Nothing relevant.

nothing relevant

Undoing Actions in Collaborative Work[1]

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

Due to lack of full awareness of other users' intentions, the possibility of inadvertent mistakes is higher in collaborative work, and yet most current collaborative systems fail to provide adequate facilities for undoing actions. This limitation occurs because undo facilities of single-user systems do not readily apply to collaborative systems. In this paper, we propose a general framework for undoing actions in collaborative software 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 criteria.

1 Introduction

The ability to reverse, or undo, the effects of previous actions has become a common feature in modern application software. This ability is particularly valuable in collaborative applications, but it is technically much more difficult to implement than in a single-user system.

Numerous collaborative editors and other group applications have been constructed, such as GROVE [Elli90], ShrEdit [CSMIL89,91], and DistEdit [Knis90], but most 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.

Undo is important in collaborative applications because it provides freedom to interact and experiment in a shared workspace. A shared document is often used as a group blackboard during (possibly distributed) meetings. If the current state of the document contains important information, people may have inhibitions about making changes because the work is not solely theirs. Knowing that any previous state can be easily recovered may free group members to demonstrate ideas in the document. This freedom also applies to asynchronous sharing, where group members work on a shared document at different times; tentative changes can be made to the section of a document, and undone at a later time if needed.

Performing undo in collaborative applications provides technical challenges three areas: choosing the operation in 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 are many parallel streams of activity from different users, and the undo needs to be more selective about choosing what to undo. Also, a return to a previous global state could have undesirable effects because it would undo actions of all users, instead of just one. 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 conflicts between the changes the users made.

The rest of the paper is organized into the following sections:

- A review of previous work on undo.
- A discussion of how our approach extends undo capabilities, particularly for group environments.
- The requirements an application must meet to use

our undo framework, and an example for text editing.

- The undo algorithms.
- Several interface possibilities for using the undo algorithms.
- Other issues relevant to group undo, including replication and the amount of undo state to store.
- Conclusions and future work.

2 Related Work

There are several basic methods for providing undo abilities in single-user systems. We discuss them here. In our discussion, we assume that the operations that can be performed on a 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 operation of A may depend on state of the document prior to A. For instance, if a DELETE operation is done on a text document that deletes three characters at position 10, then in order to determine its inverse, we must know the three characters that were deleted. The inverse operation then will be an INSERT operation that inserts those three characters back at position 10.

2.1 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 a sequence of operations

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

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

2.2 Linear undo model and US&R model

The Interlisp system [Teit78], 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, suppose that there is a sequence of operations

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

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

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

A pointer is used to keep track of next action to be undone; in this example, the undone operation E would be the next operation which could be redone. Note that 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 [Vitt84] 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). This is potentially disruptive in a group environment; other users are likely to see their work undone for at least a short while with no apparent reason. Furthermore, if the implementation is not careful, after the redo, other users' context (cursor position, window buffer) may change unexpectedly from the original context. Also, neither model addresses the problem of conflicts – that redoing some operations may not semantically make sense if an earlier operation is skipped.

2.3 History undo

In the history undo scheme, one can undo any number (to some limit) of past actions in a row. The GNU Emacs editor [FSF85] supports history undo. Once a user stops undoing his work (by doing something other than an undo, e.g., inserting a character), the undone actions become like any other actions – they can be subsequently undone if desired. Consider the sequence of operations

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

Now, suppose 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:

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

Notice that a pointer is used to keep track of the next operation to be undone. On another undo, the history list will be as follows:

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

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

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

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

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

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.

3 Our Approach

Our approach is similar to history undo, but it allows operations to be undone selectively and deals explicitly with location shifting and conflicts.

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. In our experience, use of history is simple and intuitive for most users. 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/her last action, A_2. Normal history undo mechanisms in single-user systems do not support it 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. 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 above example, the operation to be undone, A_2, is selected based on the identity of the user. More generally, the operation selected for undoing from the history list could be selected based on any other attribute, for instance region, type, time, task, 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. We would like to undo A_2, but without undoing and then redoing B_2 and B_3.

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': $INSERT(4,'x')$ followed by $INSERT(1,'yy')$, resulting in the state 'yyabcxd'. The first operation inserted 'x' at position 4, and the second operation inserted 'yy' at position 1. Assume that these operations were done by different users. Now the user who did the first operation wishes to undo the operation. However, we cannot simply perform the first operation's inverse, $DELETE(4,1)$, because the second operation has moved the 'x' to location 6. Our scheme takes this possibility of location shifting into account, so that in this example, the first operation will be undone by executing $DELETE(6,1)$.

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.

4 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.

This section describes in detail the model and requirements which our undo framework imposes upon applications. It also demonstrates how the model can be applied to simple text documents.

4.1 Document Model

In our document model, we assume that applications modify a document using only a well-defined set of *primitive operations* which are reversible. For the purpose of undo, we treat the document much like an object, for which the primitive operations are the methods. Unlike an object, however, we do not require that operations be defined to retrieve information from the document state, only to alter it.

At the user-interface level, primitive as well as more powerful operations may be provided for modifying a document. More powerful editing operations should always map to a sequence of primitive operations. For example, assume that INSERT(location, string) is one of the primitive operations. The user-level action 'indent paragraph' might result in numerous INSERT operations. In our scheme, undoing of these more powerful operations will be implemented as a sequence of undo operations of the primitive commands (see the Section 6.4 on multiple-operation undo for more details).

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. There should exist a null state representing an empty document.

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 in sequence to a null 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.

Operations will be denoted using upper case letters, and sequences of operations using sequences of upper case letters. The operations are assumed to be performed in left to right order (left-associative). We will use the letter S to denote state prior to application of an operation. A $\circ$ placed between operations represents 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,

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

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

The parameters of an operation should fall into one of two classes: operational data and positional data. The operational data, combined with the primitive, should indicate what the operation will accomplish. The positional data indicates where in the document the operation is to be performed. For a text editing primitive INSERT, the operational data would be the text to be inserted, and the positional information would contain a position where the insert will occur. Generally, an operation could be performed in a different location by changing only its positional data.

4.2 Operation History

To provide the raw ingredients for undo, the application must maintain a history of the operations which have been performed on a document. We assume that this history will be stored as a a simple list, kept in the same order as the operations were performed. Our undo algorithms need to read this list, copy portions of it, and alter the copy. Only operations stored in the history can be undone.

Each item in the history list must contain:

- The operation which was performed.
- Any additional data required to immediately reverse an operation (stored as part of that operation).
- The user who performed the operation, and other criteria for selecting what to undo.

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 the operation or the reason for the operation. Any such tag could be used to select operations to undo.

Note that if the history is complete (contains every operation ever performed on the document), then the current state of the document could be reconstructed by performing every operation in the list in sequence.

4.3 Conflict, Re-ordering, and Reversibility of Operations

Our model requires that the application supply functions which can detect conflicts between operations, reorder operations, and create inverse operations. In a synchronous group environment, these 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. 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:

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

- $Inverse(Operation) \Rightarrow (Operation)$

The following sections provide descriptions and properties for these functions.

4.3.1 Conflict

A *conflict* between two adjacent operations A and B implies that the second operation, B, affects what the operation A has done to the state; it destroys the "integrity" of that first operation. A conflict indicates that the two operations could not have been performed in parallel with a predictable result. A conflict also arises if 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.

4.3.2 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 $S \circ A \circ B$ will give the same result as executing $S \circ B' \circ A'$, 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 still be guaranteed that the resulting state will be the same. Property 2 shows that A can meaningfully be undone, leaving only the effects of B. As we will see, operation A will usually 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 [Elli89]. However, we require transpose function to be defined only when the operations do not conflict.

4.3.3 Some useful properties

As stated earlier, an $Inverse(Operation)$ function must also be supplied by the application. $Inverse$ returns a new operation which can nullify the effects of its argument. Specifically, when the inverse of an operation is performed immediately after that operation, the resulting state is the same as if neither operation had ever been executed. Some properties that we will assume in our discussion are as follows:

Property 1: $A \circ \overline{A} \equiv I$

Property 2: $\overline{\overline{A}} \equiv A$.

Property 3: $A \equiv B \Rightarrow \overline{A} \equiv \overline{B}$

Property 4: If A and B have a conflict, then $\overline{B}$ and $\overline{A}$ also have a conflict.

Property 5: If $A \circ B$ can be transposed, then $\overline{B} \circ \overline{A}$ can also be transposed.

Property 6: $\overline{A \circ B} \equiv \overline{B} \circ \overline{A}$

The only crucial properties that we really need to hold in our algorithms are Properties 1 and 6. However, we will assume that other properties also hold while discussing the examples. Property 1 says that an operation A and its inverse $\overline{A}$ be I, the *null* or identity operation, which does not affect state, i.e. $S \circ I = S$. The operator $\equiv$ denotes that the left-hand side and right-hand side are equivalent in their effect on the document's state (excluding the history list). Property 6 states that to undo two actions, undo them one by one. Not all of the above properties are independent. For instance, Property 2 can be derived from Property 1.

4.4 Document Model Applied to Text Editing

We will now apply the document model to a specific type of document: a plain text document (with no formatting). First, the operations and state of a text document must defined. Second, the storage in a history list must be considered. Third, the *Conflict*, *Transpose*, and *Inverse* functions must be defined for the operations chosen.

The state of a text document can be modeled as a single string of text, in which line breaks are represented by newline characters. Now we must define the operations which can be used to alter the state. We choose the primitives INSERT and DELETE. INSERT is given two parameters: the location of the character before which the insert will occur, and the text to be inserted. DELETE shall also have two parameters: the starting position from which to delete, and the number of characters to be deleted. Locations are defined as the absolute position in the text, with the leftmost position being one. The two primitive operations are sufficient to make any change to a document. Additional primitives, such as REPLACE, could be added but are not necessary. Other representations of position, such as line and column number, could also be used, but the absolute positions we have chosen seem simpler.

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 operation history as follows:

- INSERT(position, 'text')
- DELETE(position, 'text')

Because the operation must store sufficient information to be reversible, we cannot simply store the number of characters for the DELETE operation. So, we store the text which was actually deleted and can easily derive the number of characters which were deleted.

Consider an example. Suppose that starting with an empty state '', this sequence of operations is performed by Mike, Atul, and Mike, respectively:

- INSERT(1, 'abcde')
- DELETE(3, 2),
- INSERT(2, 'xyz')

The resulting state after all three operations is performed would be 'axyzbe'. The history after performing these three operations might be as follows, where each row is an entry in a list, column one contains operations, column two contains the user tag, and column three contains a timestamp tag.

INSERT(1, 'abcde')	Mike	1:05:32
DELETE(3, 'cd')	Atul	1:05:37
INSERT(2, 'xyz')	Mike	3:55:27

Finally, we define the *Conflict*, *Transpose*, and *Inverse* functions for the INSERT and DELETE primitives. We begin with a simple utility function which determines whether two regions of character positions overlap:

$$\begin{aligned} Overlap(pos_1, len_1, pos_2, len_2) = \quad & (pos_2 + len_2 - 1 \geq pos_1) \\ AND \quad & (pos_2 \leq pos_1 + len_1 - 1) \end{aligned}$$

Now we show the *Conflict* function, which, given two operations, determines whether the second operation conflicts with the first. An operation will conflict with an INSERT if it alters the text which was inserted. An operation will conflict with a DELETE if it alters the two characters bordering the DELETE. The reason for using the border characters for DELETE is can be seen in the following example. Suppose we begin with state 'abc' and perform DELETE(2, 1) followed by INSERT(2, 'x'), resulting in state 'axc'. If we later wish to undo the DELETE, it is not clear whether the 'b' should be placed before or after the 'x'. Therefore, this is a conflict. This definition may be conservative in the case of multiple deletes, but it is safe.

$$\begin{aligned} Conflict(\quad & INSERT(pos_1, str_1), \\ & INSERT(pos_2, str_2)) \quad = \quad pos_1 < pos_2 < pos_1 + |str_1 \\ Conflict(\quad & INSERT(pos_1, str_1), \\ & DELETE(pos_2, str_2)) \quad = \quad Overlap(pos_1, |str_1|, pos_2, | \\ Conflict(\quad & DELETE(pos_1, str_1), \\ & INSERT(pos_2, str_2)) \quad = \quad (pos_2 = pos_1) \\ Conflict(\quad & DELETE(pos_1, str_1), \\ & DELETE(pos_2, str_2)) \quad = \quad Overlaps(pos_1 - 1, 2, pos_2, \end{aligned}$$

Now we define *transpose*. Note that the transpose function must be well-defined only when there are no conflicts between the two operations.

$$\begin{aligned} & Transpose(INSERT(pos_1, str_1), INSERT(pos_2, str_2)) = \\ & \quad if(pos_2 > pos_1) \quad then(INSERT(pos_2 - |str_1|, str_2), INSERT(po \\ & \quad \quad else(INSERT(pos_2, str_2), INSERT(pos_1 + |str \\ & Transpose(INSERT(pos_1, str_1), DELETE(pos_2, str_2)) = \\ & \quad if(pos_2 > pos_1) \quad then(DELETE(pos_2 - |str_1|, str_2), INSERT(po \\ & \quad \quad else(DELETE(pos_2, str_2), INSERT(pos_1 - |str \\ & Transpose(DELETE(pos_1, str_1), INSERT(pos_2, str_2)) = \\ & \quad if(pos_2 > pos_1) \quad then(INSERT(pos_2 + |str_1|, str_2), DELETE(po \\ & \quad \quad else(INSERT(pos_2, str_2), DELETE(pos_1 + |str \\ & Transpose(DELETE(pos_1, str_1), DELETE(pos_2, str_2)) = \\ & \quad if(pos_2 \geq pos_1) \quad then(DELETE(pos_2 + |str_1|, str_2), DELETE(p \\ & \quad \quad else(DELETE(pos_2, str_2), DELETE(pos_1 - |st \end{aligned}$$

Finally, the *inverse* functions are:

$$\begin{aligned} Inverse(INSERT(pos, str)) &= DELETE(pos, str) \\ Inverse(DELETE(pos, str)) &= INSERT(pos, str) \end{aligned}$$

5 Undo Algorithms

This section presents three algorithms: a simple undo algorithm to demonstrate the basic concepts, a comprehensive undo which handles multiple undo operations, and an algorithm to derive all conflicting operations which prevent an undo.

All three algorithms assume that an operation has already been chosen to be undone. Methods of selecting which operation to undo are described in Section 6 on Undo Interfacing.

In our description of the algorithms, we assume that only one (centralized) history list is maintained for all users. We will discuss issues related to replication of editor state and history list in Section 7.1.

We also assume that all operations will be done, as requested by the users; There are no failures and no unintended effects. If necessary, editor should provide some sort of locking scheme so that two users do not perform conflicting operations in parallel.

5.1 Data Definitions

The following data types are used by the algorithms:

Operation A primitive operation, including the operational data and positional data.

History List A list of operations, including tag information such as user and time.

Figure 1 shows the structure of the history list in more detail. *History List* is a list of records in which each record contains an operation and the information used to locate operations, such as the user who performed the operation, the time at which it was executed, and any other information which will be used to select operations to undo. The algorithms are not concerned with the details of operation records such as the location and other internal data; the *Inverse*, *Conflict*, and *Transpose* operations are used to manipulate operations. The *Perform* routine performs an operation, altering the document state, and appends the operation to the end of the history list.

```
type Operation = record
    prim: Primitive;
    locationData: Application-dependent-type;
    operationData: Application-dependent-type;
end
HistoryRec= record
    operation: Operation;
    user: User;
    time: Timestamp;
    next: HistoryRec;
    /* any other desired tags go here */
end

var
HistoryList : HistoryItem; /* Stored history for the document */

function Perform(op: Operation): HistoryRec
/* Performs operation on the document, returns the new element
 * added to the history list
 */
```

Figure 1: Data types used in the undo algorithms

5.2 Simple Undo

To demonstrate the principals of our undo technique, we first present an algorithm for the simple undo, in which only one previous operation can be undone. Since the operation to be undone is not necessarily the one at the end of the history list, the operation to be undone is passed to the algorithm. The algorithm is given in Figure 2.

The basic idea is to use the *transpose* function to shift the operation all the way to 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. Note that the the history list is not being altered in the algorithm; the shifting is simulated.

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

$$A\ B\ C$$

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

```
procedure SimpleUndo(UndoItem: HistoryItem)
/* Undo the UndoItem, which is a pointer into the HistoryList */
var ShiftOp: Operation
    HistPtr: HistoryRec; /* Pointer into the history list */
begin
    ShiftOp := UndoItem.operation;
    HistPtr := UndoItem.next;
    while HistPtr.next <> nil do
        if Conflicts(ShiftOp, HistPtr.next.operation) then
            return ('Sorry. Undo conflicts with ', HistPtr.next)
        else
            /* Transpose returns two operations; store the 2nd in Sh
            (_, ShiftOp) := Transpose (ShiftOp, History[i])
        endif
    endwhile
    Perform(Inverse(ShiftOp))
    return ('Undo successful')
end
```

Figure 2: Single-step undo in collaborative applications

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

where $(B', A') = Transpose(A, B)$. Now, if $Conflicts(A', C)$ is true, the undo will fail. Otherwise, another shift will occur, resulting in:

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

where $(C', A'') = Transpose(A', C)$. To see that this is correct, consider what would happen if we perform $\overline{A''}$ on the altered list, giving:

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

Since $B' \circ C' \circ A''$ is equivalent ($\equiv$) to $A \circ B \circ C$ (by Transpose Property 1) and $B' \circ C' \circ A'' o \overline{A''} \equiv B' \circ C'$ (by 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!

While the simple algorithm is correct, it is unable to deal with the results of prior undo operations. For example, suppose the history contains $A \circ B \circ C$, where A and B conflict but neither conflicts with C. A user, wanting to undo both A and B, first does a simple undo of B, resulting in the history $A \; B \; C \; \overline{B'}$. Then, the user attempts to undo A. Simple 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.

5.3 Comprehensive Undo

We now give a comprehensive undo algorithm which is not blocked by prior undo operations. Furthermore, we demonstrate that the algorithm will work for undoing undo operations.

To perform a comprehensive undo, we must know when one operation is the undo (inverse) of another. We can detect this condition by, whenever an undo is performed, placing a pointer into the history list that links an operation to its undo. 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:

$$\begin{tikzcd} A \arrow[rrr, dash, bend right] & B & C & \overline{B'} \end{tikzcd}$$

The Comprehensive Undo algorithm (Figure 3) is same as the simple undo algorithm, except that it uses a more

```
type HistoryRec = record
    operation: Operation;
    next: HistoryRec;
    undoneBy: HistoryRec;                                   /* This fie
end


procedure Undo(UndoItem: HistoryItem)
var
    HistTemp: HistoryRec;                                   /* A copy
    HistPtr: HistoryRec;                                    /* Pointe
    ShiftOp: Operation;
    NewItem: HistoryRec;

begin
    /* Make a copy of the history list from the UndoItem forward */
    HistTemp := CopyTailofList(UndoItem);
    ShiftOp := HistTemp.operation;
    HistPtr := HistTemp.next;
    /* Shift UndoItem forward, removing all paired do/undo operatio
    while HistPtr.next <> nil do
        if TrulyConflict(HistPtr) then
            return ('Sorry. Conflicts with', HistPtr.next)
        else
            /* Transpose returns two operations; store the 2nd in Sh
            (_, ShiftOp) := Transpose (ShiftOp, History[i])
        endif
    endwhile
    NewItem := Perform(Inverse(ShiftOp));
    UndoItem.undoneBy := NewItem;
    return ('Undo successful');
end
```

Figure 3: Comprehensive undo

sophisticated conflict-checking algorithm, *TrulyConflict* (Figure 4). 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 until it reaches the end of the list. Before each shift, *TrulyConflict* is called to check if there is a conflict between the operation and the next 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 history list by *RemoveDoUndoPair* (Figure 5).

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 (Property 1). 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

```
function TrulyConflict(HistPtr: HistoryRec): boolean
    /* This function determines whether the operation in HistPtr
     * conflicts with the following operation, ignoring any operations
     * which have been undone, and stripping them from the history
     * list. Like Conflict(), returns True/False
     */

begin
    while Conflict(HistPtr.operation, HistPtr.next.operation) do
        if HistPtr.next.undoneBy <> nil then
            RemoveDoUndoPair(HistPtr.next)
        else return (False)
        endif
    endwhile
    return (True)
end
```

Figure 4: Check if there is a genuine conflict between an operation and the following operation

```
procedure RemoveDoUndoPair(doPtr: HistoryRec)
    /* This subroutine, given a pointer to an operation which is
     * later undone, physically shifts it forward in the HistTemp
     * list until it meets its undo, then removes both operations.
     * Assume: doPtr.undoneBy points to the undo operation.
     *      Any intervening conflicts have been undone,
     *      otherwise doPtr could not have been undone.
    */
begin
    while doPtr.next <> doPtr.undoneBy do
        if Conflicts(doPtr.operation, doPtr.next.operation) 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.operation, doPtr.operation) =
                    Transpose(doPtr.operation, doPtr.next.operation);
            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)
end
```

Figure 5: Remove an operation and its Undo from the temporary copy of the history list

of such an intervening Y, *RemoveDoUndoPair* is called recursively to first eliminate Y from the history list.

5.3.1 An Example of Comprehensive Undo

Suppose the history list at some point is as follows:

$$A\ B\ C\ D$$

Assume that operations B and C conflict and other than that, there are no 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:

$$\begin{tikzcd}A & B & C \arrow[rr, bend right, dash] & D & \overline{C'}\end{tikzcd}$$

Now, suppose operation B is to be undone. The algorithm will first copy *HistoryList* will be copied into *TempHistoryList* so that the original list is not affected by shifting operations. Then, *TrulyConflict()* will be called to check for conflict between B and C. *TrulyConflict()* will detect a conflict between B and C, but will notice that C has a pointer to its undo operation $\overline{C'}$. It will therefore call *RemoveDoUndoPair*() to remove the C and $\overline{C'}$ pair. The resulting (temporary) history list will be as follows, where $(D', C') = Transpose(C, D)$:

$$A\ B\ D'$$

TrulyConflict() continues to check for conflict between B and the following operation and returns false to *ComprehensiveUndo()* because there is no conflict between B and D'. After completion of the shifting operation (while loop) in *ComprehensiveUndo()*, the temporary history list will be as follows:

$$A\ D''\ B'$$

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 result:

$$\begin{tikzcd}A & B \arrow[rrrr, bend right, dash] & C \arrow[rr, bend right, dash] & D & \overline{C'} & \overline{B'}\end{tikzcd}$$

Note that in this configuration, it is not possible to redo C (by undoing $\overline{C'}$) because, assuming Property 5, $\overline{C'}$ would conflict with $\overline{B'}$, an operation that has not been undone. However, it is possible to undo $\overline{B'}$ and then undo $\overline{C'}$ without any problems. It is also possible to undo D if it does not conflict with $\overline{C'}$ or $\overline{B'}$.

5.4 Conflict List Generation

The Conflict List Generation algorithm computes a list of all operations which must be undone prior to undoing a given operation This capability can be very useful in creating the user interface for undo.

Based on the comprehensive undo algorithm, conflict list generation algorithm, *FindConflict*, works by shifting the input operation A to the end of a copy of the history list (Figure 6. However, when a conflicting operation C prevents a shift, C is undone. Should C conflict with any later operations, those too are recursively undone. The undo operations are not actually performed on the document, only simulated, and each conflict is added to a list. And, operations which have already been undone are ignored for the purpose of conflicts.

If each item in the resulting list is actually undone, most recent operation being undo first, the input operation will have no conflicts in the history and can be undone.

5.5 Performance of Algorithms

The Comprehensive Undo and Conflict List Generation algorithms have worst case run times of $O(n^2)$, where n is the number of operations between the operation to be undone and the end of the history list. For the undo, a worst case example would be undoing A for the sequence:

$$A \quad B_1 \ldots B_n \overline{B_n} \ldots \overline{B_1}$$

and where A conflicts with every B_i. In this case, B_1 must be transposed until it is before $\overline{B_1}$, and the same for every B_i. A similar situation exists for the Conflict List algorithm.

6 Undo Interfacing

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 user interfaces possible using our undo framework and algorithms. Following are some sample interface methods.

6.1 Individual History Undo

The Emacs-style history undo described in Section 2.3 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 comprehensive undo algorithm is then applied to the

```
procedure FindConflicts(UndoItem: HistoryItem)
var
    HistTemp: HistoryRec;                               /* A copy
    Conflicts:HistoryRec;


begin
    /* Make a copy of the history list from the UndoItem forward */
    HistTemp := CopyTailofList(UndoItem);
    Conflicts := NIL;

    RecFindConflicts(HistTemp);
    /* Return all conflicting operations, with UndoItem at the end o
    return (Conflicts)
end

procedure RecFindConflicts(doPtr: HistoryRec)
/* Recursively undo doPtr operation, and undo all conflicts,
  * storing the conflicts and not actually performing operations
  */
begin
    while doPtr.next <> nil do
        if TrulyConflict(doPtr) then
            RecFindConflicts(doPtr.next)
        else
            /* Transpose operations logically and physically */
            (doPtr.next.operation, doPtr.operation) =
            Transpose(doPtr.operation, doPtr.next.operation)
            ListSwap(doPtr, doPtr.next)
        endif
    endwhile
    /* Moved it to end of history. Add to conflict list & delete */
    ListAppend(Conflicts, doPtr)
    ListDelete(HistTemp, doPtr)
end
```

Figure 6: Conflict List Generation Algorithm

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 if 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, the Conflict List Generation algorithm can be used to locate all 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 canceling or proceeding to undo the operations of those other users. In the latter case, each further undo would operate on an item in the conflict list.

6.2 History Undo with Selection Filters

The history undo process need not be restricted to undoing one users changes. Any arbitrary filtering criteria can be used to select the next operation to undo while searching back through the history, as long as the necessary information is stored in the history list.

For example, history undo could use a filter to repeatedly undo actions for a set of users, for a time range, for a region of the document, for a particular task, or for any combination of these parameters.

The Individual History Undo is simply history undo with a filter which selects only operations of one user.

6.3 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 affect a region R, we start by defining a new operation S which we know will conflict with any operation performed in R. For instance, S might be an operation which deletes all of R; certainly any operation affecting R would conflict with this. 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.

To implement repeated undo operations, it may be useful to define a new primitive for regional undo; otherwise a repeated undo will fail. The issue is there because after the first undo, the region could change. What is needed is a region-identifying primitive that can be transposed with any operation even if an overlap in region exists.

6.4 Multi-operation actions

Situations may arise in which the 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.

- Inverse of a primitive operation may not be a primitive operation, but a collection of primitive operations. Again, that collection has to be treated as a single, high-level, operation in case it needs to undone.

- Undoing many steps at once can also 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 for the region covering chapter 15 with timestamps after 5PM last Tuesday), assuming sufficient history with appropriate tags is kept.

Multiple-operation undo is similar to the notion of *transactions* in databases. Either they should all be undone collectively, or conflicts should be reported and undone 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 would be hard to understand for the user.

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 collectively (using Property 6). If conflicts arise during shifting, undo should not be permitted without undoing the conflicts first. The collection of operations that are used to undo need to be treated as a high-level operation.

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

Transaction processing may lead to inefficiencies in a group environment because it hinders tight interactions between users [Elli91, Elli89]. However, for a multi-operation undo, it is highly desirable to ensure atomicity, perhaps through use of locks, so that an undo has a predictable effect.

We are still exploring whether there are important semantic or efficiency issues that may sometimes make it more appropriate to consider a high-level operation as a new primitive operation, rather than a collection of existing primitives.

7 Other Issues

7.1 Replication Issues

In a distributed environment, it is highly desirable to replicate the document, maintaining a copy at each users' site, to keep response time short. If the data were kept only at a central site, each time someone merely navigated through the document, they would have to wait for a round-trip network delay before getting feedback. Also, central data storage provides a single point of failure.

When a program replicates data, it must provide a means of concurrency control which ensures that all copies of the document are the same (or nearly so, within some bounds). This generally involves broadcasting operations to all users in combination with some form of locking or re-sequencing. [Elli91] discusses various approaches to concurrency control which vary in their response time, flexibility, and consistency guarantees.

Replication raises several questions with respect to undo:

- Should the history list be replicated?
- What messages should be broadcast for replication?
- Can replicated histories differ between users?

Deciding whether to replicate history is a trade-off between performance, storage requirements, and fault-tolerance. Because undo is probably less commonly used and less critical compared to most other operations, it might be practical to store the history at an undo server, if some delay can be tolerated.

If the history information is replicated, a decision must be made whether the semantics of an undo will be broadcast to the group, or just the results of the undo operation. If the undo itself is not broadcast for an operation, other users will not be able to ignore the undone operation in executing further undo operations. Thus, it is probably preferable to broadcast undo semantics.

Replicated histories may or may not be kept identically for all users at all times; for example, the ordering of operations might vary between different users' history lists. If the histories are not identical, the same undo operations may work differently at each location, but must guarantee the same result. Thus, concurrency control issues arise for replicating history which are very similar to and dependent upon concurrency techniques used for the actual document state.

7.2 Length of History List

In both single-user and collaborative applications, the length of the history list would dictate how far back operations can be undone. However, in single-user systems, it is easy to provide a guarantee that the user will be able to undo at least his last operation with a bounded number of operations on history list – the history list needs to only keep one operation, the last one. In collaborative applications, on the other hand, providing such a guarantee is difficult with a bounded number of operations on the history list. Say user X does an operation. Then, user Y does a sequence of operations. Now, in order to guarantee that X is able to undo his operation, the X's operation as well as all the Y's operations have to be kept on the history list. Either the history list has to be allowed to grow as needed, or the users have to be prepared to, occasionally, not be able to undo their last operation if other users have been active and they haven't been active for a while.

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 DistEdit toolkit [Knis90]. Hopefully, the framework and prototype will lead to behavioral science work exploring the right interfaces for carrying out undo operations in collaborative applications. We are also exploring whether our shifting and conflict-checking strategy could be applied to carry out an operation retroactively to have the same effect as an ⟨undo,do new operations,redo⟩ sequence, but without necessarily doing an undo-redo cycle as in the linear and US&R models. For situations where shifting is difficult to carry out, say due to too much dependence of *Transpose* and *Inverse* functions on prior state, we are investigating the integration of undo-redo schemes with our model.

In addition, we are investigating the uses of the history list in other contexts, particularly to keep track of evolution of a document at a fine level of granularity.

References

[CSMIL89,91] Cognitive Science and Machine Intelligence Laboratory, "ShrEdit, A Multi-user Shared Text Editor: User Manual," The University of Michigan, 1989 and 1991.

[Elli89] C.A. Ellis and S.J. Gibbs, "Concurrency Control in Groupware Systems", *Proceedings of the ACM SIGMOD '89 Conference on the Management of Data*, Seattle, Washington, May 1989.

[Elli90] C.A. Ellis, S.J. Gibbs, and G.L. Rein, "Design and Use of a Group Editor," in *Engineering for Human-Computer Interaction*, G. Cockton, Ed., North-Holland, Amsterdam, 1990, pp. 13-25.

[Elli91] C.A. Ellis, S.J. Gibbs, and G.L. Rein, "Groupware: Some Issues and Experiences", *Communications of the ACM*, January 1991, pp. 38-58.

[FSF85] R. Stallman, *GNU Emacs Manual*, 1985.

[Knis90] M. Knister and A. Prakash, "DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors", *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, Los Angeles, California, October 1990, pp. 343-355.

[Teit78] W. Teitelman, *Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1978.

[Vitt84] J.S. Vitter, "US&R: A New Framework for Redoing", *IEEE Software*, October 1984, pp. 39-52.