Nothing relevant.

nothing relevant

Undoing Actions in Collaborative Work

Atul Prakash
Michael J. Knister

Software Systems Research Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122
Phone: (313) 763-1585
Email: aprakash@eecs.umich.edu, mknister@eecs.umich.edu

ABSTRACT

The ability to undo operations is a standard feature in most single-user interactive applications. However, most current collaborative applications that allow several users to work simultaneously on a shared document lack undo capabilities; those which provide undo generally provide only a global undo, in which the last change made by anyone to a document is undone, rather than allowing users to individually reverse their own changes. In this paper, we propose a general framework for undoing actions in collaborative systems. The framework takes into account the possibility of conflicts between different users' actions that may prevent a normal undo. The framework also allows selection of actions to undo based on who performed them, where they occurred, or any other appropriate criterion.

KEYWORDS

Undo, collaboration, groupware, conflict analysis.

1 INTRODUCTION

The ability to undo operations is a standard and useful feature in most interactive single-user applications. For instance, the availability of undo facility in editors is invaluable if users are allowed to invoke commands that can modify a document in complex ways (e.g. remove all lines that contain the string "##"). Availability of undo can also encourage users to experiment, acting not only as a safety net, but also allowing users to try out different approaches to solving problems using backtracking [9]. Unfortunately at present, though many collaborative editors and other group applications have been built, such as GROVE [2], ShrEdit [5], and DistEdit [4], almost all lack undo capabilities. Those which provide undo generally provide only a global undo, in which the last change made by anyone to a document is undone, rather than allowing users to individually reverse their own changes.

Compared to single-user applications, performing undo in collaborative applications provides technical challenges in three areas: choosing the action to be undone, determining where the undo should occur, and resolving conflicts between different users. First, choosing the action to undo in a single-user system is usually easy: simply choose the most recent action and use it to revert to the prior state of the document. However, in a group environment, there may be parallel streams of activities from different users, and the undo needs to be more selective. Often, when parallel work on a shared document is going on, users would expect to undo their own last actions rather than globally-last actions, which might belong to other users. Second, once the correct operation is chosen to be undone, the location at which the undo of an action should be performed may be different from the location at which the action was originally performed due to the effects of other users' activity on the document. Finally, if two or more users interleave their work in the same portion of a document, it may not make sense to undo one user's changes without undoing the other users' changes. In this case, there are dependencies between the changes made that need to be taken into account during an undo.

The rest of the paper is organized as follows. Section 2 summarizes basic concepts used in conventional undo algorithms for single-user systems. Section 3 describes related work in the area. Section 4 discusses how our approach extends undo capabilities, particularly for group environments. Section 5 describes the requirements an application must meet to use our undo framework and an example for text editing. Section 6 describes our selective undo algorithm. Section 7 discusses several variations of selective undo that may be useful in collaborative systems. Finally, Section 8 summarizes our

Appeared in the Proceedings of the Fourth Conference on Computer-Supported Cooperative Work, October 1992, pp. 273-280.

conclusions and future work.

2 BASIC CONCEPTS

To provide the raw ingredients for implementing undo, a *history list* needs to be maintained. The history list consists of the sequence of operations that have been carried out so far to modify the state of the document. The operations on the history list are stored in the same order as they were performed. For instance, if the history list is

$$A\ B\ C\ D$$

then, starting from state prior to A, carrying out the operations A, B, C, and D in sequence should lead to the current state.

In this paper, we assume that all operations that modify the state of the document are *reversible*, i.e., for every operation A, we can determine an inverse operation $\overline{A}$ that will undo the effect of A. For instance, in an editor, an *INSERT* operation can be undone by a *DELETE* operation.

Note that, in general, the inverse of an operation A may depend on state of the document prior to A. For instance, on a text document, if a *DelChar(10)* operation is done, which deletes the character at position 10, then in order to determine its inverse, we must know the character that was deleted. We assume that the operations stored in the history list contain sufficient data so that their inverse can be easily determined. For instance, the above operation might be stored as *DelChar(10, c)* on the history list, where c is the deleted character.

3 RELATED WORK

There are several basic methods for providing undo capability in single-user systems. We discuss them here. A more detailed discussion of these techniques can be found in [9], and a formalization of undo and redo facilities can be found in [11].

3.1 Single-step Undo

Single-step undo is common in most Macintosh and Windows applications, as well as editors such as *vi*. It allows undo of the last operation. For instance, given the history list

$$A\ B\ C\ D\ E$$

single-step undo allows undoing of operation E, but not a subsequent undo of operation D. Usually redo of the last undo is also allowed (often implemented as an undo of the last undo) so that, in the above example, E can be redone.

3.2 Linear Undo Model and US&R Model

The Interlisp system [8], one of the early systems to provide undo, used the linear undo model. The linear undo model allows undoing of a sequence of operations and keeps a pointer which tracks the last operation undone. Operations can then be redone, after possibly doing some new operations. For instance, given the history list:

$$A\ B\ C\ D\ E,$$

operations E and D can be undone (in sequence), then a new operation F done, and then D redone, giving the following history list:

$$\begin{array}{c} A\ B\ C\ F\ D\ E \\ \uparrow \end{array}$$

The pointer indicates that the next operation to be undone is D, and E is the next operation that could be redone. Note that, in this model, undo operations are not explicitly stored in the history list. So, if one wants to back to the original sequence without the F, it is not possible. One could undo F, but then D and E must be done manually.

The Undo, Skip, Redo (US&R) model [10] supports redo like the linear undo model, but also allows a more user-friendly skipping of some operations during the redo. Instead of a linear list, US&R model keeps a tree data structure for maintaining history so that it becomes possible to restore state to any point in the history (unlike the linear undo model). In the above example, F would be stored on a different branch of the tree from the sequence $D\ E$ so that F could be undone and then D and E could be redone if the user so desired.

A limitation of both the linear undo model and the US&R model is that in order to undo one operation O several steps back in the history, all subsequent operations must first be undone and then redone (skipping O during the redo). If the implementation is not careful, this can be potentially disruptive in a group environment; other users may see their work undone for at least a short while with no apparent reason. Furthermore, the models do not address the issue that simply redoing operations may not semantically make sense or may lead to unexpected results if an earlier operation is skipped.

3.3 History Undo

The history undo scheme, used in the *Gnu Emacs* editor [7], also allows undoing of a sequence of operations but, unlike the linear undo and US&R schemes, it appends the undo operations to the end of the history list. The undo operations in the history list are treated as any other operations, allowing them to be undone later if desired. For instance, given the history list,

$$A\ B\ C\ D\ E$$

suppose that E is undone. Then in the history undo, the history list will be as follows, where $\overline{E}$ is the operation that reverses the effects of E:

$$\begin{array}{c} A\ B\ C\ D\ E\ \overline{E} \\ \uparrow \end{array}$$

If one now breaks out of the undo mode by doing some operation other than an undo, say F, the history list will become:

$$A\ B\ C\ D\ E\ \overline{E}\ F$$
$$\uparrow$$

At this point, doing two more undo operations will result in:

$$A\ B\ C\ D\ \underset{\uparrow}{E}\ \overline{E}\ \ F\ \overline{F}\ \overline{\overline{E}}$$

History undo has the nice property that it is possible to go back to any previous state, and the possibility of conflicts does not arise (in single-user applications) since operations are never skipped.

4 OUR APPROACH

In this paper, we present an approach called *selective undo*. The approach is based on history undo, but we allow operations to be undone selectively and deal explicitly with location shifting and conflicts. In our experience, history undo is simple and intuitive for most users. However, if desired, the techniques given in the paper can also be applied to linear and US&R models.

We use data structures similar to those used in history undo; in particular, upon an undo, the inverse of an operation is appended at the end of the history list. However, in a collaborative application, since the last operation done by a user may not be *globally* last (other users may have done operations subsequently), we need to allow undoing of a particular *user's* last operation from the history list. For example, consider the following history list, where A_i's refer to operations done by user A, and B_i's refer to operations done by other users:

$$A_1\ B_1\ A_2\ B_2\ B_3$$

Now, suppose user A wishes to undo his last action, A_2. Normal history undo mechanisms in single-user systems do not support such a task because they would require undoing B_2 and B_3 as well. In the $US\&R$ model, it is possible to undo the last three operations and then redo B_2 and B_3, but as pointed out in the previous section, that can be disconcerting to other users of the system and may not even be correct if there are dependencies between A_2 and B_2, B_3. Note that user A may not be aware that operations B_2 and B_3 have been carried out on the document by other users, and the other users may not aware of activities of user A. In the algorithms presented in this paper, it is possible to undo A_2 without undoing/redoing B_2 and B_3.

In the above example, the operation to be undone, A_2, is selected based on the identity of the user. More generally, the operation to undo could be selected based on any other attribute, such as region, time, or anything else. Thus, we term our scheme as *selective undo*, since the operation to be undone is not necessarily the last one, but is selected using some attribute attached to the operation.

To selectively undo an operation, we cannot simply execute the inverse of the operation because later operations could have shifted the location where the undo must be performed. For example, suppose the following two text operations have been applied to the starting state 'abcd': $InsChar(4,'x')$ followed by $InsChar(1,'y')$, resulting in the state 'yabcxd'. The first operation inserted 'x' at position 4, and the second operation inserted 'y' at position 1. Assume that these operations were done by different users. Now the user who did the first operation does an undo. We cannot simply perform the first operation's inverse, $DelChar(4)$, because the second operation has moved the 'x' to location 5. Our scheme takes this possibility of location shifting into account, so that in this example, the first operation will be undone by executing $DelChar(5)$.

We also take into account the possibility of *conflicts*. In the above example, B_2 may have modified the same region of the document as A_2, so that it no longer makes sense semantically to undo A_2 without first undoing B_2. We do not allow an operation to be undone until any prior conflicting operations have been undone.

5 APPLICATION REQUIREMENTS

Our undo framework assumes an application model in which all changes to a document are performed using a set of primitive operations. As operations are performed, they are archived in a history list to provide the basis for undo. The operations must be reversible and capable of being re-ordered when no conflicts between the operations exist.

All applications maintain a current *state* of the document that is being edited. This state can be represented in different data structures, and our framework places no restrictions on the representation.

Primitive operations, or just *operations*, are the only means by which the state of a document can be altered. An operation applied to a state results in a new state. Any given state is simply the result of a sequence of zero or more operations applied to the starting state. Operations can also have parameters which specify exactly what the operation is to accomplish and where it is to be performed. For instance, a DELETE operation would have parameters to indicate what is to be deleted.

We will use the letter S to denote state prior to application of an operation. A $\circ$ indicates that the operation is being applied. For example,

$$S \circ M \circ N$$

denotes the state resulting from application of operation M followed by operation N on a document in state S. Sometimes, we will also use $A \circ B$ to denote the compound operation that first applies A and then applies B.

Two sequences of operations are *equivalent* if they produce the same state. Equivalence is represented by $\equiv$. For example,

$$M \circ N \equiv P \circ Q$$

indicates that the two sequences produce the same state, even though the operations in each sequence are not identical.

5.1 Extensions to the History List

For implementing undo in collaborative systems, some extensions are needed to the basic history list described in Section 2. In order to selectively undo a particular user's operation, we must tag each operation in the history with the identity of the user who performed it. Other tags could be stored as well, such as the time of or the reason for the operation. Our selective undo algorithm allows any such tag to be used to choose operations to undo.

5.2 Conflict, Re-ordering, and Reversibility of Operations

Our model requires that the application supply functions which can detect conflicts between operations, re-order operations, and create inverse operations. In a synchronous group environment, similar functions would usually be needed anyway to ensure predictable results when parallel streams of activities are going on. For instance, if two users are working simultaneously in a document, conflict checking may involve making sure that their changes do not overlap, e.g., through use of locks. Mechanisms for reordering of parallel, independent operations are also needed because the order in which two operations will be done may be unpredictable. The editor must be prepared to accept the two operations in either order with the same resulting effect.

The functions which the application must provide are:

- $Inverse(Operation) \Rightarrow Operation$
- $Conflict(Operation, Operation) \Rightarrow Boolean$
- $Transpose(Operation, Operation) \Rightarrow (Operation, Operation)$

It is assumed that operations that result from these functions are also primitive operations — or can be expressed in terms of primitive operations (see Section 7.3 for extensions needed for multi-operation undo). This allows the operations that result from applying the above functions to be treated just like other operations in the history list. The *Inverse* function has already been explained in Section 2. The following sections provide descriptions and properties for *Conflict* and *Transpose* functions.

5.2.1 Conflict

A *conflict* between two adjacent operations A and B implies that the second operation, B depends on A and is not meaningful without having performed A.

Suppose, for example, that a graphics document is being edited. Operation A creates a circle in the document, and operation B resizes that circle. In this case, there is a conflict between A and B. If operation A had not been done, operation B would make little sense.

The $Conflict(A, B)$ function supplied by the application must return $True$ if there exists a conflict when the two operations are performed in sequence, and $False$ if no such conflict exists. The importance of the notion of a conflict is that an operation cannot be undone if it conflicts with a later operation, unless the later operation is undone first.

5.2.2 Transpose

If no conflict exists between two operations, we require that it be possible to *transpose* them. That is, by making some adjustments to the operations, it is possible to perform them in a different order and still obtain the same result.

The $Transpose(A, B)$ function, given two non-conflicting operations A and B, will return two new operations B' and A', which satisfy the following two properties:

Transpose Property 1: Performing $S \circ B' \circ A'$ will give the same result as executing $S \circ A \circ B$, irrespective of the initial valid state S.

Transpose Property 2: B' is the operation that would have been done to the document instead of B if operation A had not been done before B.

Property 1 allows us to move operations around in the history list and guarantees that the resulting state will be the same. Property 2 shows that A can meaningfully be undone, leaving only the effects of B. Quite often, but not always, operation A will be identical to A', and B to B', except that the position data may be different.

Our notion of transpose is similar to the one described in [1]. However, we require transpose function to be defined only when the operations do not conflict.

Example 1: Document Model applied to Text Editing

Consider a text editor supporting the following two primitive operations:

- $InsChar(position, char)$ to insert a character at the specified position; and
- $DelChar(position)$ to delete a character at the specified position.

Note that the model does not dictate the actual data structure which is used to store the document state. The current state could be represented as a linked list of lines, as a single array of characters, or any other way. The application is responsible for correctly applying operations so that its internal data structure represents the correct state.

We will denote operations to be stored in the history list as follows:

- $InsChar(position, char)$
- $DelChar(position, char)$

Note that the character deleted is also stored in the history list as part of the $DelChar$ operation so that we can easily derive its inverse. The above two operations happen to be inverses of each other.

Following are the definitions of *Conflict* and *Transpose* for the sequence *InsChar()* followed by *DelChar()*:

$$Conflict(InsChar(p_1, c_1), DelChar(p_2, c_2)) = \begin{cases} true, & \text{if } p_1 = p_2; \\ false, & \text{otherwise.} \end{cases}$$

$$Transpose(InsChar(p_1, c_1), DelChar(p_2, c_2)) = \begin{cases} (DelChar(p_2 - 1, c_2), InsChar(p_1, c_1)) & \text{if } p_1 < p_2; \\ undefined, & \text{if } p_1 = p_2; \\ (DelChar(p_2, c_2), InsChar(p_1 - 1, c_1)) & \text{if } p_1 > p_2 \end{cases}$$

The above definitions say that there is a conflict if the character deleted is the same as the one that was inserted. Otherwise, the two operations are considered to be independent and transposable. Notice the change in position argument in *Transpose()* so that Transpose Properties are satisfied. We leave it to the reader to determine the Conflict and Transpose definitions for the other three combinations of these two operations. A complete definition of these two functions for general string insert and delete operations can be found in [6].

Example 2: Document Model Applied to Graphics Editors

Let's assume that two of the commands that are stored on the history list of a graphical editor are

- *DrawCircle(x,y, radius, CircleID)*: Draw a circle at position (x,y) of the specified radius. *CircleID* is the object identifier returned by the command and stored in the history list to permit easy reversal and transpose.
- *ChangeRadius(CircleID, NewRadius, OldRadius)*: Change the radius of the circle *CircleID* to *NewRadius*. *OldRadius* is stored so that inverse is easy to compute.

In this case, the *Conflict* and *Transpose* functions are straightforward:

- *Conflict:* the two operations will conflict if and only if they refer to the same circle, i.e., their *CircleID*'s match.
- *Transpose:* Transposing the two operations simply requires interchanging the two operations if they refer to different circles; else the *Transpose* is undefined.

Note that the graphical operations, unlike those in text editors, usually will not require parameter changes since graphical operations usually use absolute (x,y) coordinates rather than coordinates that change with the position of other objects; if relative positioning is desired, then different operations should be provided that do use relative coordinates so that they can be correctly transposed.

6 UNDO ALGORITHMS

This section presents two versions of our undo algorithm: a limited selective undo to demonstrate the basic concepts, followed by the full selective undo algorithm. Both algorithms assume that an operation has already been chosen to be undone. Methods of selecting which operation to undo are described in Section 7.

The algorithms are independent of whether a single centralized history list is maintained for all users or every site has its own (possibly different, but equivalent) history list and editor state. For multiple history lists and editor states, the communication protocol between editors should ensure that all editors eventually reach the same state even when operations are being done in parallel at various sites. [6].

6.1 Limited Selective Undo

To demonstrate the principles of our undo technique, we first describe a limited version of the algorithm and present an example.

The algorithm works as follows: the *transpose* function is used to repeatedly shift the operation to be undone until it reaches the end of the history list. If it cannot be shifted to the end due to a conflict along the way, it cannot be undone. If the operation can be shifted to the end, we can simply execute the inverse of the shifted operation to undo it. By shifting the operation, we have effectively determined where the undo must be performed.

An example will help demonstrate the algorithm. Assume that we want to undo A given the history list:

$$A\ B\ C$$

Suppose A conflicts with B. Then $Conflicts(A, B)$ will be true, and the undo will fail, as it should. If A does not conflict with B, the result after one iteration will be:

$$B'\ A'\ C$$

where $(B', A') = Transpose(A, B)$. Note that the the history list need not be actually altered because the only the new A' is used in the next iteration. We show the entire list here for clarity.

Next, if $Conflicts(A', C)$ is true, the undo will fail. Otherwise, another shift will occur, resulting in:

$$B'\ C'\ A''$$

where $(C', A'') = Transpose(A', C)$. Now that A has been shifted to the end of the list, $\overline{A''}$ can be performed giving the history list:

$$A\ B\ C\ \overline{A''}$$

To see that this is correct, note that since $B' \circ C' \circ A''$ is equivalent ($\equiv$) to $A \circ B \circ C$ (by Transpose Property 1), we find:

$$A \circ B \circ C \circ \overline{A''} \equiv B' \circ C' \circ A'' \circ \overline{A''} \equiv B' \circ C'$$

Thus, performing $\overline{A''}$ at the end of the original history gives the same result as if operation A had never been performed (by Transpose Property 2); the undo has succeeded!

This algorithm, while correct, is unable to deal with the results of prior undo operations. For example, suppose that the history contains $A\ B\ C$, where A and B conflict but neither conflicts with C. A user, wanting to undo both A and B, first undoes B, resulting in the history $A\ B\ C\ \overline{B'}$. Then, the user attempts to undo A. The limited undo determines that A conflicts with B, and is unable to shift A to the end of the history. However, since B is undone, we should be able to undo A.

6.2 Selective Undo

We now give a selective undo algorithm which is not limited by prior undo operations (Figure 1). The algorithm is similar to the limited algorithm in Section 6.1, but it uses a more sophisticated conflict-checking technique.

To avoid the prior undo limitation, we must track which operations have already been undone. We do this by placing a pointer into the history list that links an operation to its corresponding undo operation. Thus, upon undoing B from the sequence $A\ B\ C$, the history list would appear as follows; note that the oval line beneath the sequence indicates a *do-undo* pointer:

$$A\ \underbrace{B\ C\ \overline{B'}}$$

The undo algorithm works by making a copy of the end of the history list, from the operation to undo onward. The operation to undo is shifted using *transpose* until it reaches the end of the list. Before each shift, we check whether a conflict exists with the following operation. If a conflict is found with an operation which has been later undone (i.e. there is really no conflict), that operation and its undo are removed from the copied history list by procedure *RemoveDoUndoPair*.

The *RemoveDoUndoPair* subroutine, given an operation X which is later undone by $\overline{X}$, shifts X until it is adjacent to $\overline{X}$, and then removes both operations. This is valid because $X \circ \overline{X}$ is an identity operator. X will not conflict with another operation Y in the history between it and $\overline{X}$, unless Y itself has been undone (otherwise, X could not have been undone). In the case of such an intervening Y, *RemoveDoUndoPair* is called recursively to first eliminate Y from the history list.

6.2.1 An Example of Selective Undo

Let us say that the history list at some point is as follows:

$$A\ B\ C\ D$$

Assume that operations B and C conflict, and there are no other conflicts. If the operation C is undone, the history list will be a follows, where C' is the operation that results from shifting C past D:

$$A\ \ B\ \ \underbrace{C\ \ D\ \ \overline{C'}}$$

Now, suppose operation B is to be undone. The algorithm will first copy *HistoryList* from B onwards into *TempHistoryList* so that the original list is not affected by shifting operations. Since there is a conflict between B and C, and C has a *do-undo* pointer, *RemoveDoUndoPair()* will be called to remove the C and $\overline{C'}$ pair. The resulting (temporary) history list from B onwards will be as follows:

$$B\ D'$$

where $(D', C') = Transpose(C, D)$.

Assuming that there is no conflict between B and D', B will be shifted past D' giving the operation B' where $(D'', B') = Transpose(B, D')$. Now that operation B has been shifted to the end of the list, it can be successfully undone using the operation $\overline{B'}$. This operation is carried out and appended to the original history list, with the appropriate *do-undo* pointers added, giving the desired result:

$$A\ \underbrace{B\ \ \underbrace{C\ \ D\ \ \overline{C'}}\ \ \overline{B'}}$$

7 VARIATIONS OF SELECTIVE UNDO

Before undo algorithms given above can be used, a means must be provided for a user to select the operation he wishes to undo. There are many variations

```
type HistoryRec = record
    op: Operation;
    next: ^HistoryRec;
    /* Following field is for pairing do/undo */
    undoneBy: ^HistoryRec; end

proc Undo(UndoItem: ^HistoryRec)
    HistTemp: ^HistoryRec;        /* temporary list */
    PrevPtr, HistPtr: ^HistoryRec;/* node pointers */
    ShiftOp: Operation;
    NewItem: ^HistoryRec;

    /* Make a copy of the history list
       from the UndoItem onward */
    HistTemp := CopyTailofList(UndoItem);
    /* Shift UndoItem forward, removing
       all paired do/undo operations */
    ShiftOp := HistTemp^.op;
    PrevPtr := HistTemp; HistPtr := HistTemp^.next;
    while HistPtr <> nil do
        if Conflicts(ShiftOp,HistPtr^.op) then
            if (HistPtr^.undoneBy <> nil)
                RemoveDoUndoPair(HistPtr);
                HistPtr := PrevPtr^.next;
            else return ("Sorry. Conflicts with", HistPtr);
            endif
        else /* Transpose returns two operations;
                store the 2nd in ShiftOp */
            (_,ShiftOp) := Transpose (ShiftOp,HistPtr^.op)
            PrevPtr := HistPtr; HistPtr := HistPtr^.next;
        endif
    endwhile
    /* Perform executes the operation, appends it to
       the end of the history list, and returns a
       pointer to the appended node */
    NewItem := Perform(Inverse(ShiftOp));
    UndoItem^.undoneBy := NewItem;
    return ("Undo successful");
endproc

proc RemoveDoUndoPair(doPtr: ^HistoryRec)
    while doPtr^.next <> doPtr^.undoneBy do
        if Conflicts(doPtr^.op, doPtr^.next^.op) then
            /* if there is a conflict, it must have been
                undone, so can be removed */
            RemoveDoUndoPair(doPtr^.next)
        else /* Transpose the two operations,
                logically and physically */
            (doPtr^.next^.op, doPtr^.op) =
                Transpose(doPtr^.op, doPtr^.next^.op);
            ListSwap(doPtr, doPtr^.next)
        endif
    endwhile
    /* The operation is now adjacent to its undo;
    remove them both from HistTemp list */
    ListDelete(HistTemp, doPtr^.next);
    ListDelete(HistTemp, doPtr)
endproc
```

Figure 1: Selective Undo Algorithm

by which operations to be undone can be selected. Determining the most appropriate variation is a subject of future research and is not the focus of this paper. However, we give some of the interesting variations to illustrate the basic techniques.

7.1 Individual History Undo

The Emacs-style history undo described in Section 3.3 can, with minor modifications, be made to work in our framework, allowing each user to undo his most recent operations one by one.

The first time a user does an undo, the system searches backward from the end of the history list until an operation tagged with that user's identity is located; a pointer to that history record is stored for later use by the user. The selective undo algorithm is then applied to the operation. Should the user immediately do another undo, the history search continues backward from the stored pointer. Thus, the user can proceed back through his most recent changes. When an operation other than another undo is performed, the stored pointer is deleted, making the undo operations appear as normal operations which can be undone.

If the undo algorithm fails due to a conflict, a simple Conflict List Generation [6] algorithm can be used to locate the conflicting operations, which must belong to other users. At this point, the interface can inform the user of the problem and show whose work must be undone. He might then be given a choice of canceling or proceeding to undo the operations of those other users.

7.2 Regional Undo

Another useful criterion for selecting undo operations is a region in the document. For example, a user may want to undo his most recent changes to the abstract of a paper, but not any other changes.

Using a region as a selection criterion is slightly more difficult than using user-id or timestamps, because operations performed historically on a region refer to the location where the region used to be, rather than where it is now.

To locate an operation which affects a region R, we start by defining a special region-identifying operation S which we define to conflict with any operation performed in R. We place S at the end of the history list, and use transpose to shift it backward. If it cannot be transposed due to a conflict, that conflicting operation must be within the region, and can now be undone. Note that for any operation A, $Transpose(A, S)$ should give (S', A), where S' is the region that corresponds to S prior to doing A. To implement repeated undo on a region, it is necessary that *Transpose(A,S)* be defined even if A conflicts with S, so that S can be shifted past

A after A is undone for subsequent undos. This apparent anomaly is not a problem since S is not a update operation — it is simply introduced to identify a region and determine which operations were carried out in the associated region.

7.3 Multi-operation actions

Situations often arise in which an application may wish to treat a group of primitive operations as a single, high-level, operation. For instance, consider the following scenarios:

- One user-level action (e.g. *IndentParagraph*) could result in numerous primitive operations (a bunch of *INSERTs*). Users would expect to be able to undo the high-level operation in entirety using one undo operation rather than having to undo the primitive operations one by one.

- Undoing many steps at once could be useful for returning to a known previous state. For example, a user may wish to revert chapter 15 of a paper back to the way it was at 5PM last Tuesday (i.e., undo all operations done on chapter 15's region with timestamps after 5PM last Tuesday), assuming sufficient history with appropriate tags is kept.

Multiple-operation actions and corresponding undo actions are similar to the notion of *transactions* in databases. Either all the primitive operations should be performed collectively, or conflicts should be reported and handled first. For instance, suppose that a paragraph is indented and then modified so that conflicts arise. It would not be desirable to allow a partial undo — its effect is likely to be hard to understand.

Multi-operation undo can be implemented in our framework with the following extensions:

1. The history list needs to be extended to keep sufficient information around so that the set of operations that constitute a high-level operation can be determined.

2. When undoing a high-level operation, all the primitive operations that constitute the high-level operation need to be shifted to the end and then undone as a single transaction. If conflicts arise during shifting, the undo should not be permitted without first undoing the conflicts.

3. Do-undo pointers need to go between corresponding operations, which could be high-level.

8 CONCLUSIONS

We have presented a framework for group undo which is simple and generally applicable to a variety of documents. The techniques proposed in this paper are presently being implemented in the DistEdit toolkit [4]. The techniques are presented in the context of history undo; however, many aspects of the techniques, such as the notions of *Transpose* and *Conflict*, are also applicable to implementing undo based on linear and US&R models. The focus of the paper was on developing a general framework for group undo; research is still needed to determine appropriate interfaces for supporting undo in collaborative applications.

REFERENCES

[1] Ellis, C.A. and Gibbs, S.J. Concurrency Control in Groupware Systems, in *Proceedings of the ACM SIGMOD '89 Conference on the Management of Data* (Seattle, Washington, May 1989), ACM Press, pp. 399-407.

[2] Ellis, C.A., Gibbs, S.J., and Rein, G.L. Design and Use of a Group Editor. In *Engineering for Human-Computer Interaction*, G. Cockton, Ed., North-Holland, Amsterdam, 1990, pp. 13-25.

[3] Ellis, C.A., Gibbs, S.J., and Rein, G.L. Groupware: Some Issues and Experiences. *Communications of the ACM*(January 1991), 38-58.

[4] Knister, M. and Prakash, A. DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors, in *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, Los Angeles, California, October 1990, pp. 343-355.

[5] McGuffin, L. and Olson, G.M. ShrEdit: A Shared Electronic Workspace. *CSMIL Technical Report No. 45*, The University of Michigan, Ann Arbor, 1992.

[6] Prakash, A. and Knister, M.. Undoing Actions in Collaborative Work. *Technical Report CSE-TR-125-92*, Computer Science and Engineering Division, The University of Michigan, Ann Arbor, March 1992.

[7] Stallman, R. *GNU Emacs Manual*, 1985.

[8] Teitelman, W. *Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1978.

[9] Thimbleby, H. *User Interface Design*. ACM Press, New York, 1990, pp. 261-286.

[10] Vitter, J.S. US&R: A New Framework for Redoing. In *IEEE Software* (October 1984), pp. 39-52.

[11] Yang, Y. A New Conceptual Model for Interactive User Recovery and and Command Reuse Facilities, in *Proc. CHI'88 Conference on Human Factors in Computing Systems* (Washington, D.C., May 15-19, 1988), ACM Press, pp. 165-170.