^n(f_n \dots f_1 t) = t',$$

where t' is equivalent to $f_n \dots f_1 t$, and

$$\begin{aligned} v(t') &= p^n v(f_n \dots f_1 t) \\ &= p^{n-1} v(f_{n-1} \dots f_1 t) \text{ by Definitions 4 and 5} \\ &\cdot \\ &\cdot \\ &= v(t). \end{aligned}$$

Therefore $dv(u_t^n f_n \dots f_1 t) = dv(t)$.

Since $p(x) = x$ when x is the root, if $v(t)$ is the root, then $pv(t) = v(t)$ and thus $u_t(t) = t$. Therefore the Travel Undo satisfies the Root Property (2.13).

The second logical view is that undo expands history by creating a new current state which is a copy of the previous state.

Definition 7. The Recall Undo, u_r , is defined as follows. If t is a history tree, then

$$u_r(t) = a(t, dpv(t)) \equiv t'.$$

That is, the history tree t' has current vertex corresponding to the document state $dpv(t)$.

Since $pv(t)$ is unique and d and a are single-valued, the Recall Undo is single-valued.

From Definitions 4, 5 and 7, it is seen that for all k belonging to K

$$dv(u_r k(t)) = dv(t),$$

and therefore the Recall Undo satisfies the Basic Undo Property (2.4).

The Recall Undo satisfies Self-applicability (2.7), since by Definitions 4, 5 and 7

$$u_r u_r(t) = a(a(t, dpv(t)), dv(t)),$$

and the current document state of the resulting tree is $dv(t)$.

Therefore $dv(u_r u_r(t)) = dv(t)$.

The difference between the Self-applicable undo and the Unstacking undo can be seen with these two views of undo. By the extension of the undo command to the complete history space, as defined in this section, we can see the operation of these two types of undo. When applied once, each undo produces the same current document state, but creates a different history. When applied again to the resulting history, the two undo commands produce different results. In particular, the result of $uugf(t)$ depends on which type of undo is used.

By Definition 5 for extended edit commands **f** and **g**:

$$\mathbf{f}(t) = a(t, \text{fdv}(t))$$

$$\mathbf{gf}(t) = a(a(t, \text{fdv}(t)), \text{gfdv}(t)).$$

For the Travel Undo, by Definition 6:

$u_t u_t \mathbf{gf}(t)$ is a tree equivalent to $\mathbf{gf}(t)$ with current document state $\text{dv}(t)$.

For the Recall Undo, by Definition 7:

$$u_r u_r \mathbf{gf}(t) = a(a(a(a(t, \text{fdv}(t)), \text{gfdv}(t)), \text{fdv}(t)), \text{gfdv}(t))$$

and the current document state of the resulting tree is $\text{gfdv}(t)$.

Therefore the Travel Undo is Unstacking and gives us result (2.10a), and the Recall Undo is Self-applicable and gives us result (2.10b).

There are times when we want the undo to perform as in equation (2.10a) and other times when we want the result of equation (2.10b). When we change the original state by executing the edit command **f** and then **g** and then two undo commands, we expect the result to be the original state. This is because the first undo command should eliminate the effect of its preceding command **g**, and then the second undo command should eliminate the effect of command **f**. For example, suppose that we had made two insertions in a report by executing **f** and then **g**. Then undoing both insertions would get us back to the original report, as in (2.10a).

On the other hand, we might expect the resulting state to be $\text{gfc}(e)$. The execution of the second undo should eliminate the effect of the prior

command, the first undo. For example, suppose that after making the two insertions in the report, we execute undo (eliminating the second insertion) and find we prefer the second insertion in the report. We execute undo again to eliminate the effect of the first undo, as in (2.10b).

The Travel and Recall undo commands can be considered generalizations of the β and ϕ commands of [10], respectively. The undo command in both COPE [2] and POLITE [14] would produce the result $c(e)$ in (2.10a), whereas the execution of the same undo command twice in Bravo [9], PEDIT [6], or P-EDIT [8] would produce the result $gfc(e)$ in (2.10b).

The logical view that led to the definition of the Travel Undo (i.e., putting the user back to a previous vertex in the history) can give us a variation of the Travel Undo command. We can define a command that points back to a previous vertex and deletes all its successor vertices. This command returns us to a prior vertex, such that the resulting history t is the same as when that vertex was first created. This command is of interest, because it satisfies the stronger recovery condition of equation (2.3), $uk(t) = t$, for any k in K .

Definition 8. The Retract Undo u_R is a command with domain and range T such that, if t is a history tree with current vertex $v(t)$, then

$$u_R(t) = t' ,$$

where t' is equivalent to t with vertex $v(t)$ removed, and

$$v(t') = pv(t).$$

By Definitions 4, 5 and 8, it is seen that $u_R k(t) = t$, and therefore the Retract Undo is Thorough.

For the Retract Undo, $u_R u_R g f(t) = t$, giving the same current document state $dv(t)$ as the Travel Undo.

We can set up a hierarchy of the undo commands from strong to weak recovery based on how completely the state is recovered.

For any k belonging to K , the Retract Undo satisfies the following equations:

$$uk(t) = t$$

$$v(uk(t)) = v(t) \tag{3.1}$$

$$dv(uk(t)) = dv(t) \tag{3.2}$$

As shown above, the Travel Undo satisfies only equations (3.1) and (3.2), and the Recall Undo satisfies only equation (3.2) (the Basic Undo Property (2.4)).

With the Travel Undo and the Retract Undo, undo is considered a time machine, providing the capability to move back to a previous state in the history. With the Recall Undo, we interpret undo as perfect memory, providing a copy of a previous state and making it the most recent state in the history. We showed that the Travel and Retract Undo are Unstacking, producing the result in equation (2.10a) and that the Recall Undo is Self-applicable, producing the result in equation (2.10b).

Figure 6 displays the three forms of undo in the Venn diagram of figure 1. In summary, the Retract Undo is Thorough and, as such, is not Invertible. The Travel Undo is Invertible by a redo operator and so is not Thorough and not Self-applicable. The Recall Undo is Self-applicable and so cannot unstack.

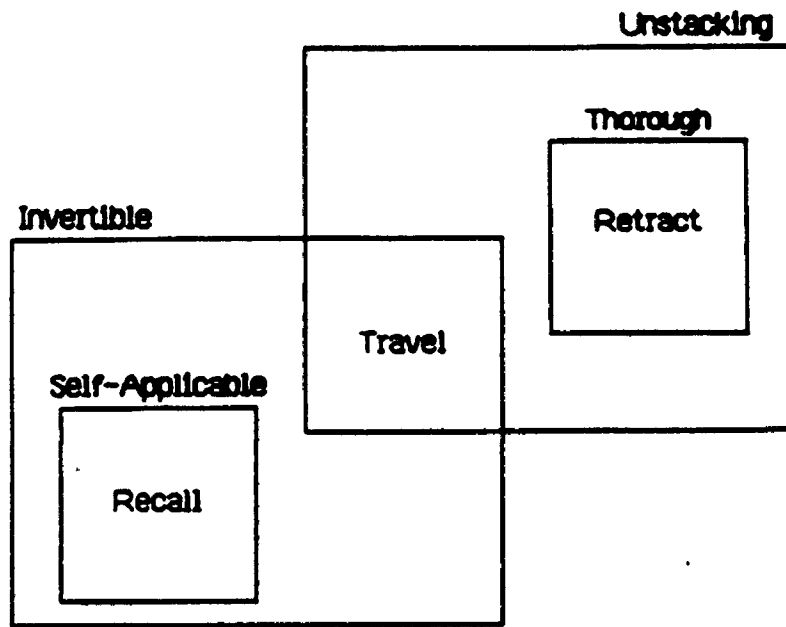


Fig. 6 Types of Undo

We will explore the consequences of the Travel Undo and the Retract Undo in Section 4 and then determine the corresponding results for the Recall Undo in Section 5.

4. The Travel Undo and the Retract Undo

In this section, we examine the characteristics of the Travel Undo and the Retract Undo. First, we look at the result of successive applications of the Travel Undo. This leads us to examine how to recover from an undo and to determine the history structure that results from the use of the Travel Undo. Then we summarize the implications of these results for the Travel Undo and determine the corresponding results for the Retract Undo.

The Travel Undo is not Self-applicable. From the definition of the Travel Undo, we see that applying the Travel Undo i times moves us back i prior states from the current state in history and unstacks i edit commands.

The Travel Undo is a metacommand. By this we mean that unlike edit commands, the Travel Undo command is not "undone" by the next execution of a Travel Undo and does not add a vertex to the history tree.

Because repeatedly executing the Travel Undo moves the value of the current vertex function v back in history, the Travel Undo provides no capability to move the current vertex forward again, that is to recover from an undo. Therefore, we need to define a second command, which we will call redo. We would like the redo and the undo commands to allow us to recover to any prior state.

First we define $nv(t)$ to be the next (successor) vertex to $v(t)$ on the highest ordered arc emanating from $v(t)$, and if $v(t)$ is a terminal vertex, we define $nv(t) = v(t)$.

Definition 9. If t is a history tree with current vertex $v(t)$, then the

$$r(t) = t'$$

where t' is equivalent to t and $v(t') = nv(t)$.

With the Travel Undo, the redo command does not change the history tree structure; it changes only the value of the current vertex. The redo command r is the inverse of u_t :

$$ru_t(t) = t.$$

However, we still do not have the capability to recover to any prior state with the redo and undo commands. We cannot recover to a state once we change the state with an undo command followed by another edit command.

For example, suppose we start with an original report, document state t , and insert text A with command f , undo to the original report, insert text B with command g , and undo to the original. Redo then eliminates the last undo, and we have text B inserted again in the report, $v(t)$.

That is, by Definitions 5, 6, and 9, the document state of the current vertex of tree $ru_tgu_t f(t)$ is $gdv(t)$, since

$$f(t) = a(t, fdv(t)) \quad \text{with current state } fdv(t).$$

$$u_t f(t) = t' \quad \text{where } v(t') = v(t) \quad \text{with state } dv(t).$$

$$gu_t f(t) = a(t', gdv(t)) \quad \text{with current state } gdv(t).$$

$$u_t gu_t f(t) = t'' \quad \text{where } v(t'') = v(t) \quad \text{with state } dv(t).$$

$$ru_t gu_t f(t) = t''' \quad \text{with current state } gdv(t).$$

However, we cannot recover the state $fdv(t)$, although there are situations when we want this to be the result. Consider the following scenario for $ru_t g u_t f(t)$. We insert text A with command f , undo to the original report, insert text B with command g and decide that we prefer text A. We would like to issue an undo command to eliminate text B and then a redo command to get back text A. However, we only get back text B.

A vertex can have several immediate successor vertices, as the vertex $v(t)$ in the above example has successor vertices corresponding to states $fdv(t)$ and $gdv(t)$. Therefore, the redo command must have a parameter to select the desired vertex, and thus, we require the form r_i . We shall examine the structure of the history that results from the Travel Undo and define r_i formally.

The need for a parameter for the redo command is a result of the fact that with the Travel Undo the states correspond to the nodes of a tree with possibly multiple branches emanating from a node, and we need to specify which branch to redo. The edit commands create successor vertices forming a branch, with each vertex having only one immediate predecessor. The tree structure arises, because the Travel Undo points back to a previous vertex, making it the current vertex. Any further editing from that point causes an additional branch to be created from that node. All the vertices are edit-created and all the arcs are the edit commands in the resulting arborescence.

The Travel Undo allows the user to move to the predecessor of the current vertex. The redo command r_i allows the user to move to the successor vertex on the i^{th} branch from the current vertex. The two commands u_t and r_i thus provide the means to traverse all vertices of the tree.

Definition 9A. If the outdegree of the vertex $v(t)$ is $m \geq 1$, then we

sequence based on the order that the branches were last reached, e.g. branch m was visited most recently. We identify the successor vertex of $v(t)$ on branch i by $n_i v(t)$. If $v(t)$ is a terminal vertex (outdegree = 0), then $n_0 v(t) = v(t)$.

We define the redo command r_i from T to T , such that

$$r_i(t) = t'$$

where t' is equivalent to t except for branch ordering from $v(t)$, and $v(t') = n_i v(t)$ making this branch the highest ordered.

In general then r_i will change the ordering of the arcs emanating from $v(t)$. The arc selected by redo will be given the highest order, and the others will be resorted.

The redo command r of Definition 9 is equal to r_m , where m is the out-degree of the current vertex; r selects the successor vertex on the most recent branch to redo. Therefore, this is the only parameter value for redo that does not reorder the arcs.

The recovered state behaves the same with respect to edit commands and the undo command as when originally created by an edit, because its vertex in the tree structure is not changed by undo or redo. It therefore has the same predecessor vertex as when first created. However, more information (successor vertices) is available after undo. Thus redo can select the successor vertices which were not known when the state was first created. Because redo may reorder the arcs, the redo command will in general give different results when applied to the same recovered state at different times in the process.

By viewing prior states, we mean the ability to see a prior state without disturbing history, without any record that the state was visited. This gives the user a chance to examine a past state before deciding whether to undo to the state and change the history. Issuing undo and redo commands without edit commands in between allows movement back and forth through the vertices of the tree without altering the tree structure, except for branch ordering. Thus, this viewing of prior states can be accomplished without impact on future edit and undo commands, but it will impact future redo commands. We will examine facilities in section 6 to move through the history tree without any change to the history.

Examining the characteristics of the Retract Undo, we note that successive issuances of this undo give the same current vertex as the Travel Undo. Therefore the Retract Undo is Unstacking. For the Retract Undo, $u_R u_R g f(t) = t$, and for the Travel Undo, $u_t u_t g f(t) = t'$, with $v(t') = v(t)$.

There is no meaning to a redo function with this undo, since the successor vertices have been deleted. Therefore the Retract Undo is not Recoverable. The history tree is identical to the history tree when the recovered state was first created. There is no knowledge of successor states. The effect of future edit and undo commands on the history tree is therefore the same as when originally created. By Definitions 5 and 8, it is seen that the Retract Undo is the inverse function for all the extended commands in K , i.e., the Retract Undo satisfies equation (2.3) and is Thorough.

The Retract Undo is a metacommand. It is not "undone" by the issuance of the next Retract Undo, as an edit command is, and in terms of the history structure, it does not add any vertices as does an edit command.

If only the Retract Undo is used, the history tree consists of one branch, because the original successor branch is eliminated by this undo and the edit commands just add successor vertices along one branch.

The Retract Undo cannot be used for just viewing, because it changes the history tree when it is executed.

5. The Recall Undo

This section follows the same ordering of results and consequences for the Recall Undo as Section 4 followed for the Travel Undo. We examine the result of successive Recall Undo commands. This leads us to investigate how to reach any previous state in the history and to determine the resulting history structure. We then examine the implications of these results on the undo state and on viewing previous states.

The Recall Undo is Self-applicable. The effect of the Recall Undo on itself is the same as its effect on any command in K . From the definition of the Recall Undo,

$$u_r^m(t) = t' ,$$

where

$$\begin{aligned} dv(t') &= dv(t) && \text{for } m \text{ even,} \\ &= dpv(t) && \text{for } m \text{ odd.} \end{aligned}$$

The Recall Undo causes oscillation between two states, the current and the preceding state, when applied successively.

Based on the above, the Recall Undo requires a parameter to reach vertices beyond the immediate prior vertex.

Definition 10. For $i \geq 1$, the Recall Undo with parameter, $u_{r,i}$ from T to T , selects the document state $dp^i v(t)$ corresponding to the i^{th} prior vertex along the history path from the current vertex $v(t)$ and appends a new vertex corresponding to that document state.

$$u_{r,i}(t) = a(t, dp^i v(t)).$$

Any prior state can be recovered by this parameterized undo. The Recall Undo with parameter corresponds to the undo command in PEDIT [6] and P-EDIT [8].

Since for each selected prior state, a vertex is added to the history as the current vertex by the Recall Undo with parameter, and each edit command adds a vertex to the history, the resulting history tree reduces to one branch. Then the history tree t reduces to the history path h with terminus at the current vertex. The history path h can be viewed as a vector, whose components are in chronological order from the root h_0 (corresponding to the initial state of the document) to the current vertex, say h_m . The chronological order is the order that the corresponding states are seen by the user, both when first created and again if selected by an undo command. A state selected by an undo command then corresponds to more than one component of h . The function $p^i v(t)$ is equal to h_{m-i} .

The tree structure that results from the Travel Undo and the single branch that results from its variation, the Retract Undo, consist solely of edit-created vertices, in contrast to this history structure which is comprised of edit and undo vertices.

We will determine the effect of successive issuances of the Recall Undo with parameter. Assume h has components h_n , for $n = 0, \dots, m$, then with $t \equiv h$,

$$u_{r,i}(t) = a(t, d(h_{m-i}))$$

and

$$u_{r,j} u_{r,i}(t) = a(a(t, d(h_{m-i})), d(h_{m+1-j})) \quad \text{for } 1 \leq i \leq m, 1 \leq j \leq m+1.$$

The resulting document state $d(h_{m+1-j})$ of the current vertex (last component) of $u_{r,j}u_{r,i}(t)$ is the document state of the last component of $u_{r,j-1}(t)$:

$$u_{r,j-1}(t) = a(t, d(h_{m+1-j}));$$

the results are analogous to those for the ϕ operator in [10, p. 22]. Note that $u_{r,1}$ will recover from $u_{r,j}$, for any allowable j .

The resulting state behaves the same with respect to edit commands as when originally created, but behaves differently with respect to undo. This is because the selected state is added as the last component of the history path. Therefore, the state's corresponding vertex has a different predecessor than the original state's vertex.

Viewing, without altering history, cannot be accomplished with the Recall Undo with parameter, because executing this undo changes the history tree by adding a vertex to the history path h . Viewing then would require the introduction of another command.

Variations of the Recall Undo with parameter can be defined based on the number of vertices that are added to the history path with one execution of undo. These variations follow the second interpretation of undo, that is; that undo expands history by creating a new vertex which corresponds to a prior state. The Recall Undo with parameter creates one new vertex, corresponding to the selected prior state. We could define an undo command that creates a sequence of vertices corresponding to all states back from the current state to the selected state, adding them sequentially (reverse sequence to original creation) to the end of the history path. This command may be useful, because the Recall Undo with parameter causes abrupt changes in states along the history, which may make it confusing for the user to search back over the history, whereas this undo provides

a continuous stream of state changes. However, this undo does cause an increasing repetition of state sequences along the history path. To recover to the state that existed prior to the execution of this undo, say with parameter equal to j , we would issue this undo again with the same parameter value j .

6. User Interface Implications

In section 6.1, we summarize the results pertaining to the minimum requirements for the Travel Undo and the Recall Undo to recover to any prior state of the document. In section 6.2, we state the difficulties that a user may encounter with the undo command, and in section 6.3, we offer suggestions to add to the minimum requirements to enhance the use of undo.

6.1 Minimum Requirements for Recoverability.

In section 2 we showed that Unstacking undo commands satisfying the Root Property (2.13) and Self-applicable undo commands cannot provide Recoverability. That is, more than one undo button is necessary to provide Recoverability for each of the above types of undo. We observed that in addition any undo command requires a parameter to provide direct recovery. In sections 4 and 5 we showed that the Travel Undo and the Recall Undo must have the following functions to provide direct recovery:

1. For the Travel Undo

a) Undo command: u_t

b) Redo command with parameter: r_i

2. For the Recall Undo

a) Undo command with parameter: $u_{r,i}$

6.2 Problems.

The ability to recover to any prior state may give the user complete freedom to experiment without worry about backup, but this is only true if it is easy to identify the desired recovery state and to execute the appropriate undo command. However, the user may encounter difficulty with the basic undo facility in determining which state to recover, where the corresponding vertex is in the history, and how to reach it. The more the undo command is used, the more complicated will be the history.

The fact that the underlying history space of the Travel Undo is a tree structure can create problems for the user in determining the path through the branches to the selected prior state's vertex and the sequence of undo and redo commands to traverse this path. Furthermore, the redo command changes the branch ordering. Depending on the user and the particular editing session, at times the relational order of edit-created states in the tree structure is the appropriate model for the user; at other times, a chronological state order may be preferable.

The Recall Undo changes the history each time it is issued. Each time the user executes this undo command, whether for changing a state or just viewing, he needs to supply a different parameter (previous parameter plus one) to reach the same past state. Furthermore, the Recall Undo creates duplicate states in the history, increasing its size and complicating its order. It may be confusing at times that the history consists of both edit-created and undo-created states, and that the states are in chronological order based on when they were first created and when they were reproduced by undo.

Therefore, there is a need to add to the minimum requirements an enhanced capability to identify, view, and then select the desired prior state.

to show the user graphically and textually the result of each redo option. At a branch point, the user is shown the alternative commands that he can redo and the resulting states, beginning with the most recent branch and ending when the user agrees to the displayed selection. Vitter suggests also alternative interfaces including display windows, as the road map above, that show the data structure tree to assist the user in selecting the proper branch.

Another suggestion is to label the arcs of the history tree, rather than the vertices. The arcs would be labeled with their corresponding commands, and these commands could be displayed on the road map or the chronological list. This list of commands corresponds to the log display in the user interface of COPE [2].

6.3.2 History Simplification.

There may be portions of the history that the user will never want to use again, and therefore we should provide a mechanism to delete these vertices, in order to simplify the history search. The use of the Retract Undo removes all successor vertices after the selected vertex, and the user may want to issue it when he knows that his recent editing path is not improving the document. More generally, in the case of the Travel Undo, we would want to give the user the capability to delete nodes or branches that no longer have value to him. Similarly, for the Recall Undo, the user could be provided with a facility to remove components that are erroneous or no longer needed or to delete some repeating sequences and duplicate states.

6.3.3 Viewing.

Recall Undo command without first viewing. Secondly, he could view the desired vertex with the view command and be provided with a command to undo to that vertex (view-key/undo). Thirdly, he could be provided with a command (undo-key), similar to the view-key command, to undo to the vertex given by the identifier key.

In contrast to the view command which only selects the vertex to view, these three options execute the undo command, changing the value of the current vertex to the selected vertex. The latter two options recover to a specified vertex and are not dependent on the current vertex. They can be implemented independently of the history structure determined by each logical view of undo, or by a subroutine for each type of undo. The subroutine would determine the appropriate power for the Travel Undo and parameter(s) for the redo command(s) to obtain the selected vertex in the history tree or the appropriate parameter for the Recall Undo to obtain the selected component of the history vector.

REFERENCES

1. Apple Computer, Inc. LisaWrite. Cupertino, Ca (1983).
2. Archer, J.E. Jr., Conway, R., and Schneider, F.B. User Recovery and Reversal in Interactive Systems. ACM Transactions on Programming Languages and Systems, 6, 1 (Jan. 1984), 1-19.
3. Good, M. An Ease of Use Evaluation of an Integrated Editor and Formatter. MIT/LCS TR 266, Massachusetts Institute of Technology (Nov. 1981).
4. Hammer, M., Ilson, R., Anderson, T., Gilbert, E., Good, M., Niamir, B., Rosenstein, L., and Schoichet, S. The Implementation of Etude, An Integrated and Interactive Document Production System. ACM SIGPLAN Notices, 16, 6 (June 1981), 137-146.
5. IBM Corporation Personal Editor (User Manual 6936761). (Nov. 1982).
6. Kosinski, P.R. PEDIT, A Programmable Editor. IBM (June 18, 1980).
7. Kosinski, P.R. personal communication.
8. Kruskal, V.J. P-EDIT, A Text Editor for Parametric Files, User Guide. IBM, RC8352 (Jul. 10, 1980).
9. Lampson, B.W. Bravo Manual. Xerox Palo Alto Research Center (1979).
10. Leeman, G.B. A Formal Approach to Undo Operations in Programming Languages. RC 10310, IBM Thomas J. Watson Research Center (Jan. 1984).
11. Medina-Mora, R., and Feiler, P. An Incremental Programming Environment. IEEE Trans on Software Engineering, SE-7 (Sept. 1981), 472-481.
12. Picard, C.F. Graphs and Questionnaires, North-Holland (1980).
13. Pournelle, J. User's Column: Epson QX-10, Zenith Z-29, CP/M-68K, and More. Byte Magazine, 8, 8 (Aug. 1983), 434-454.
14. Prager, J.M. and Borkin, S.A. POLITE Project Progress Report. Report G320-2140, IBM Cambridge Scientific Center (Apr. 1982).
15. Seybold, J.W. Apple's Lisa: The 'Next Generation' Computer? The Seybold Report, 12, 10 (Jan. 31, 1983).
16. Teitelman, W. Interlisp Reference Manual. Fourth Edition, Xerox Palo Alto Research Center (1978).