A Framework for Undoing Actions in Collaborative Systems

ATUL PRAKASH
MICHAEL J. KNISTER

University of Michigan, Ann Arbor

The ability to undo operations is a standard feature in most single-user interactive applications. In this paper, we propose a general framework for implementing undo in collaborative systems. The framework allows users to individually reverse their own changes, taking into account the possibility of conflicts between different users' operations that may prevent an undo. The proposed framework has been incorporated into DistEdit, a toolkit for building group text-editors. Based on our experience with DistEdit's undo facilities, we discuss several issues that need to be taken into account in using the framework, in order to ensure that a reasonable undo behavior is provided to users. We show that the framework is also applicable to single-user systems, since the operations to undo can be selected not just on the basis of who performed them, but by any appropriate criterion, such as the document region in which the operations occurred or the time interval in which the operations were carried out.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques – *User Interfaces*; H.1.2 [**Models and Principles**]: User/Machine Systems – *Human Factors*; H.2.2 [**Database Management**]: Physical Design – *Recovery and Restart*; H.2.4 [**Database Management**]: Systems – *Concurrency*; H.5.2 [**Human Interfaces and Presentation**]: User Interfaces – *Theory and Methods*; H.5.3 [**Human Interfaces and Presentation**]: Group and Organization Interfaces

General Terms: Algorithms, Design, Human Factors

Additional keywords and Phrases: Undo, groupware, computer supported cooperative work, selective undo, DistEdit, user recovery, state recovery, concurrency control.

This work was supported in part by the National Science Foundation under the cooperative agreement IRI-9216848. A preliminary report on this research appears in the *Proceedings of the Fourth ACM Conference on Computer-Supported Cooperative Work* (Toronto, Canada, October 1992), pp. 273-280.
Author's addresses: A. Prakash, Software Systems Research Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122; email: aprakash@eecs.umich.edu; M. J. Knister, 930 Tahoe Blvd #802-185, Incline Village, NV 89451; email: knister@libre.com.

1 Introduction

The ability to undo operations is a standard and useful feature in most interactive single-user applications. For instance, the availability of an undo facility in editors is useful for reversing erroneous actions [19]. It can also help reduce user frustration with new systems [14], particularly if those systems allow users to invoke commands that can modify system state in complex ways. The availability of undo can also encourage users to experiment, by acting not only a safety net but also by allowing users to try out different approaches to solving a problem via backtracking [39].

In recent years, there has been a growing interest in the area of *computer-supported cooperative work*, or *groupware*, the goal of which is to provide support for collaborative work among users in a shared workspace [2, 3, 11, 25, 36]. An undo facility can be important in groupware systems for several reasons. Groupware systems, whether used by an individual or by a group, should provide abilities, including undo, comparable to similar single-user tools. Another reason is that having an ability to undo in a group environment may provide freedom to interact and experiment in a shared workspace. A shared document is often used as a group white-board 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 their work can be undone without undoing other users' work may encourage group members to freely express their ideas in the document.

Compared to single-user applications, performing undo in groupware applications provides technical challenges in following areas: *selecting* the operation to be undone, determining *what* operation will result in a correct undo, and dealing with *dependencies* between different users' operations. First, in a group environment, there may be parallel streams of activities from different users. When work on a shared document occurs in parallel, users usually expect an undo to reverse their own last operation rather than the globally-last operation, which may belong to another user. An undo framework for groupware systems needs to allow selection of operation to undo based on who performed it. Second, once the correct operation to be undone is selected, the operation to execute to effect an undo has to be determined. Simply executing the inverse of the operation to be undone may not work because of modifications done by other users to the document. Finally, if multiple users interleave their work in the same region of a document, it may not be possible to undo one user's changes without undoing some of the other users' changes. In this case, there are dependencies between the changes which need to be taken into account during an undo.

Many groupware applications have been built that support multi-user work on a shared document, e.g, Grove [9], ShrEdit[26], CES [17], and MACE [28]. None, as far as we are aware, provide an undo facility that addresses all the above issues. Those applications that do support undo usually only provide a global undo facility rather than a per-user undo facility. MACE [28] does support a simple form of per-user undo, allowing users to undo their own modifications made to a section provided they acquire a lock on the section prior to making modifications and do not release the lock prior to the undo.

This paper presents a framework for implementing undo in groupware applications that addresses the above-mentioned technical issues. The framework is quite general, being applicable to a variety of documents types, such as text, graphics, and multimedia. The proposed techniques have been incorporated into DistEdit [21, 22], a group text-editor toolkit, and into a version of SASSE [5], a group editor. The basic ideas of our undo framework were presented in an earlier version of this paper [32]. This paper makes several additional contributions. First, it includes a detailed discussion of properties that editing operations should satisfy in order to help provide correct undo behavior to users. Second, it includes a comparison with various group undo techniques that have been suggested recently. Third, it addresses design and efficiency issues that occur in implementing the framework. Finally, it includes our experience in using the proposed undo facilities

in the DistEdit toolkit.

Several groupware applications, such as Quilt [13] and Prep [27], support *asynchronous* sharing, where group members work on a shared document at different times. Our undo framework can be applied to these applications as well — users can make tentative changes to a document, knowing that their changes can be undone at a later time if needed, even if other users also subsequently modify the document.

The proposed undo framework can also be applied to single-user editors for undoing changes selectively. For example, the framework is used in DistEdit to provide a *region-undo* facility that allows users to undo only changes made to a selected region of a document. Other possible applications include undoing modifications made during a certain time interval, by a set of users, or by a combination of these selection conditions.

The rest of the paper is organized as follows. Section 2 discusses the related background work in the area. Section 3 discusses our approach to providing undo capabilities in group environments. Section 4 describes the application requirements and the document model needed in order to use our undo framework. Section 5 describes our selective undo algorithm and contains proofs for several useful properties of the algorithm. Section 6 suggests ways to improve the performance of the algorithm for many common situations. Section 7 describes some methods used in DistEdit for selecting operations to undo. Section 8 describes our experience with DistEdit's undo facilities, problems that arose, and ways of addressing those problems. Finally, Section 9 presents our conclusions and directions for future work.

2 Related Work

There are several basic methods for providing undo capability, most of them designed for single-user systems. Almost all of them require maintaining a *history list*, 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 order in which they were performed. For instance, suppose the history list is as follows:

$$A\ B\ C\ D$$

Then, starting from the state prior to A, and performing the operations A, B, C, D in sequence should lead to the current state.

Furthermore, most undo schemes assume, as does the scheme in this paper, that the 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. In general, the inverse of an operation A may depend on state of the document prior to A [15]. For example, 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. The operations stored in the history list should contain sufficient data so that their inverses 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.

Below, we summarize the primary methods for doing undo in single-user systems. A more detailed discussion of these techniques can be found in [39], and one formalization of undo and redo facilities can be found in [41]. We also discuss undo techniques that have been proposed for groupware systems.

2.1 Single-step undo

Several early editors, including Lampson's Bravo for the Alto [23] and Hammer *et al*'s ETUDE editor/formatter [18], provided single-step undo. Single-step undo is also available in many current

systems, including most Macintosh and Windows applications, as well as editors such as *vi*. Single-step undo allows undo of only the last operation. For instance, suppose the history list is as follows:

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

Single-step undo allows undoing of operation E, but not a subsequent undo of operation D. Usually the *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.

2.2 Linear undo model and US&R model

The Interlisp system [38], COPE [4], and Aloe [24] are examples of systems that use the linear undo model. The linear undo model allows the undoing of a sequence of operations by using a pointer that tracks the next operation to be undone. Operations can then be redone, after possibly doing some new operations. For example, 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, resulting in the following history list:

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

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, inverse operations are not explicitly stored in the history list. Thus, reverting back to the original state (without the F) is not possible. One could undo F, but then D and E must be done manually.

The Undo, Skip, Redo (US&R) model [40] supports redo like the linear undo model but also allows skipping of some operations during the redo. Instead of a linear list, the 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 X several steps back in the history, all subsequent operations must first be undone and then redone (skipping X during the redo). If not implemented carefully, 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 re-doing operations may lead to incorrect or unexpected results if an earlier operation is skipped.

2.3 History undo

The history undo scheme, used in the *Emacs* editor [35], also allows undoing of a sequence of operations, but, unlike the linear undo and the US&R schemes, it appends the inverse operations to the end of the history list. The inverse operations in the history list are treated like 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 history undo, the history list will be as follows, where $\overline{E}$ is the operation that reverses the effects of E and the pointer indicates the next operation to be undone:

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

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

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

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

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

History undo has the nice property that it is possible to go back to any previous state, and the need for tracking dependencies among operations does not arise since operations are never skipped.

2.4 Work in databases

Strategies used for implementing undo in editors differ somewhat from those used for aborting transactions in databases. In databases, the modifications to data due to a transaction usually become visible to other transactions only after the transaction has committed. In addition, a transaction cannot be aborted once it is committed. In contrast, an editing operation can be undone by users at any time, even after the effect of the operation has been seen by other users. This complicates the design of undo algorithms because dependencies between the operation being undone and later operations can arise.

Furthermore, databases typically use a checkpoint and recovery strategy for aborting transactions and for dealing with failures [16]. Such schemes are typically not used for implementing undo because in editor-type applications it is usually more efficient to execute the inverse of operations to get back to an earlier state.

2.5 Work on optimistic concurrency control

The capability to undo/redo operations has been used in optimistic concurrency control schemes for maintaining consistency of replicated data. For instance, Karesenty and Beaudouin-Lafon [20] describe an algorithm that uses the undo/redo capability to ensure consistency among replicated copies of a document during group editing. The following example illustrates the use of undo/redo in their scheme. Suppose that one of the sites in a groupware system executes an operation B after receiving the broadcast of an operation A, which was issued at another site. However, a third site receives the broadcast of operation B first, executes it, and then receives the broadcast of operation A. Their algorithm will allow out-of-order execution of A at the third site if A and B commute or if execution of A makes executing B unnecessary. Otherwise, it will "undo" A, execute B, and then "redo" B to correct the execution order. The undo/redo here is internal to the system and is used only for ensuring consistency. No undo/redo capability is provided to end-users. In particular, support for undoing operations that were executed in the correct execution order is not provided.

2.6 Group-undo schemes

There has been substantial interest in the design of undo facilities for groupware systems recently. Independently and about the same time as our work [32, 33], undo schemes for collaborative systems were proposed by Berlage and Genau [6], Chaudhary and Dewan [7], and Abowd and Dix [1]. We provide a comparison with these schemes below.

Berlage and Genau [6] and Chaudhary and Dewan [7] suggest undoing any operation in the history list simply by executing its inverse, provided the inverse is executable in the current state. This approach essentially assumes that any operation in the history list can be undone simply by executing its inverse from the current state, irrespective of the following operations on the history list. Unfortunately, not taking into account dependencies among operations can lead to unexpected or hard-to-predict undo behavior in certain situations. To see some of the problems that arise when operations have dependencies among them, consider the following simple example. Let's say that a graphical document contains a circle of size 6 and that the following two operations are done, leading to a circle of size 4:

- *Operation 1:* double the radius of the circle
- *Operation 2:* set the radius of the circle to 4

Assume that the inverses of the above operations are chosen to be:

- halve the radius of the circle
- restore the radius of the circle to 12 (the size prior to doing operation 2)

Suppose a user wishes to undo operation 1. Using the above undo scheme, the inverse of operation 1 is executed, resulting in a circle of radius 2. Since the circle was never of size 2, this may be a result that is difficult for the user to understand. Another problem with the scheme is that the result of undoing a set of operations may depend on the *order* in which the operations are undone. In the above example, one can end up with a circle of size 12 or a circle of size 6, depending on the order in which the above two operations are undone. Note that one of the possible results is different from 6, the initial size of the circle. The framework and algorithms described in this paper are more general and take into account the possibilities of conflicts and dependencies among operations.

Abowd and Dix [1] recognize the need to deal with dependencies among users' operations and suggest a basic framework similar to that described in this paper for dealing with dependencies. The focus in their work has been on trying to understand formally the behavior desired of undo in a group environment. We provide algorithms for implementing group undo schemes and report our usage experience. Furthermore, we provide formal properties that the undo framework should satisfy in order for the undo to work correctly in a group environment.

3 Our Approach — Selective Undo

To provide undo facilities in groupware applications, we now present an approach called *selective undo* and describe algorithms for implementing it. The approach is based on history undo, but we allow operations to be undone selectively and deal explicitly with location shifting and dependencies among users' operations. 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 the linear and the 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 to the end of the history list. However, in a groupware application, since the last operation done by a user is not necessarily *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 one user, say Ann, and B_i's refer to operations done by other users:

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

Now, suppose Ann wishes to undo her last operation, 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 Ann may not be aware that operations B_2 and B_3 have been carried out on the document, and other users may not be aware of changes made by Ann. Using 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. To allow such selection, each operation on the history list is *tagged* with appropriate selection attributes, such as user id and time of the operation. We term our scheme *selective undo*, since the operation to be undone is not necessarily the last one, but is selected using some attributes attached to the operation.

To selectively undo an operation, we cannot simply execute the inverse of the operation because subsequent operations could have shifted the location at which the operation was originally performed. For example, suppose the following two text operations are applied to the starting state *abcd*: $InsChar(4,'x')$ followed by $InsChar(1,'y')$. The first operation inserts 'x' at position 4 and the second operation inserts 'y' at position 1, resulting in the state *yabcxd*. 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 *dependencies*, or *conflicts*. In the above example, B_2 may have modified the same region of the document as A_2; so it may not be possible to undo A_2 without first undoing B_2. Our framework detects when an operation cannot be undone because of later dependent operations that have not been undone.

In any undo scheme, it is important that undo behave according to users' expectations. Our experience indicates that a naive implementation of selective undo can easily behave unexpectedly or even give incorrect results. We provide formal properties that the undo framework should satisfy in order to help ensure that undo behaves correctly and according to users' expectations.

4 Document Model and Application Requirements

4.1 Document model

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 dependencies 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. Operations can 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.

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. We use the letter S to denote the 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.

We assume the existence of an identity operation, I, which leaves the document state unchanged. In other words, for any document state S,

$$S \circ I = S$$

The following definition is useful later on in the paper:

Definition: Two sequences (lists) of operations are said to be **equivalent** if they produce the same state, given the same initial state.

For example, a sequence of operations $M\ N$ is equivalent to a sequence of operations $P\ Q$ if, for any initial state S,

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

4.2 Conflict, re-ordering, and reversibility of operations

To allow an arbitrary operation on the history list to be undone, our model requires that the application supply functions which can detect *dependence* between operations, *re-order* independent operations, and create *inverse* operations. In a synchronous group environment, similar functions are usually 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, dependence checking may involve making sure that their changes do not overlap, e.g., through the use of locks [22]. Mechanisms for reordering two parallel, independent operations are also needed because the order in which the two operations will be executed may be unpredictable [10]. The editor must be prepared to accept the two operations in either order with the same resulting effect.

The functions that the application must provide for selective undo are:

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

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. Below, we provide descriptions and properties for Conflict, Transpose, and Inverse functions.

4.2.1 Conflict function

Operations on a document can have dependencies among them that may prevent them from being reordered. Suppose, for example, that a graphics document is being edited. Operation A creates a circle in the document, and operation B resizes that circle. Clearly, operation B cannot be executed prior to the execution of operation A.

The $Conflict(A, B)$ function supplied by the application must return $true$ if the adjacent operations A and B performed in sequence cannot be reordered, and $false$ otherwise[1]. The importance of the notion of conflict is that it imposes an ordering on operations A and B. If $Conflict(A, B)$ is true, then the order of operations A and B cannot, in general, be changed without affecting the results. Furthermore, in general, operation A cannot be undone, unless the following operation B is undone too.

In the discussion that follows, we will say that A *conflicts* with B when $Conflict(A, B)$ is true. We will say that A and B are *independent* or *non-conflicting* if $Conflict(A, B)$ is false.

[1] A conflict between two operations carried out by two users does not imply that the work done by the users is not cooperative. The notion of conflict defined above is simply a formal notion to determine whether two operations can be reordered. It does not indicate in any way whether the users are cooperative or non-cooperative.

4.2.2 Transpose function

If an operation A does not conflict with an operation B, we require that it be possible to *reorder* or *transpose* A and B. That is, after possibly making some adjustments to the operations, it must be possible to perform them in the reverse order and still obtain the same result. The Transpose function is used to reorder operations. For any two adjacent operations A and B, if A does not conflict with B, $Transpose(A, B)$ must return (B', A'), a reordering of the two operations; otherwise, $Transpose(A, B)$ is undefined.

For notational convenience, we will introduce an additional function, $RShift(A, B)$, defined as follows: if $Transpose(A, B) = (B', A')$, then $RShift(A, B) = A'$. Otherwise $RShift(A, B)$ is undefined.

We require that the following properties be satisfied by the Transpose and RShift functions:

Transpose Property 1: If $Transpose(A, B) = (B', A')$ then $S \circ A \circ B = S \circ B' \circ A'$.

Transpose Property 2: If $Transpose(A, B) = (B', A')$, then B' is the operation that would have been executed, instead of B, if operation A had not been executed earlier. Furthermore, A' is the operation that would have been executed, instead of A, if operation B' had been already executed.

Transpose Property 3: If $Transpose(A, B) = (B', A')$ then $Transpose(B', A') = (A, B)$.

Transpose Property 4: For all operations A, $Transpose(A, I) = (I, A)$, where I is the identity operation.

Transpose Property 5: If $Transpose(A, B) = (B', A')$, then

$$RShift(RShift(C, A), B) = RShift(RShift(C, B'), A')$$

Property 1 allows reordering of operations in the history list and guarantees that the resulting state will be the same. Property 2 allows us to meaningfully undo A, leaving only the effect of B, by transposing them and executing $\overline{A'}$. Properties 3, 4, and 5, in a careful design, should follow from Property 2 (see [34]). However, we state them because Properties 3, 4, and 5 can be checked formally for a given definition of the Transpose function. Property 3 says that if A and B can be reordered into B' and A', it is reasonable to be able to reorder B' and A' into A and B. Property 4 asserts that reordering an operation with respect to a null (identity) operation should not result in the modification of the operation. Property 5 says that reordering two operations (A and B) using the Transpose function should not affect how an earlier operation (C) is reordered with respect to the combination of the two operations.

Note that Transpose Properties 1, 3, 4, and 5 are automatically satisfied for the special case when, for any two operations A and B, $Conflict(A, B)$ implies $Conflict(B, A)$, and $Transpose(A, B) = (B, A)$ whenever A does not conflict with B.

For group text-editors, such as those based on DistEdit, operations A' and B' are usually identical to operations A and B, except that the position data is different. For other applications, operations A' and B' are usually identical to operations A and B.

4.2.3 Inverse function

Let us denote $Inverse(X)$ by $\overline{X}$. We will assume that the Inverse function satisfies the following properties, where I is the identity operation:

Inverse Property 1: $X \circ \overline{X}$ is equivalent to I in terms of its effect on the state, i.e., $S \circ X \circ \overline{X} = S$.

Inverse Property 2: Reordering any operation, say A, with respect to $X \circ \overline{X}$ should give results equivalent to reordering the operation with respect to I (Transpose Property 4). More formally, if $Transpose(A, X) = (X', A')$, then $Transpose(A', \overline{X}) = (\overline{X'}, A)$.

Inverse Property 1 ensures that a sequence $X \circ \overline{X}$ is equivalent to the identity operation with respect to its effect on the state of the document. Property 2 ensures that $X \circ \overline{X}$ behaves like an identity operation when the two together are transposed with other operations. So, if given the sequence $A \circ B \circ \overline{B}$, A is reordered with respect to the sequence $B \circ \overline{B}$, we will get the result $B' \circ \overline{B'} \circ A$, where $Transpose(A, B) = (B', A')$. This property make the definition of Inverse consistent with Property 4 of the Transpose function.

The following definitions are used later in the paper:

Definition: A list of operations, L_1, is said to be **transpose-reducible** to an equivalent list of operations, L_2, if any of the following cases hold:

- L_2 can be obtained from L_1 by reordering two neighboring operations on L_1 using the Transpose function
- L_2 can be obtained from L_1 by eliminating an operation and its inverse when the two operations are next to each other
- L_1 is transpose-reducible to a list L_3 and L_3 is transpose-reducible to L_2

Definition: Given that a list of operations, L_1, is transpose-reducible to a list of operations, L_2, an operation A' on the list L_2 is said to **correspond** to an operation A on the list L_1 if A on the list L_1 becomes A' as a result of transpose-reducing L_1 to L_2.

It follows from Transpose Property 1 and Inverse Property 1 that if L_1 is transpose-reducible to L_2, then L_1 is equivalent to L_2, i.e., both lists result in the same final state if their operations are applied to the same initial document state.

4.3 Document model examples

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.

Positions are defined as the absolute position in the text, with the first character in the document having the position 1. Line breaks are represented by newline characters, and treated like any other characters. Other representations of position, such as line and column number, could also have been 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 as any other data structure. 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.

Next, we need to establish when two editing operations conflict and how to reorder them when they do not conflict. One possibility is to consider two editing operations to conflict if they affect the same or neighboring characters. This definition is based on the assumption that, for the purpose of reordering and undo, users view the insertion and deletion of a character as a modification of sequence relations with neighboring characters. So, if two operations affect the same sequence relation, they are considered to have a dependency between them and cannot be reordered. Furthermore, any reordering should be consistent with the above assumption. Under this assumption, following is the definition of the Transpose function for various combination of the above two operations:

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

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

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

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

The Conflict function is true wherever the Transpose function is undefined. In the definition of the Transpose function, notice the change in position arguments so that Transpose Property 1 is satisfied. It can be verified that Transpose Properties 3, 4, and 5, and both the Inverse properties are also satisfied. Transpose Property 2 cannot be formally verified, but is assumed to be satisfied, given that our function definitions are consistent with the above assumption of users' view of the editing operations.

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 is 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:* Conflict(A,B) is true if and only if A and B 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 graphical operations, unlike those in text editors, will not usually require parameter changes as long as they use absolute (x, y) coordinates rather than coordinates relative to positions of other objects. If relative positioning among objects is desired in a graphical editor, then additional operations, which use relative coordinates, should be provided so that they can be correctly transposed.

4.4 Discussion of the model

The notion of conflict in our model has similarities to the notion of conflict used in concurrency theory of database transactions [12, 30, 31, 37] in that we also define conflict in terms of the ability to reorder operations. One difference is that we allow for operations to be modified when reordering them; in current database theory, operations are not modified when they are reordered. Another difference is that the conflict model in databases is primarily used to enhance concurrency during transaction execution. We use the conflict model to provide safe undo of operations.

The concept of reordering or transforming operations has been also used in groupware systems to ensure consistency of replicated document state when operations are done locally first and then broadcast [10]. The transformations in [10] take two parallel operations, say A and B, and determine the operation to execute if one of them has already been executed. Our Transpose function, in contrast, transforms one ordering of operations into another, equivalent, ordering.

Note that our framework leaves the task of determining when two operations conflict, and how they should be transposed so that Transpose and Inverse properties are satisfied, up to the application designer. This task is, in general, non-trivial and, as we did in the text-editing example, requires making assumptions about user's expectations from selective undo [34]. To further illustrate the point, consider the situation where one user modifies the abstract of a paper and another user, after reading the abstract, carries out a change elsewhere in the document. The application designer has to determine whether to define a conflict between an operation in the first set of modifications and an operation in the second set of modifications. The answer to some extent depends on whether users expect to be able to undo the first set of modifications without being forced to undo the second set of modifications. Abowd and Dix [1] explored this issue and recommended the design principle that the system should not prevent users from undoing operations if they can achieve the same effect through normal editing. In this example, if a user issues commands to undo the modifications to the abstract, according to the principle, the system should undo those changes, since the user could have undone those changes through normal editing. As discussed in Section 8, our experience with using the framework in DistEdit supports the above principle.

5 Selective Undo Algorithms

This section presents two versions of our undo algorithm: a limited selective undo algorithm 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, based on the identity of the user or some other criterion.

If an operation A is undone, we assume that users want their document to go to a state that it would have gone to if operation A had never been performed, but all other operations had been performed. For example, suppose that on a document in state S, operations A and B are performed in sequence, and then A is undone. By Transpose Property 2, assuming $Transpose(A, B) = (B', A')$, if A had never been performed, the system would have performed operation B' in place

of B. Therefore, after undoing A, we want a selective undo algorithm to result in the document's state being $S \circ B'$. If the history list from A onward after undoing A is transpose-reducible, and thus equivalent, to B', we have achieved the desired undo effect.

More generally, if a set of operations are undone by using a selective undo algorithm, the resulting history list should be transpose-reducible to a list that would have resulted if none of the undone operations had been performed[2].

Furthermore, suppose a list L_1 is transpose-reducible to a list L_2. Then, undoing an operation A on L_1 should give the the same result as undoing the operation that corresponds to A on L_2. The full selective undo algorithm presented in this section meets these requirements.

One desirable feature in an undo algorithm is that it be possible to truncate the history list at any time without losing the ability to undo operations that remain on the history list. Most editors remove old operations from the history list, in order to keep the amount of memory consumed by the history list bounded. The algorithms presented below have this feature.

The algorithms presented in this section are independent of whether all sites in a group use a common, centralized history list or every site maintains its own copy of the history list. However, if multiple history lists are used, care must be taken to ensure that the history lists remain consistent with each other. For some tradeoffs in using a single history list versus using multiple history lists, see [33].

5.1 Limited selective undo algorithm

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 the operation that must be executed in the current state in order to cancel the effect of the selected operation, which was executed in an earlier state.

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

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

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

where $(C', A'') = Transpose(A', C)$. It follows from Transpose Property 2 that B' and C' are the operations that the system would have executed, instead of operations B and C, if operation A had not been executed earlier. Now that A has been shifted to the end of the list, $\overline{A''}$ can be performed giving the list:

$$B'\ C'$$

[2]From now on in the paper, when we talk about a list that would have resulted if none of the undone operations had been performed, we are implying that such a list must include all other operations, possibly in a modified form so as to be consistent with Transpose Property 2.

Note that performing $\overline{A''}$ in the present state correctly cancelled A, giving the document state, $S \circ B' \circ C'$, the state that would have resulted if operation A had never been performed; the undo has succeeded!

The list in the above form, containing modified operations, is not very useful for doing further undo operations. In particular, operations A and $\overline{A''}$ do not appear on the list above. It is, therefore, preferable to maintain a history list that contains the actual sequence of operations, as follows:

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

The above history list is transpose-reducible, and thus equivalent, to $B'C'$.

This algorithm, though based on the correct idea of shifting operations to the end of the list, can fail if some operations were undone earlier. For example, suppose that the history list contains $A\ B\ C$, where A conflicts with B but neither conflicts with C. A user, wanting to undo both A and B, first undoes B, resulting in a history list that contains $A\ B\ C\ \overline{B'}$. Then, the user attempts to undo A. The limited selective undo algorithm determines that A conflicts with B, and is unable to shift A to the end of the history. However, since B is undone, the user should have been able to undo A.

5.2 Selective undo algorithm

We now give a selective undo algorithm that is not limited by prior undo operations (Figure 1). The algorithm is similar to the limited selective undo algorithm, 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, with the oval line beneath the sequence indicating 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 that has been later undone (i.e. there is really no conflict), then that operation and its undo are removed from the copied history list by the procedure *RemoveDoUndoPair*.

The *RemoveDoUndoPair* procedure, given an operation X that is later undone by Y, shifts X until it is adjacent to Y and then removes both operations. This is valid because X' and Y must be inverses of each other, where X' is the operation that results from shifting X. X will not conflict with another operation Z in the history between it and Y, unless Z itself has been undone (otherwise, X could not have been undone in the first place). In the case of such an intervening Z, *RemoveDoUndoPair* is called recursively to first eliminate Z from the history list.

5.2.1 An example of selective undo

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

$$A\ B\ C\ D$$

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

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

```
type HistoryRec = record
   op: Operation;
   next: ^HistoryRec;
   undoneBy: ^HistoryRec;          /* for pairing do/undo */
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 Conflict(ShiftOp,HistPtr^.op) or (HistTemp^.undoneBy = HistPtr) 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 Conflict(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 */
   ListDelete(HistTemp, doPtr^.next); ListDelete(HistTemp, doPtr);
endproc
```

Figure 1: The Selective Undo Algorithm

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 B conflicts with C, and C has a *do-undo* pointer, *RemoveDoUndoPair* is called to remove the C and $\overline{C'}$ pair. The resulting (temporary) history list from B onward will be as follows:

$$B \; D'$$

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

Assuming that B does not conflict with D', B will be shifted past D' giving the operation B' where $(D'', B') = Transpose(B, D')$. Now that 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:

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

It can be verified that the above list is transpose-reducible, and thus equivalent, to AD'', where D'' is the operation that would have been executed, instead of D, if operations A and B had never been performed (this follows from Transpose Property 2). Thus, the undo has succeeded, giving the desired state.

5.3 Properties of the selective undo algorithm

It is important to understand the effect of our assumptions regarding the definitions of Transpose, Conflict, and Inverse functions on the behavior of undo when using the selective undo algorithm. Below, we state some properties of the algorithm that follow from the definitions of Conflict, Transpose, and Inverse functions and discuss the implications of those properties on the behavior of undo.

Lemma 1: If A does not conflict with B and $Transpose(A, B) = (B', A')$, then A' does not conflict with $\overline{B}$ and $Transpose(A', \overline{B}) = (\overline{B'}, A)$.

Proof: This is essentially a restatement of Inverse Property 2. By that property, if $Transpose(A, B) = (B', A')$, then $Transpose(A', \overline{B}) = (\overline{B'}, A)$. Since $Transpose(A', \overline{B})$ is defined, A' does not conflict with $\overline{B}$. □

Lemma 1 ensures that given non-conflicting operations A and B where B has been undone, A can be undone by shifting it past B and $\overline{B}$. Note that our selective undo algorithm calls *RemoveDoUndoPair* only when there is a conflict. It is therefore important that if A is shifted past B, it can also be shifted past $\overline{B}$. If this lemma didn't hold, the selective undo algorithm could fail to do even simple single-user history undo. Fortunately, the lemma is automatically satisfied when Transpose and Inverse functions are defined as required.

Lemma 2: If A does not conflict with B and $Transpose(A, B) = (B', A')$, then B does not conflict with $\overline{A'}$ and $Transpose(B, \overline{A'}) = (\overline{A}, B')$.

Proof: This lemma follows from Transpose Property 3 and Inverse Property 2. By Transpose Property 3, $Transpose(A, B) = (B', A')$ implies that $Transpose(B', A') = (A, B)$. Using Inverse Property 2, that in turn implies that $Transpose(B, \overline{A'}) = (\overline{A}, B')$. Since $Transpose(B, \overline{A'})$ is defined, B does not conflict with $\overline{A'}$. □

The importance of Lemma 2 is that operation B in the sequence $A \; B \; \overline{A'}$ can be undone by shifting it past $\overline{A'}$. This shifting, without taking into account the earlier shifting of A past B, is exactly what is done in the selective undo algorithm.

Lemma 3: If A does not conflict with B and $Transpose(A, B) = (B', A')$, then $\overline{B}$ does not conflict with $\overline{A}$ and $Transpose(\overline{B}, \overline{A}) = (\overline{A'}, \overline{B'})$.

Proof: Since $Transpose(A, B) = (B', A')$, by Inverse Property 2, it follows that $Transpose(A', \overline{B}) = (\overline{B'}, A)$. Using Lemma 2, this implies that $\overline{B}$ does not conflict with $\overline{A}$ and $Transpose(\overline{B}, \overline{A}) = (\overline{A'}, \overline{B'})$. □

Lemmas 2 and 3 collectively imply that, given the sequence $A\ B$, if operations A and B can be undone in either order, the results will be the same irrespective of the order in which they are undone. This is a crucial result for undo since we should get the same result irrespective of the order in which a set of operations are undone.

Lemma 4: If A does not conflict with B and $Transpose(A, B) = (B', A')$, then $\overline{A'}$ does not conflict with $\overline{B'}$.

Proof: Since $Transpose(A, B) = (B', A')$, it follows from Transpose Property 3 that $Transpose(B', A') = (A, B)$. Then, using Lemma 3, it follows that $\overline{A'}$ does not conflict with $\overline{B'}$. □

Note that if A does not conflict with B, then the sequence $A\ B$ can be undone by either by executing $\overline{B} \circ \overline{A}$ or by executing $\overline{A'} \circ \overline{B'}$. Lemmas 3 and 4 together imply that, in such a case, the two undo operations can also be undone in either order, since they are guaranteed not to conflict. If these lemmas didn't hold, users might encounter the strange behavior that they can undo two operations in either order but cannot later undo the inverse operations (redo operations) in either order.

Lemma 5: The algorithm will not undo an operation twice, unless the undo operation itself has been undone before the second undo.

Proof: Let's assume that operation A has been undone by an operation $\overline{A'}$, after possibly executing other operations. Suppose that a request is now made to undo A, after possibly executing additional operations. Clearly, there must exist a *do-undo* pointer from A to $\overline{A'}$. Furthermore, in order to undo A, the algorithm at some point must shift A past $\overline{A'}$. However, such a shift is not possible unless $\overline{A'}$ has also been undone, because an operation is never transposed with a conflicting operation or with an operation pointed to by its *do-undo* pointer (see the condition for the first if-statement in Figure 1). Thus the lemma follows.

Lemma 5 is important because a user could inadvertently select an operation to undo that has already been undone. So, given the history list $A\ \overline{A}$ with a *do-undo* pointer from A to $\overline{A}$, the lemma ensures that A will not be transposed with $\overline{A}$ and, thus, will not be undone unless $\overline{A}$ is undone first.

Lemma 6: Given an operation A such that no undo has been done after A, the undo of A results in a history list that can be transpose-reduced to the list that would have resulted if A had never been performed.

Proof: Let's assume that there are n operations after A on the history list. We will prove the lemma by induction on n. First, the lemma clearly holds if $n = 0$, i.e., when A is the last operation on the history list, since executing $\overline{A}$ results in a history list that can be transpose-reduced to one without A and its inverse.

Let the induction hypothesis be that the lemma holds for $n = i$. We need to show that it holds for $n = i + 1$. Let the the operations that are executed after A be $B_1, \ldots, B_i + 1$. Therefore, prior to the undo of A, the history list from A onward is as follows:

$$A\ B_1\ \cdots\ B_{i+1}$$

Since it is given that none of the operations $B_1, \ldots, B_{i+1}$ are a result of undo commands, the undo of A is done by shifting A past the following operations. If A is shifted past B_1 by using Transpose, we get the equivalent list:

$$B_1'\ A'\ B_2\ \cdots\ B_{i+1}$$

where $Transpose(A, B_1) = (B_1', A')$. From Transpose Property 2, it follows that undoing A in the original history list is equivalent to undoing A' in the above equivalent list. However, by induction hypothesis, undo of A' will result in a history list that can be transpose-reduced to one that would have resulted if A' had not been performed. Thus, the lemma holds for $i+1$. □

Lemma 7: Transpose-reducing the history list, prior to doing an undo using the selective undo algorithm, does not affect the operation that will be executed to effect the undo.

Proof: Let A_i be the operation to be undone on the history list $A_1, \ldots, A_i, \ldots, A_n$. The selective undo algorithm will attempt to shift A_i past $A_{i+1}, \ldots, A_n$ to determine the operation that will undo A_i. Consider each potential step, and its effect, of any other method that transpose-reduces the history list before shifting A_i forward:

- *Transposing two operations that occur before A_i:* This obviously has no effect on A_i or later operations, and thus on the operation that will result from shifting A_i past later operations.
- *Transposing A_{i-1} with A_i:* Suppose A_{i-1} and A_i are transposed, resulting in (A_i', A_{i-1}'). In that case, by Transpose Property 3, $Transpose(A_i', A_{i-1}') = (A_{i-1}, A_i)$. Thus, the result of shifting the resulting operation A_i' past $A_{i-1}', A_{i+1}, .., A_n$ will the same as the result of shifting A_i past $A_{i+1}, \ldots, A_n$.
- *Transposing A_i with A_{i+1}:* Suppose A_i and A_{i+1} are transposed, resulting in (A_{i+1}', A_i'). Clearly, the result of shifting A_i past $A_{i+1}, \ldots, A_n$ is the same as the result of shifting A_i' past $A_{i+2}, \ldots, A_n$.
- *Transposing two operations that occur after A_i:* Using Transpose Property 5, it can be easily shown by an inductive argument that this will not affect the operation that will result from shifting A_i to the end of the list.
- *Removing an operation and its inverse from the list when they are neighbors:* It is reasonable to assume that A_i cannot be the operation being removed, because then it cannot be shifted to the end of the list for the purpose of undo. By Inverse Property 2, an operation and its inverse together behave as I with respect to the Transpose function. Therefore, removal of an operation and its neighboring inverse will not affect the shifting of an earlier operation in the list. It obviously cannot affect the shifting of a later operation on the list. Thus such a removal will not affect the operation that will result from shifting A_i to the end of the list.

We have shown that the operation resulting from shifting A_i to the end of a list is the same as the operation that will result from shifting A_i (or A_i', if A_i was transposed) to the end of the list after applying one of the above steps. By induction on the number of steps, it easily follows that applying any number of above steps does not change the operation that will result from shifting A_i to the end of the list, and thus does not change the operation that will undo A_i. □

Lemma 7 is quite powerful. For instance, it implies that the result of undoing A in the sequence

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

will be the same as undoing A' from the following transpose-reduced, and thus equivalent, sequence:

$$A'\ \overline{D'}$$

where A' and D' are the operations that would have been executed on the document, instead of A and D, if operations B and C had never been performed.

Lemma 7 also implies that, in order to undo an operation A, we can disregard operations executed prior to A. In the above example, the lemma implies that we do not lose the ability to undo A, even if the history list is truncated to exclude all operations prior to A.

Theorem: The selective undo algorithm works correctly. That is, undoing operations always results in a history list that is transpose-reducible to a list that would have resulted if the undone operations had never been performed.

Proof: We will prove this by induction. Let there be n undo operations performed so far on the document. Lemma 6 implies that the theorem holds for $n = 1$. As induction hypothesis, assume that the theorem holds for $n = i$. We will now show that it holds for $n = i + 1$.

Let A be the operation that is undone on the $(i+1)^{st}$ undo. Consider the history list prior to this undo. By induction hypothesis, at this point, the history list is transpose-reducible to a list L that would have resulted if all the i undone operations and their inverses (except possibly for A and the operation it cancelled, if A resulted from an undo) had not been performed. Let operation A on the history list correspond to operation A' on the list L. From Lemma 7, it follows that the operation that will be used to undo A from the history list will be the same as the operation that will undo A' from the list L. Let the undo result in the execution of an operation U. Consider the list L appended with U. Clearly, on the list L, there can be no undo operations between A' and U. Therefore, by Lemma 6, the list $L\ U$ can be transpose-reduced to a list L_2 in which A' and U do not occur. This, however, implies that the history list appended with U can be transpose-reduced to L_2, by first transforming the original history list to L, and then transpose-reducing $L\ U$ to L_2. Thus the theorem follows. □

6 Performance Improvements in the Selective Undo Algorithm

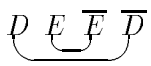
The worst-case time complexity of the above selective undo algorithm for undoing an arbitrary operation on the history list is $O(n^2)$, where n is the number of operations after the selected operation on the history list. This assumes that the operation to undo has already been selected and that reversing an operation and transposing two operations takes $O(1)$ time. The non-linear complexity is due to the call on the *RemoveDoUndoPair* procedure, which, to remove n nested do-undo pointers, takes $O(n^2)$ time. In this section, we discuss the situations in which the algorithm can be made more efficient.

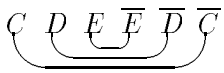
6.1 Efficient undo using undo blocks

Each history undo operation in a single-user editor, such as Emacs, can be executed in constant time, independent of the length of the history list; to undo an operation, the system simply needs to execute the operation's inverse and advance the pointer that identifies the next operation to

undo (see Section 2.3). It would be desirable to ensure that a group editor's undo is as efficient when the editor is used with only one user editing, as is often the case in practice [5, 29].

The selective undo algorithm, as described above, is $O(n^2)$ for single-user history undo. Our experience with the use of undo in the DistEdit-based Emacs editor indicates that occasionally n can become sufficiently large for the delay in the selective undo to become noticeable, particularly when compared to the history undo of single-user Emacs. Fortunately, however, it is easy to enhance the algorithm so that it takes constant time for single-user history undo. The basic idea is to introduce the notion of *undo blocks*. An undo block is a sequence of operations on the history list that is equivalent to an identity operation. Some examples of undo blocks are the following:





When only one user is editing the document, one ends up with undo blocks of the above kind (a sequence of operations followed by their inverses).

Undo blocks are easy to maintain during the selective undo algorithm. An *undo-block tag* can be associated with an operation to mark that everything between it and its corresponding undo operation is an undo block (for instance, D would be tagged in the first example above). When an operation, say C in the second example above, is undone, the operation is assigned an undo-block tag if either

- it is the last operation on the history list, or
- the operation immediately after it, say D in the above example, is also tagged, and its inverse, $\overline{D}$, is the last operation on the history list.

The key characteristic of an undo block is that it can be ignored as far as the undo of operations prior to the undo block is concerned. Therefore, when creating a temporary copy of the history list (see the first step of the algorithm in Figure 1) one can skip over any undo blocks. In particular, for the single-user situation, only the operation to be undone would exist on the temporary history list, leading to a constant time undo.

There is no reason why undo blocks should not be maintained even during group work. The space/time cost to maintain them is little compared to the potential improvement in response time as a result of not having to invoke *RemoveDoUndoPair* as often.

6.2 Pure transpose functions

Undo blocks help make the selective undo algorithm efficient in some common situations. However, the worst-case complexity of the algorithm remains $O(n^2)$ if few undo blocks occur, as could happen when several users are working simultaneously on the document. It turns out that it is possible to reduce the algorithm's worst-case complexity to $O(n)$ if the transpose function is *pure*, i.e., $Transpose(A, B) = (B, A)$ whenever $Conflict(A, B)$ is false.

In such a situation, it is possible to remove all the redundant do-undo pairs from the temporary history list in $O(n)$ time. This removal simply requires a traversal of the list to delete any operation that has been later undone, along with its inverse.

This sequential deletion of do-undo pairs without using the *RemoveDoUndoPair* procedure works because, with pure transpose functions, any shifting of A to bring it next to $\overline{A}$ would not modify the operations between A and $\overline{A}$.

Text editors are difficult to design in such a way that the transpose function is pure — most operations use character positions or line numbers to refer to entities in the text for efficiency reasons. Graphical editors, on the other hand, appear to be easier to design to use a pure transpose function and, thus, can take advantage of a more efficient implementation of selective undo.

6.3 More efficient algorithm?

Even with pure transpose functions, the algorithm is $O(n)$ primarily because it still has to traverse the history list to check if the operation to be undone conflicts with a later operation. If it were possible to check whether an operation conflicts with a later operation in less time, we would have a more efficient selective undo algorithm, assuming that the transpose function is pure.

Unfortunately, even with pure transpose functions, it appears difficult to develop a general, more efficient way of checking an operation for conflicts with later operations. To see the difficulty, consider the following history list:

$$X_1\ \overline{X_1}\ X_2\ \overline{X_2} \ldots\ X_{n-1}\ \overline{X_{n-1}}\ X_n$$

where all the X_i's conflict. In this situation, none of the operations except X_n are undoable. Now, when X_n is undone via $\overline{X_n}$, n of the operations, $\overline{X_1}, \overline{X_2}, \ldots, \overline{X_n}$, become undoable. It appears to us that an algorithm will take linear time either to mark all the previous nodes that become undoable when X_n is undone, or, in the absence of a marking, to determine if an operation such as $\overline{X_2}$ is undoable.

7 Selecting Operations to Undo

Before the selective undo algorithm can be used in practice, a means must be provided for a user to select the operation to be undone. There are many variations by which operations to be undone can be selected. We give some useful variations below to illustrate the basic techniques. All of these variations have been implemented in the DistEdit toolkit.

7.1 Per-user history undo

The Emacs-style history undo described in Section 2.3 can be made to work in our framework, allowing users to undo their own 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. When an operation other than another undo is performed, the stored pointer is deleted, making the undo operations appear as normal operations that can be later undone.

7.2 Region-undo

In region-undo, only operations that affect a specified region of a document are undone. For example, a user may want to undo changes to the abstract of a paper, but not undo any other changes. Region-undo can be a useful feature not only in group editors, but also in single-user editors.

Using a region as a selection criterion is slightly more difficult than using user-id or timestamps because operations stored on the history list may refer to locations where the region used to be, rather than where the region is now.

One way to locate operations that affects a region R is to define a special region-identifying operation S such that $Conflict(A, S)$ is true if operation A was performed in the region R. We

place S at the end of the history list, and use transpose to shift it backward. If a conflict arises, the conflicting operation must be within the region and can now be undone. For any operation A, $Transpose(A, S)$ should give (S', A), where S' identifies the region that corresponds to S prior to executing A.

Note that, to allow 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, in order to carry out subsequent undo operations. This apparent anomaly is not a problem since S is not an update operation — it is simply introduced to identify a region and to determine which operations were carried out in that region.

7.3 Multi-operation undo

Situations often arise in which an application may wish to treat a group of primitive operations as a single action for the purpose of undo. For instance, consider the following scenarios:

- One user-level action (e.g. *IndentParagraph*) could result in numerous primitive operations (a bunch of *InsChar* operations). Users would expect to be able to undo the user-level action in its 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 time-stamps after 5PM last Tuesday), assuming sufficient history with appropriate tags is kept.

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

- The history list is extended to keep sufficient information around so that the set of operations that constitute a multi-operation action can be correctly selected.

- When undoing a multi-operation action, all the primitive operations that constitute the action are shifted to the end and then undone collectively.

One open issue is whether a partial undo should be allowed if conflicts arise during shifting of some, but not all, of the operations that constitute a multi-operation action. The present version of DistEdit carries out partial undo of multi-operation actions upon a conflict, based on the assumption that users would prefer to undo as much of a complex change as possible than to undo none of it. An alternative would be require that either all the primitive operations are undone collectively or, if conflicts arise, none of them are undone.

8 Usage Experience

For implementing per-user undo in DistEdit, we first used conflict and transpose functions similar to those given in Section 4.3. Using these functions, if the characters affected by two operations were the same or were neighbors, the operations were assumed to conflict. Else, transpose functions similar to those in Section 4.3 were used to reorder the operations[3]. We present some of the problems that were encountered in using this version of the system, the lessons learned, and one solution to those problems. We also discuss whether providing a global undo facility is desirable in groupware systems.

[3]The functions used in DistEdit are slightly more general versions of the functions in Section 4.3 because DistEdit's primitive operations allow insertion and deletion of strings.

8.1 The problem of conflicts

Undo operations in DistEdit sometimes failed due to conflicts when users expected them to succeed. The following example illustrates the type of situation in which undesired conflicts arose:

A user, say Ann, deletes a sequence of characters, say $c_1 ... c_n$ one by one. Another user, say Bob, subsequently inserts some characters following c_n. Unfortunately, now, if Ann issues an undo command, it will fail. She cannot undo the deletion of c_n because Bob later did an operation next to c_n. To compound the problem, she also cannot undo the deletions of $c_1 ... c_{n-1}$ because each deletion conflicts with the next one! In most cases, the reason for such conflicts was not obvious to users because they expected the undo operations to succeed in recovering their lost text.

Conflicts sometimes also arose when the editor was being used by a single user. This happened when the region-undo command was invoked. The following is an example of the problem. Suppose that a user deletes a paragraph and subsequently adds some characters at the beginning of the next paragraph. Now, the user selects the region from which the paragraph was deleted, and invokes the region-undo command, expecting the deleted paragraph to be restored. Unfortunately, the region-undo operation would fail. The reason for the failure, of course, is because of a conflict with an operation that added the character immediately outside the selected region. Conflicts during single-user use were quite annoying because the editor's user expected *all* undo operations to succeed and with specific effect (e.g. reappearance of a paragraph, etc.).

One could attempt to avoid the above problem by defining Conflict to be false for neighboring operations. Below, we describe the resulting behavior obtained at the user level.

8.1.1 Inability to undo operations

Since attempts to undo *DelChar* operations were the ones that caused us to notice that there was a problem, we first tried out the seemingly obvious solution of defining the Conflict function to be false for neighboring deletes. A transpose function that satisfies Transpose Property 1 and appears to be quite reasonable for two neighboring deletes is as follows:

$$Transpose(DelChar(p, c_1), DelChar(p, c_2)) = (DelChar(p+1, c_2), DelChar(p, c_1))$$
$$Transpose(DelChar(p, c_1), DelChar(p-1, c_2)) = (DelChar(p-1, c_2), DelChar(p-1, c_1))$$

Unfortunately, the above change did not lead to desirable behavior in practice, as illustrated by the following example:

Example: Let's say the initial string in a text buffer is *abcd*. Characters *b* and *c* are deleted using separate operations. These two operations end up on the history list as *DelChar(2, 'b')* and *DelChar(2, 'c')*. Now, consider the problem of undoing these two operations, so as to get back the original string *abcd*. Clearly, these two operations can be potentially undone in two ways:

1. by undoing the second operation first and then the first operation, or
2. by undoing the first operation and then the second operation.

The first way should always work, since it is the same as doing single-user history undo. Either way, the result should be the original string *abcd* after performing the two undo operations.

Unfortunately, doing undo the first way fails if the Conflict function is defined to be false for neighboring deletes, but true for other neighboring operations. After the first undo, the history list becomes:

$$DelChar(2,'b')\ DelChar(2,'c')\ InsChar(2,'c')$$

However, now the second undo fails because *DelChar(2, 'b')* can be shifted past *DelChar(2, 'c')* (there is no conflict now for neighboring delete operations) but not past *InsChar(2, 'c')* (there is still a conflict between neighboring insert and delete operations). The above relaxation of the Conflict function alone is clearly not sufficient to ensure good undo behavior at the user level.

8.1.2 Incorrect results from undo

At this point, one might be tempted, as we were, to redefine the Conflict and Transpose functions so that none of the neighboring operations conflict. We tried the following definition for the Transpose function for reordering an insert operation and a delete operation at the same location:

$$Transpose(DelChar(p,c_1), InsChar(p,c_2)) = (InsChar(p,c_2), DelChar(p+1,c_1))$$

Definitions of other neighboring operations can be similarly defined.

Usage of DistEdit with above definitions added quickly showed flaws with the definitions. In particular, consider the two ways of doing undo in the example above. The first way gives the correct result, but with the second way, the first undo gives the result *acd* and the history list:

$$DelChar(2,'b')\ DelChar(2,'c')\ InsChar(2,'c')$$

Now for the second undo, the selective undo algorithm shifts the operation past the two following operations using the Transpose function (since there is no conflict). The operation after the two shifts is *DelChar(3, 'b')*. Executing the inverse of this operation results in the string *acbd*, an incorrect result.

To make the second way work, the definition above could have been defined as follows:

$$Transpose(DelChar(p,c_1), InsChar(p,c_2)) = (InsChar(p+1,c_2), DelChar(p,c_1))$$

It can be checked that, like the earlier definition, the above definition also satisfies Transpose Property 1. Unfortunately, as can be verified, now the first way of doing undo fails to work, giving the incorrect result *acbd*!

8.1.3 Result of the experience

Relaxing the Conflict function for neighboring operations as described above led to anomalies or incorrect results because, as can be verified, the resulting Transpose functions violated Transpose Property 4 or Inverse Property 2. Thus, the lemmas and the theorem given in Section 5.3 no longer held.

At the same time, not being able to undo an operation if a later operation was done next to it turned out to be too restrictive in practice, even though it always gave correct results whenever the undo succeeded, and it satisfied the lemmas and the theorem of Section 5.3.

From the above experience, we conclude the following:

- Relaxing the Conflict function so that it is false whenever possible is important. Otherwise, users tend to get frustrated when they are unable to undo something that they consider to be undoable.

- It is important to verify that any definition of Transpose and Inverse functions satisfies the transpose properties and the inverse properties. Otherwise, selective undo can behave contrary to users' expectations and, even worse, lead to incorrect results.

In the case of text editing, fortunately, there is a way to allow safe transpose of neighboring operations by storing additional data in the operations on the history list. The operations stored on the history list are modified to be as follows:

- *InsChar(locator, position, c)*
- *DelChar(locator, position, c)*

Locator is used to associate a unique point of insertion or deletion with every operation. Locators have the following properties:

- Given an operation $op(l, p, c)$, the position of the first character after the locator l is p. We say $pos(l)$ is p.
- Given two different locators, say l_1 and l_2, it is possible that $pos(l_1) = pos(l_2)$. In other words, the two locators identify different points between the same two characters.
- An ordering relation, `precedes`, exists between any two different locators. So one locator can always be considered to be before the other. Given two locators l_1 and l_2, the following are true:
  - $pos(l_1) < pos(l_2)$ implies l_1 `precedes` l_2.
  - l_1 `precedes` l_2 implies $pos(l_1) \leq pos(l_2)$.

A new locator is associated with every new operation to insert or delete a character. When adding a new operation to the history list, the locator for that operation needs to be ordered correctly with respect to other locators at that position. If it is a delete operation, the locator for the operation is considered to be immediately before the next character and after any previous locators at that position. If it is an insert operation, the locator is considered to be immediately after the previous character and before any other locators at that position.

The Inverse function can now be defined as follows:

$$Inverse(InsChar(l, p, c)) = DelChar(l, p, c)$$
$$Inverse(DelChar(l, p, c)) = InsChar(l, p, c)$$

Notice that the locator of the operation and its inverse are identical.

The Transpose function is defined as follows:

$$Transpose(DelChar(l_1, p_1, c_1), InsChar(l_2, p_2, c_2))$$
$$= \begin{cases} (InsChar(l_2, p_2 + 1, c_2), DelChar(l_1, p_1, c_1)) & \text{if } p_1 < p_2 \text{ or } (p_1 = p_2 \text{ and } l_1 \textbf{ precedes } l_2); \\ (InsChar(l_2, p_2, c_2), DelChar(l_1, p_1 + 1, c_1)) & \text{if } p_1 > p_2 \text{ or } (p_1 = p_2 \text{ and } l_2 \textbf{ precedes } l_1); \\ undefined & \text{if } l_1 \text{ is same as } l_2 \end{cases}$$

$$Transpose(InsChar(l_1, p_1, c_1), DelChar(l_2, p_2, c_2))$$
$$= \begin{cases} (DelChar(l_2, p_2 - 1, c_2), InsChar(l_1, p_1, c_1)) & \text{if } p_1 < p_2; \\ undefined, & \text{if } p_1 = p_2; \\ (DelChar(l_2, p_2, c_2), InsChar(l_1, p_1 - 1, c_1)) & \text{if } p_1 > p_2 \end{cases}$$

$$Transpose(DelChar(l_1, p_1, c_1), DelChar(l_2, p_2, c_2))$$
$$= \begin{cases} (DelChar(l_2, p_2 + 1, c_2), DelChar(l_1, p_1, c_1)) & \text{if } p_1 \leq p_2; \\ (DelChar(l_2, p_2, c_2), DelChar(l_1, p_1 - 1, c_1)) & \text{if } p_1 > p_2 \end{cases}$$

$$Transpose(InsChar(l_1, p_1, c_1), InsChar(l_2, p_2, c_2))$$
$$= \begin{cases} (InsChar(l_2, p_2 + 1, c_2), InsChar(l_1, p_1, c_1)) & \text{if } p_1 \leq p_2; \\ (InsChar(l_2, p_2, c_2), InsChar(l_1, p_1 + 1, c_1)) & \text{if } p_1 > p_2 \end{cases}$$

The Conflict function is true wherever the Transpose function is undefined.

Locators essentially help us to reorder a delete operation followed by an insert operation correctly when they have the same position. Without the locator, as seen in Section 8.1.2, there are two ways to reorder a delete operation followed by an insert operation, and both fail to satisfy the needed properties, giving incorrect results. With the use of locators, it can be verified that the various properties in Section 4 are satisfied and, therefore, undo behaves correctly and without the anomalies described in the previous subsections.

Methods to efficiently determine the ordering among locators are not the focus of this paper. But to show that the ordering can be determined, we give two simple methods here. One method is for the system to maintain a list of locators, along with their current positions in the buffer. The list is ordered so that a locator precedes all locators that appear after it in the list. The positions in the list are updated as insertions or deletions occur, so as to reflect the positions of locators in the current state[4]. This method is similar to that used for maintaining *markers* in Emacs [35] and positions of locks in DistEdit [22]. Another method to determine the ordering among locators is to do so by using the information on the history list, and only when needed. The history list contains all the information needed to determine the ordering between two locators. The second method will generally be more efficient than the first because the cost of updating a list on every insertion/deletion is avoided.

8.1.4 Dealing with Remaining Conflicts

For text, the use of locators eliminates most of the conflicts during undo. However, the possibility of conflict remains, for instance, when one user creates a string and another user modifies it. The possibility of failure of an undo command because of conflicts bring up an interesting issue. What should a user/system do when an undo command fails due to a conflict? We discuss two approaches to address the problem.

The first approach is for the system to determine all the conflicting operations (using a conflict list generation algorithm, such as the one described in [33]) and to give the user the option to undo the requested operation, along with all the conflicting operations. How should the user be presented with such an option is an unexplored user interface design issue. It probably would be desirable to show to the user the effect of undoing the conflicting operations, since some of those operations may have been executed by other users. One possible scheme would be to allow the user to undo/redo the conflicting operations several times while highlighting the affected regions in the document.

The second approach, the one currently adopted in DistEdit, is simply to tell the user about the conflict, ignore the operation for undo, and allow the user to go on to the next older operation in the history. This approach, though not as powerful as the first approach, has turned out to be acceptable in practice. For many usage patterns of group editors, users work on different parts

[4]In fact, if such a list is maintained, it is possible to rewrite the Transpose function so that it is pure.

of a document and thus are unlikely to make overlapping changes, so conflicts should be rare. Overlapping changes are much more likely to occur in closely-coupled editing; however in such cases, the use of global undo, region-undo, or normal editing can help in getting around conflicts. Of course, as concluded in Section 8.1.3, Conflict and Transpose functions needs to be carefully designed so that operations can be reordered in as many cases as possible.

8.2 Per-user undo and global undo

In DistEdit, to allow experimentation with various undo facilities, we provide support for both per-user undo and global undo. Per-user undo undoes the user's last action. Global undo undoes the last action irrespective of the owner of the action.

An important issue in the design of group undo facilities is whether global undo is useful. Our preliminary experience in using DistEdit indicates that global undo can often be confusing for users. Two key problems are:

- *Lack of predictability.* When a user invokes a global undo command, it is difficult for the user to predict what will be undone. A user does not normally know what the globally last action is. The globally last action may have been done on a part of the document that the user is not currently observing. We feel that predictability of effect is very important for any editing operation, including undo.

- *Definition of globally last action:* The notion of globally last action is not well defined. Suppose two users do two actions simultaneously in their editors. In this case, no particular action can be guaranteed to be globally last. In fact, the two operations could end up in different orders on different history list(s).

Some scenarios where we found global undo to be of some use and having predictable effect are:

- When the group as a whole wishes to go to a known earlier state of a document by discarding all recent changes, irrespective of who made them.

- When a group is working with synchronized views (effectively as one person) — everyone is looking at the same part of the document, changes are being made one at a time, and everyone is known to be looking at everyone's changes.

A reasonable way to handle such scenarios is to provide a special editing mode where a user acquires the undo rights of an entire group. DistEdit, for instance, provides a *lockstep* editing mode in which views of all participants in a group session are synchronized, and users have to acquire the floor in order to edit. In fact, it may be sufficient to provide a single undo button that does per-user undo for operations executed in the normal (non-lockstep) editing mode and does global undo for operations executed in the lockstep-style editing mode.

Figure 2 shows the user interface of DistEmacs, a DistEdit-based group editor. The DistEdit toolkit provides a status/control window that informs users about the state of the group session and provides functions specific to group-editing. From the control window, two types of undo are provided — per-user undo and region-undo. It is also possible to reset the undo pointer to the end of the history list and skip an operation for the purpose of undo. Global undo is also provided by the toolkit and, in the case of DistEmacs, can be invoked from the editing window using a text command.

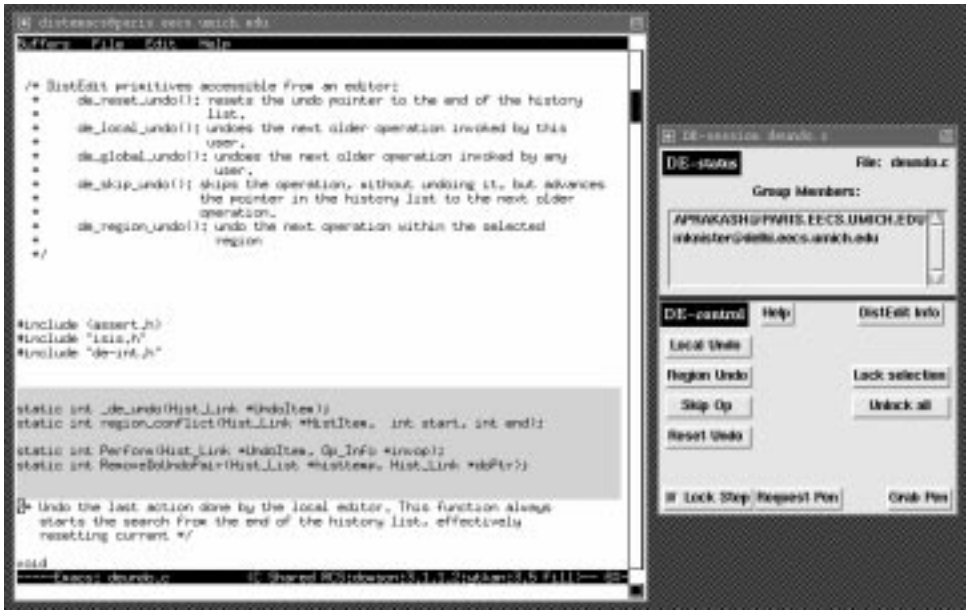


Figure 2: Sample screen display of a DistEdit-based group editor. The window on the right gives status information about the collaborative session and also provides access to various features of DistEdit, such as per-user undo and region-undo.

9 Conclusions and Future Work

We have presented a formal framework, and algorithms based on it, for allowing users to selectively undo operations on a document. The framework is quite general and is applicable to a variety of documents. The primary motivation for introducing the framework was to allow users of a groupware system to individually reverse their own changes. However, the framework can also be applied to single-user systems for implementing additional undo facilities, such as region-undo. We presented our framework in the context of history undo; however, many aspects of the framework, such as the notions of *Transpose* and *Conflict*, are also applicable to implementing undo based on the linear and the US&R models.

The main requirement to use the framework for implementing selective undo is for the application designer to provide functions to reverse and reorder operations. We stated properties that should be satisfied by such functions in order to provide undo behavior to the users that gives correct results and is free from certain potential anomalies.

We reported our preliminary experience in using undo facilities based on the framework in an actual system. Our experience indicates that functions to reorder operations must be designed carefully. A design that violates the required formal properties is likely to lead to unexpected or incorrect results from undo. It is also important that reordering conflicts are minimized so that users do not get frustrated in attempting to undo operations. More systematic studies will be useful, particularly for determining good user interfaces for dealing with conflicts.

History lists have previously been applied to uses other than undo, such as to see a trace of the evolution of text [8]. The mechanisms for transposing operations provided in the selective undo framework could be useful for providing a *selective* evolution of the text, for instance seeing the evolution history of a particular section of the document. In the future, we plan to explore other applications of the framework.

Acknowledgements

We would like to thank the referees and the editor, Jonathan Grudin, for many helpful comments on an earlier version of this paper.

REFERENCES

[1] Gregory D. Abowd and Alan J. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.

[2] S.R. Ahuja, J.R. Ensor, D.N. Horn, and S.E. Lucco. The Rapport Multimedia Conferencing System: A Software Overview. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 52–58, March 1988.

[3] V. Anupam and C. Bajaj. Shastra: Multimedia collaborative design environment. *IEEE Multimedia*, 1(2):39–49, Summer 1994.

[4] J.E. Archer and R. Conway. COPE: A cooperative programming environment. Technical Report TR-81-459, Cornell University, June 1981.

[5] R.M. Baecker, D. Nastos, I.R. Posner, and K.L. Mawby. The user-centered iterative design of collaborative software. In *INTERCHI'93 Conference Proceedings*, pages 399–405. Addison-Wesley, 1993.

[6] T. Berlage and A. Genau. A framework for shared applications with a replicated architecture. In *Proc. of the ACM Symposium on User Interface Software and Technology*, 1993.

[7] R. Chaudhary and P. Dewan. Multi-user undo/redo. Technical Report Technical Report TR125P, Computer Science Department, Purdue University, 1992.

[8] W.D. Elliott, W.A. Potas, and A. Van Dam. Computer assisted tracing of text evolution. In Proceedings of AFIPS Fall Joint Computer Conference, pages 533–540, 1971.

[9] C. Ellis, S.J. Gibbs, and G. Rein. Design and use of a group editor. In G. Cockton, editor, *Engineering for Human-Computer Interaction*, pages 13–25. North-Holland, Amsterdam, September 1988.

[10] C. Ellis, S.J. Gibbs, and G. Rein. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD '89 Conference on Management of Data*, pages 399–407. ACM Press, 1989.

[11] C.A. Ellis, S.J. Gibbs, and G.L. Rein. Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):38–58, January 1991.

[12] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notions of consistency and predicate locks in a relational database system. *Communications of the ACM*, 8(11):624–633, 1976.

[13] R. Fish, R. Kraut, M. Leland, and M. Cohen. Quilt: A collaborative tool for cooperative writing. In *Proceedings of ACM SIGOIS Conference*, pages 30–37, 1988.

[14] J.D. Foley and V.L. Wallace. The art of natural graphical man-machine conversion. *Proceedings of the IEEE*, 62(4):4622–471, April 1974.

[15] R.F. Gordon, G.B. Leeman, and C.H. Lewis. Concepts and implications of undo for interactive recovery. In *Proceedings of the 1985 ACM Annual Conference*, pages 150–157. ACM Press, New York, 1985.

[16] J.N. Gray. *Notes on Database Operating Systems*, pages 394–481. Springer-Verlag, 1978.

[17] I. Grief, R. Seliger, and W. Weihl. Atomic data abstractions in a distributed collaborative editing system. In *Proc. of the 13th Annual Symposium on Principles of Programming Languages*, pages 160–172, 1976.

[18] M. Hammer, R. Ilson, T. Anderson, E. Gilbert, M. Good, B. Niamir, L. Rosenstein, and S. Schoichet. The implementation of Etude, an integrated and interactive document production system. In *Proceedings of the ACM SIGPLAN/SIGOA Conference on Text Manipulation*, pages 137–146. ACM, New York, June 1981.

[19] W.J. Hansen. User engineering principles for interactive systems. In *AFIPS Conference Processings*, volume 39, pages 523–532. AFIPS Press, 1971.

[20] A. Karsenty and M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 195–202. IEEE Press, 1993.

[21] M. Knister and A. Prakash. DistEdit: A distributed toolkit for supporting multiple group editors. In *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, pages 343–355, Los Angeles, California, October 1990.

[22] M. Knister and A. Prakash. Issues in the design of a toolkit for supporting multiple group editors. *Computing Systems – The Journal of the Usenix Association*, 6(2):135–166, Spring 1993.

[23] B.W. Lampson. *Bravo Manual. In* Alto User's Handbook. Xerox Palo Alto Research Center, 1978.

[24] C. Linxi and A.N. Habermann. A history mechanism and undo/redo/reuse support in ALOE. Technical Report Technical Report CMU-CS-86-148, CS Department, Carnegie-Mellon University, 1986.

[25] M. Mantei. Capturing the capture lab concepts: A case study in the design of computer supported meeting environments. In *Proc. of the Conference on Computer-Supported Cooperative Work*, pages 257–270, 1988.

[26] L. McGuffin and G. M. Olson. ShrEdit: A shared electronic workspace. Technical Report CSMIL Technical Report No. 45, The University of Michigan, Ann Arbor, 1992.

[27] C.M. Neuwirth, D.S. Kaufer, R. Chandhok, and J.H. Morris. Issues in the design of computer support for co-authoring and commenting. In *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, pages 183–195, Los Angeles, California, October 1990.

[28] R.E. Newman-Wolfe and H. K. Pelimuhandiram. MACE: A fine-grained concurrent editor. In *Proceedings of the ACM/IEEE Conference on Organizational Computing Systems (COCS 91)*, pages 240–254, Atlanta, Georgia, November 1991.

[29] J.S. Olson, G.M. Olson, M. Storrøsten, and M. Carter. Groupware close up: A comparison of the group design process with and without a simple group editor. *ACM Transactions on Information Systems*, 11(4):321–348, October 1993.

[30] C.H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[31] C.H. Papadimitriou. *Database Concurrency Control.* Computer Science Press, 1986.

[32] A. Prakash and M. Knister. Undoing actions in collaborative work. In *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work*, pages 273–280, Toronto, Canada, October 1992.

[33] A. Prakash and M. Knister. Undoing actions in collaborative work. Technical Report CSE-TR-125-92, CSE Division, Department of EECS, The University of Michigan, Ann Arbor, March 1992.

[34] A. Prakash and M. Knister. Undoing actions in collaborative work: Framework and experience. Technical Report CSE-TR-196-94, CSE Division, Department of EECS, The University of Michigan, Ann Arbor, March 1994.

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

[36] M. Stefik, G. Foster, D.G. Bobrow, K. Kahn, S. Lanning, and L. Suchman. Beyond the Chalkboard: Computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30(1):32–47, Jan. 1987.

[37] R.E. Sterns, P.M. Lewis II, and D.J. Rosenkrantz. Concurrency controls for database systems. In *Proceedings of the 17th IEEE Symposium on Foundations of Computer Science*, pages 19–32, 1976.

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

[39] H. Thimbleby. *User Interface Design*, pages 261–286. ACM Press, New York, 1990.

[40] J.S. Vitter. US&R: A new framework for redoing. *IEEE Software*, pages 39–52, October 1984.

[41] Y. Yang. A new conceptual model for interactive user recovery and and command reuse facilities. In *Proceedings of the CHI'88 Conference on Human Factors in Computing Systems*, pages 165–170. ACM Press, May 1988.