

Undoing Actions in Collaborative Work: Framework and Experience

Technical Report CSE-TR-196-94

Atul Prakash
Michael J. Knister

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

Abstract

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

Keywords

Undo, collaboration, groupware, user recovery

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 [20], and reducing user frustration with new systems [15], especially those that allow users to invoke commands that can modify system state in complex ways. The availability of undo can also encourage users to experiment, acting not only as a safety net, but also allowing users to try out different approaches to solving problems using backtracking [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 [1, 11, 12, 36]. An undo facility can be important in groupware systems for several

reasons. Many groupware systems, such as group editors, can also be used in single-user mode and therefore, to be equally user-friendly as similar single-user systems, such systems should provide similar, if not more powerful, undo facilities. 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 whiteboard 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 help free group members to demonstrate ideas in the document.

Compared to single-user applications, performing undo in groupware applications provides technical challenges in following areas: *selecting* the action to be undone, determining *what* operation will result in a correct undo, dealing with *dependencies* between different users' operations, and providing a *reasonable, predictable* behavior to its users. First, in a group environment, there may be parallel streams of activities from different users. When parallel work on a shared document is going on, users would often expect to undo their own last operation rather than globally-last operation, which might belong to other users. An undo framework for groupware systems needs to be able to select the operation that a user intends to undo rather than simply selecting the globally-last operation in the system. 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. Third, if two or more users interleave their work in the same portion of a document, it may not make sense to undo one user's changes without undoing some of the other users' changes. In this case, there are dependencies between the changes made and they need to be taken into account during an undo. Finally, any undo scheme has to ensure that the behavior of undo is consistent with the users' view of the document. In particular, one user may not be aware of all actions done by other users. The behavior of undo should be consistent with a user's awareness of actions done on the document.

Many groupware applications have been built that support multi-user work on a shared document, e.g, Grove [9], ShrEdit[25], CES [18], and MACE [28]. Almost none, as far as we are aware, provide an undo facility that addresses all the above issues. Those applications that do provide an undo usually only provide a global undo facility rather than a per-user undo facility. MACE [28] does provide a simple form of per-user undo facility, allowing a user to undo only those modifications that he made by explicitly locking modified sections of the document, and only if he hasn't released the locks since the modifications. If the lock was released or if the modified section weren't explicitly locked prior to modifications, changes cannot be undone using the scheme.

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 successfully incorporated in DistEdit [22], a group editor toolkit, and a version of of SASSE [26, 4], 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 provide reasonable, predictable undo behavior to the users. Second, it includes a comparison with various group undo techniques that have been suggested recently and discusses the conditions under which their use may be appropriate. Third, it addresses design issues that occur in implementing group undo. In particular, we discuss the issue of what undo facilities are important at the interface level — those that allow only users to undo only their own actions, or also those that also allow them to undo other users' actions. Finally, it includes our experience in implementing and using the proposed undo facilities in the DistEdit toolkit [22].

Several groupware applications support *asynchronous* sharing, e.g., Quilt [14] and Prep [27], where group members work on a shared document at different times. The proposed undo techniques can be applied to these applications as well — a user can make tentative changes to a section of a document, knowing that they can be undone at a later time if needed, even if other users also subsequently modify the document. The proposed undo techniques can also be applied to single-user editors for undoing changes selectively, for instance those restricted to a particular region of a document.

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 needs in order to use our undo framework. Section 5 describes our selective undo algorithm and proves several useful properties of the algorithm. Section 6 suggests ways of improving the performance of the algorithm for many common situations. Section 7 discusses the design and implementation issues we faced in incorporating the undo scheme in the group text editor toolkit, DistEdit [21, 22]. In the same section, based on our experience with DistEdit’s undo facilities, we also discuss some important user-interface design issues, for instance the utility of providing both a per-user undo and a global undo in a groupware system. Finally, Section 8 summarizes our conclusions and future work.

2 Related Work

There are several basic methods for providing undo capability in systems, most of them designed for single-user systems. Almost all of them require maintenance of 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 same order as they were performed. For instance, if the history list is

$$A B C D$$

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

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 \bar{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 [16]. For instance, on a text document, if a *DelChar(10)* operation is done, which deletes the character at position 10, then in order to determine its inverse, we must know the character that was deleted. The operations stored in the history list should contain sufficient data so that their inverse can be easily determined. For instance, the above operation might be stored as *DelChar(10, c)* on the history list, where c is the deleted character.

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 a 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 [19], provided single-step undo. Single-step undo is common in more recent systems including many Macintosh and Windows applications as well as editors such as *vi*. Single-step undo allows undo of only the last operation. For instance, given the history list

$$A B C D E$$

it 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 [3], and Aloe [24] are some of the systems that used the linear undo model. The linear undo model allows undoing of a sequence of operations and keeps a pointer which tracks the next operation to be undone. Operations can then be redone, after possibly doing some new operations. For instance, given the history list:

$$A B C D E,$$

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

$$A B C F D E$$

 ↑

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

The Undo, Skip, Redo (US&R) model [40] supports redo like the linear undo model, but also allows a user-friendly skipping of some operations during the redo. Instead of a linear list, US&R model keeps a tree data structure for maintaining history so that it becomes possible to restore state to any point in the history (unlike the linear undo model). In the above example, F would be stored on a different branch of the tree from the sequence $D E$ so that F could be undone and then D and E could be redone if the user so desired.

A limitation of both the linear undo model and the US&R model is that in order to undo one operation X several steps back in the history, all subsequent operations must first be undone and then redone (skipping X during the redo). If the implementation is not careful, this can be potentially disruptive in a group environment; other users may see their work undone for at least a short while with no apparent reason. Furthermore, the models do not address the issue that simply re-doing operations may not semantically make sense or may lead to 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 US&R schemes, it appends the undo operations to the end of the history list. The undo operations in the history list are treated as any other operations, allowing them to be undone later if desired. For instance, given the history list,

$$A B C D E$$

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

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

 ↑

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

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

 ↑

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

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

 ↑

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

2.4 Checkpoint and Recovery Schemes

Checkpoint and recovery is a common strategy for dealing with failures and abort of transactions in databases [17]. Such schemes are usually not used for implementing undo in editor-type applications because it is usually less expensive to execute the inverse of operations to get back to an earlier state than to maintain a number of checkpoints and do forward execution to get to the current state. Databases typically cannot take advantage of reverse execution, and instead use checkpoint and recovery schemes, because changes to data due to a transaction must be invisible to other transactions until the transaction is committed.

2.5 Group-undo Schemes

There has been substantial interest in undo in groupware systems recently. Independently and about the same time as our work [32, 33], undo schemes for collaborative systems were proposed by Berlage [5], Chaudhary and Dewan [6], and Abowd and Dix [2]. We provide a comparison with these schemes below.

Berlage [5] and Chaudhary and Dewan [6] both recommend undoing any operation on the history list by simply executing its inverse provided the inverse is executable in the current state. The approach essentially assumes that any operation in the history list can be undone simply by executing its inverse in the current state, irrespective of later operations on the history list. Unfortunately, as we discuss in Section 6.4, not taking into account the dependencies among operations can lead to unexpected or hard-to-predict undo behavior in many situations. The framework and algorithms described in this paper are more general and take into account the possibilities of conflicts and dependencies among operations. In Section 6.4, we elaborate, in terms of the framework presented in this paper, on the semantics of the above approach and some situations in which it may be appropriate to use.

Abowd and Dix [2] recognize the need for dealing with dependencies among users' operations and suggest a basic framework similar to that suggested 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. Our focus has been on providing algorithms and implementation guidelines for group undo schemes. Furthermore, for defining dependencies between operations, we provide a formal definition of dependency among operations in terms of a *conceptual model* — a model of the users' view of a generic document and operations on it. Use of a formal definition helps facilitate a formal proof of correctness of undo algorithms, and helps predict their behavior 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' actions. In our experience, history undo is simple and intuitive for most users. However, if desired, the techniques given in the paper can also be applied to linear and US&R models.

We use data structures similar to those used in history undo; in particular, upon an undo, the inverse of an operation is appended at the end of the history list. However, in a groupware

application, since the last operation done by a user may not be *globally* last (other users may have done operations subsequently), we need to allow undoing of a *particular user's* last operation from the history list. For example, consider the following history list, where A_i 's refer to operations done by user A , and B_i 's refer to operations done by other users:

$A_1 B_1 A_2 B_2 B_3$

Now, suppose user A wishes to undo his last action, A_2 . Normal history undo mechanisms in single-user systems do not support such a task because they would require undoing B_2 and B_3 as well. In the *US&R* model, it is possible to undo the last three operations and then redo B_2 and B_3 , but as pointed out in the previous section, that can be disconcerting to other users of the system and may not even be correct if there are dependencies between A_2 and B_2, B_3 . Note that user A may not be aware that operations B_2 and B_3 have been carried out on the document by other users, and the other users may not be aware of activities of user A . 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 or time of operation. We term our scheme as *selective undo*, since the operation to be undone is not necessarily the last one, but is selected using some attribute attached to the operation.

To selectively undo an operation, we cannot simply execute the inverse of the operation because later operations could have shifted the location 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'), resulting in the state 'yabcd'. The first operation inserted 'x' at position 4, and the second operation inserted 'y' at position 1. Assume that these operations were done by different users. Now the user who did the first operation does an undo. We cannot simply perform the first operation's inverse, *DelChar*(4), because the second operation has moved the 'x' to location 5. Our scheme takes this possibility of location shifting into account, so that in this example, the first operation will be undone by executing *DelChar*(5).

We also take into account the possibility of *dependencies* or *conflicts*. In the above example, B_2 may have modified the same region of the document as A_2 ; so semantically it may no longer make sense to undo A_2 without first undoing B_2 . Our framework prevents an operation from being undone (erroneously) when later dependent operations exist that have not been undone.

In any undo scheme, it is important that undo behave according to users' expectations. In our framework, we distinguish between the *physical model* of a document and the *conceptual model* of a document. The physical model of a document is the view seen by the system and is designed for efficiency. In contrast, the conceptual model of a document is intended to closely model the users' view of a document. As long as the conceptual model of a document is properly defined to match users' view of a document, the undo framework presented in this paper ensures that the proposed selective undo algorithm has predictable and reasonable behavior at the user level.

4 Document Model and Application Requirements

4.1 Operations and State Equivalence

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

$$S \circ M \circ N$$

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

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

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

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

4.2 Physical Model and Conceptual Model of a Document

In a group environment, the notion of state equivalence is insufficient to ensure that undo of an arbitrary operation on the history list has predictable and reasonable behavior for the users. What is needed is a model of the users' view of a document, which we call the *conceptual model*. For the conceptual model to be useful for a large class of documents, it needs to be fairly simple and general, something which most users can relate to. The notion of dependencies between operations can then be defined in an implementable way in terms of the conceptual model.

In this paper, we assume that the conceptual model of a document consists of a set of attributed *objects* with *relations* between them. The conceptual model is based on the idea of the entity-relationship model used in modeling databases [7]. The *physical model* of a document, in contrast, is the actual data structure used to represent the document. An operation on a document in the physical model modifies the data structure used to represent the document. The same operation in terms of the conceptual model has the effect of *reading* or *writing* (deleting/creating/modifying) objects, their attributes, or the relations between them. More specifically, we require that a physical-model operation A done in state S on a document can be *mapped* to conceptual-model operations $C_{(S,A)}$ on the document. We will think of $C_{(S,A)}$ as being the *semantics of operation A when executed in state S* , or simply as the *semantics of A* when the state is obvious from the context.

As an example, a plain text document could be viewed, in terms of the conceptual model, as consisting of objects (characters in the document) and relations (**precedes** relationship between neighboring characters). The notion of position may not appear in the conceptual model if users, when editing, are expected to be interested in modifying the ordering of characters, rather than their specific positions in the document.

Following the work on concurrency control in databases [13, 31, 31, 37], one can define the notion of *read-set* and *write-set* for an operation in terms of its effect on the conceptual-model state of the document. The read-set of an operation done in a particular state is the set of objects/attributes/relations read by the operation. The write-set is the set of objects/attributes/relations deleted, created, or modified by the operation. A *read-write dependency* between two operations done in their respective states exists if their read set and write set respectively overlap. The no-

Table 1: Comparison of Conceptual and Physical Models

Physical model	Conceptual model
Defined by the system to ease implementation	Abstract view that represents users' view of a <i>generic</i> document
Operations are efficient to execute	Operations can be very inefficient to execute
usually efficient representation	usually inefficient to represent directly
Independent operations may not commute	Independent operations commute
Operations seen by the system and stored on the history list for the purpose of undo	Conceptual view of the operations is not directly seen by the system.

tions of *write-read* and *write-write* dependencies can similarly be defined. If there is no write-read, read-write, or write-write dependency between the operations, the operations can be considered to be *independent* or *non-conflicting*.

We require that independent operations in the conceptual model *commute*, i.e., reordering them should result in the same document state. Commutativity of independent operations does not necessarily hold in the physical view. For instance, consider the example discussed in Section 4. Two users see a string *abcd*. The first user inserts a 'y' before 'a' and another user inserts a 'x' before the character 'd', resulting in the string *yabcd*. In terms of the physical model, these operations could be carried out as *InsChar(4, 'x')* followed by *InsChar(1, 'y')*, giving the string *yabcd*. Clearly, these operations are not commutative: executing them in the opposite order would give a different string, *yabxcd*. In terms of the object-based conceptual model, the first operation adds a new object for character *y* and adds a `precedes(y, a)` relationship; the second operation adds an object for character *x*, deletes a `precedes(c, d)` relationship, and adds `precedes(c, x)` and `precedes(x, d)` relationships. Clearly, these operations commute.

A comparison of physical model and the conceptual model of a document is given in Table 4.2.

In this paper, we assume that the operations stored on the history list are those as seen by the system — i.e., as defined by the physical model. If the system were to store and work with the operations according to the conceptual model, it would probably be very inefficient — physical model is generally chosen in implementations for efficiency reasons. Of course, if the conceptual model of the document and operations on it can be implemented efficiently, one can design the system to simply just use the conceptual model — in that case, conceptual model and the physical model coincide for the system.

4.3 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 would usually be needed anyway to ensure predictable results when parallel streams of activities are going on. For instance, if two users are working simultaneously in a document, dependence checking may involve making sure that their changes do not overlap, e.g., through use of locks. Mechanisms for reordering of parallel, independent operations are also needed because the order in

which two operations will be done may be unpredictable. The editor must be prepared to accept the two operations in either order with the same resulting effect.

The functions which the application must provide are:

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

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

4.3.1 Inverse Function

We will assume that the Inverse function satisfies the following properties:

Inverse Property 1: $A \circ \bar{A} \equiv I$.

Inverse Property 2: \bar{A} has the same read-write set as A (more precisely, $C_{(S,A)}$ and $C_{(S \circ A, \bar{A})}$ modify the same objects/relations/attributes).

Property 1 relates operations A and \bar{A} in terms of their effect on the state of the document whereas Property 2 relates them in terms of the conceptual model. The above two properties are useful to ensure several properties (described in Section 5.3) that guarantee reasonable behavior of undo in a group environment.

4.3.2 Conflict Function

Operations on a document can have dependencies among them. Suppose, for example, that a graphics document is being edited. Operation A creates a circle in the document, and operation B resizes that circle. In this case, there is a write-write dependency between A and B — operation A created an object and B modified it. If operation A had not been done, operation B would make little sense.

Using the conceptual model of a document, notion of conflict or dependency between operations can be defined. Let's say that operations A and B are done in sequence in some arbitrary state S . $Conflict(A, B)$ function supplied by the application must return *True* if there is a potential write-read, write-write, or read-write dependency between operations A and B (more precisely, between conceptual versions of A and B — $C_{(S,A)}$ and $C_{(S \circ A, B)}$). It should return *False* if no such dependency exists.

The importance of the notion of conflict is that it imposes an ordering on operations A and B — the order of operations A and B cannot, in general, be interchanged without affecting the results. Furthermore, if $Conflict(A, B)$ is true, then, in general, operation A cannot be undone, unless the following operation B is undone too.

Note that 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 capture semantic dependencies between operations. It does not indicate in any way whether the users are cooperative or non-cooperative.

In the discussion that follows, we will often say that operations A and B *conflict*, or 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.

4.3.3 Transpose Function

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

The $Transpose(A, B)$ function, given two independent operations A and B, will return two new operations B' and A' , which satisfy the following property:

Transpose Property: Operation B' executed in an arbitrary state S should map to the same conceptual-model operation as operation B executed in state $S \circ A$, i.e., $C_{(S, B')} = C_{(S \circ A, B)}$. In other words, B' is the operation that should have been done by the system to the document instead of B if operation A had not been done before B . Analogously, operation A' executed in state $S \circ B'$ should map to the same conceptual-model operation as operation A executed in state S .

The Transpose function allows us to undo A , leaving only the effects of B , by transposing them and then undoing A' . Usually, operation A will be identical to A' , and B to B' , except that the position data may be different.

Corollary 1: It follows from the Transpose Property that operation B' executed in state $S \circ A$ has exactly the same read-write set as that of operation B executed in state S and operation A' executed in state $S \circ B'$ has the same read-write set as that of operation A executed in state S .

Corollary 2: Another corollary of the transpose property is that $S \circ B' \circ A' \equiv S \circ A \circ B$ for an arbitrary state S .

4.4 Relation to the work in Databases

Our conceptual model of a document and the operations on it is similar to the *read-write* model of transactions in databases [17, 31]. Our notion of conflict is similar to the notion of conflict in the concurrency theory of database transactions [13, 30, 31, 37]. One difference is that we allow for a Transpose function that modifies the operations when interchanging them; in current database theory, operations are not modified when they are reordered. As we will see, that complicates the algorithms for undo substantially.

As in the database work on concurrency control [34], we could have defined more elaborate conceptual models that distinguish between different types of write operations, thereby reducing conflicts. For instance, in a graphical editor, two editing commands to double and triple the radius of a circle will have a conflict in terms of the read-write model, but may not in terms of a more elaborate model that takes into account the semantics of the write operation, since the operations are commutative. More elaborated conceptual models could be used, if desired, without substantially affecting the results in this paper.

4.5 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 as 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 any other way. The application is responsible for correctly applying operations so that its internal data structure represents the correct state.

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

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

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

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

$$Conflict(InsChar(p_1, c_1), DelChar(p_2, c_2)) = \begin{cases} true, & \text{if } p_2 - 1 \leq p_1 \leq p_2 + 1; \\ false, & \text{otherwise.} \end{cases}$$

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

The definition for $Conflict$ says that there is a conflict if the character deleted is the same as or is next to the character that was inserted. Otherwise, the two operations are considered to be independent and transposable. This definition is based on the conceptual model of a text document discussed in Section 4.2. An $InsChar$ operation creates an object, deletes an old **precedes** relation, and adds two new **precedes** relations. A $DelChar$ operation deletes an object, deletes the **precedes** relations of the object with its neighbors, and adds a **precedes** relation between the object's neighbors. There will be an overlap in objects/relations affected if $DelChar()$ operation occurs within one position of the prior $InsChar()$ operation.

In the definition of $Transpose()$, notice the change in position argument in $Transpose()$ so that the Transpose Property is satisfied. The $Transpose()$ function above is similar to the T_{opt} matrix defined in [10].

We leave it to the reader to determine the $Conflict$ and $Transpose$ definitions for the other three combinations of these two operations. A definition of these two functions for general string insert and delete operations can be found in [33].

Example 2: Document Model Applied to Graphics Editors

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

- $DrawCircle(x, y, radius, CircleID)$: Draw a circle at position (x,y) of the specified radius. $CircleID$ is the object identifier returned by the command and stored in the history list to permit easy reversal and transpose.

- *ChangeRadius(CircleID, NewRadius, OldRadius)*: Change the radius of the circle *CircleID* to *NewRadius*. *OldRadius* is stored so that inverse is easy to compute.

In this case, assuming that the conceptual model of a generic graphical document is a set of objects with attributes indicating their absolute positions, the *Conflict* and *Transpose* functions are straightforward:

- *Conflict*: $\text{Conflict}(A,B)$ is true if and only if they refer to the same circle, i.e., their *CircleID*'s match.
- *Transpose*: Transposing the two operations simply requires interchanging the two operations if they refer to different circles; else the *Transpose* is undefined.

Note that the graphical operations, unlike those in text editors, usually will not require parameter changes 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.

5 Undo Algorithms

This section presents two versions of our undo algorithm: a limited selective undo to demonstrate the basic concepts, followed by the full selective undo algorithm. Both algorithms assume that an operation has already been chosen to be undone, based on the identity of the user or some other criterion.

The algorithms are independent of whether a single centralized history list is maintained for all users or every site has its own (possibly different, but equivalent) history list and editor state. For multiple history lists and editor states, the communication protocol between editors should ensure that all editors eventually reach the same state even when operations are being done in parallel at various sites. This issue of maintaining history lists is discussed further in Section 7.

5.1 Limited Selective Undo

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

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

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

$$A B C$$

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

$$B' A' C$$

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

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

$$B' C' A''$$

where $(C', A'') = Transpose(A', C)$. It follows from the Transpose Property that the operation A'' carried out at the end of the above list has the same semantics as the operation A in its original place. Now that A has been shifted to the end of the list, $\overline{A''}$ can be performed giving the history list:

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

Performing $\overline{A''}$ in the present state correctly cancel the semantic effect of A , giving the document state as if operation A had never been performed ; the undo has succeeded!

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

5.2 Selective Undo

We now give a selective undo algorithm which is not limited by prior undo operations (Figure 1). The algorithm is similar to the limited algorithm in Section 5.1, but it uses a more sophisticated conflict-checking technique.

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

$$A \underbrace{B C \overline{B'}}_{\text{do-undo pointer}}$$

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

The *RemoveDoUndoPair* subroutine, given an operation X which is later undone by 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 operations B and C conflict, 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 :

$$A B \underbrace{C D \overline{C'}}_{\text{do-undo pointer}}$$

Now, suppose operation B is to be undone. The algorithm will first copy *HistoryList* from B onwards into *TempHistoryList* so that the original list is not affected by shifting operations. Since there is a conflict between B and C , and C has a *do-undo* pointer, *RemoveDoUndoPair()* will be

```

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

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

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

proc RemoveDoUndoPair(doPtr: ^HistoryRec)
  while doPtr^.next <> doPtr^.undoneBy do
    if 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 from HistTemp list */
  ListDelete(HistTemp, doPtr^.next);
  ListDelete(HistTemp, doPtr)
endproc

```

Figure 1: The Selective Undo Algorithm

called to remove the C and $\overline{C'}$ pair. The resulting (temporary) history list from B onwards will be as follows:

$$B D'$$

where $(D', C') = \text{Transpose}(C, D)$.

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

$$A \underbrace{B C D C' B'}_{\text{do-undo}}$$

5.3 Properties of the Selective Undo Algorithm

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

Lemma 1: A conflicts with \overline{A} .

Proof: Since A and \overline{A} have overlapping read-write sets (by Inverse Property 2), they must conflict.

The implication of this Lemma is that A will not be undone if it has already been undone (unless its undo operation is effectively cancelled by being undone later). This property is important because, in a group environment, it is conceivable that one user requests the system to undo an operation that has already been undone by someone else. Lemma 1 ensures that nothing special is required to handle such a situation — the system will detect a conflict and reject the undo. In the absence of Lemma 1, our algorithms would have to be modified to specifically check whether A , the operation to be undone, has already been undone, to avoid executing \overline{A} twice.

Lemma 2: A and B do not conflict iff \overline{B} and \overline{A} do not conflict

Proof: First notice that, to undo a sequence $A B$, one can execute the sequence $\overline{B} \overline{A}$. From Inverse Property 2, it follows that A and B have overlapping read-write sets iff \overline{B} and \overline{A} have overlapping read-write sets. Therefore, $\text{Conflict}(A, B)$ is true (i.e., A and B have overlapping read-write sets) iff $\text{Conflict}(\overline{B}, \overline{A})$ is true (i.e., \overline{B} and \overline{A} have overlapping read-write sets).

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

Proof: The Lemma follows directly from the Transpose Property. The Transpose Property implies that B' and A' have the same read-write sets as B and A respectively.

Note that if A and B do not conflict, then the sequence $A \circ B$ can be undone by either by executing $\overline{B} \circ \overline{A}$ or by executing $\overline{A'} \circ \overline{B'}$. Lemmas 2 and 3 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 Lemma didn't hold, users might see the strange behavior that they can undo two operations in either order but cannot later redo the operations in either order.

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

Proof: From the Transpose Property, it follows that B' and A' have the same read-write sets as B and A . Furthermore, by Inverse Property 3, A' and $\overline{A'}$ should have the same read-write sets. It thus follows that if there is no overlap in the read-write sets of A and B , then there is no overlap in the read-write sets of B' and $\overline{A'}$, i.e. B and $\overline{A'}$ do not conflict.

The importance of Lemma 4 is that given the sequence $A \circ B \circ \overline{A'}$ it allows B to 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 5: If A does not conflict with B and $Transpose(A, B) = (B', A')$, then A' should not conflict with \overline{B} .

Proof: We omit the proof as it is similar to that of Lemma 3.

The importance of Lemma 5 is that given non-conflicting operations A and B , where B has been undone, it ensures that 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 can fail to do even simple single-user history undo. Fortunately, the Lemma is automatically satisfied when Conflict and Transpose functions are defined as required.

Lemma 6: Any sequence of shifts of an operation in the history list using the Transpose function preserves the semantics of the operation in terms of the conceptual model.

Proof: The lemma follows directly by induction from the Transpose Property. One step of the shift of an operation A in a sequence $A \circ B$ gives A' where, by the Transpose Property, A' is conceptually equivalent to A (in their respective states). By induction on the number of steps in the shift, the operation obtained at the end of the shift has the same semantics as the original operation A .

Corollary of Lemma 6: Shifting an operation to the end of the history list using the Transpose function preserves the semantics of the operation in terms of the conceptual model.

The above corollary ensures that if an operation is undone using the selective undo algorithm, it will be undone in a manner compatible with the conceptual model. If the conceptual model matches with the users' view of the document, undo will have reasonable, predictable behavior at the user level.

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. The non-linear complexity is due to the call on the *RemoveDoUndoPair()* routine, 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 enhanced to provide linear or constant time performance.

6.1 Efficient undo using undo blocks

Each history undo operation in a single-user editor, such as Emacs, can be done in constant time, as a function of the length of the history list. 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 [4, 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 when there is only one user. 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 operator. Some examples of undo blocks are the following:

$$\underbrace{D E \bar{E} \bar{D}}$$

$$\underbrace{C D E \bar{E} \bar{D} \bar{C}}$$

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 as 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 as in the above example, is also tagged and its inverse, \bar{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 one operation would be there on the temporary history list, the operation to be undone, 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 the use of *RemoveDoUndoPair()* works because, with pure transpose functions, any shifting of A to bring it next to \bar{A} would not modify the operations between A and \bar{A} .

Text editors are difficult to design in such a way that transpose functions are pure — the notion of sequencing between characters is very strong in text and most operations have to 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 comparatively easier to design to ensure pure transpose functions and could take advantage of the above property to get a more efficient implementation of undo.

6.3 Constant time algorithm?

Even with pure transpose functions, the algorithm is $O(n)$ primarily because one 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 constant time, we would have a constant time selective undo algorithm assuming that the transpose functions are pure.

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

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

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, $\overline{X_1}, \overline{X_2}, \dots, \overline{X_n}$, become undoable. It appears to us that either one has to pay non-constant time when X_n is undone to mark all the previous nodes that have now become undoable, or pay non-constant time to determine if an operation such as $\overline{X_2}$ is undoable in the absence of marking.

6.4 State-based undo

We have shown above that a general algorithm for doing constant time selective undo appears infeasible if the algorithm takes conflicting operations into account. Several researchers have suggested group undo schemes which are based on the idea of simply executing the inverse of an operation if the inverse is executable in the current state [5, 6]. To distinguish it from our approach, we will broadly call such schemes as *state-based* undo. Unfortunately, the conditions under which state-based undo schemes actually give predictable, reasonable behavior and their semantic effect are usually not stated. Below, we attempt to list, in terms of our model, one set of conditions under which the use of state-based undo schemes may be appropriate:

1. the document can be viewed as a set of non-overlapping objects in the physical model;
2. operations are assumed to conflict if they modify the state of the same object;
3. transpose function is pure, i.e., operations are interchangeable if they do not conflict;
4. an operation is undone by restoring the state of the affected object to its state prior to doing the operation; and
5. when an operation is undone, we also wish to undo all later conflicting operations (including any undo of the operation).

If the above conditions are reasonable for a system, one can carry out an undo by simply executing the inverse of an operation. The inverse operation should restore the state of the affected object to its state prior to the operation undone. In the process, it effectively also undoes the effect of all the later operations that affect the same object.

The above conditions provide some basis for deciding whether a more general selective undo algorithm, such as the one described above, is appropriate or a simpler, constant-time state-based undo suffices. To see why the above conditions are important, consider a situation where condition

(4) is violated. Let's say a graphical document contains a circle of size 6 and the following two operations are done, leading to a circle of size 4.

1. double the radius of the circle
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 (violates condition (5))
- restore the radius of the circle to 12 (the size prior to doing the 2nd operation).

Suppose a user wishes to undo operation 1. If the undo scheme simply executes its inverse, we end up with a circle of radius 2. This is a result likely to be difficult for the user to understand, particularly if the user had not even seen the second operation because some other user carried it out. The result is probably much more understandable and predictable if the inverse of the first operation is chosen to be to restore the radius of the circle to its prior size, 6 (i.e, observing condition (4)).

One problem with use of state-based undo, even under the above conditions, is that the result of undoing a set of operations depends on the *order* in which the operations are undone. For instance, in the above example, if now the second operation is undone, we end up with a circle of size 12, a result different from the initial size, 6, of the circle!

Under some situations, the above interpretation of undo may be acceptable to the users. However, in that case, it should be made clear to the users that that results of undoing two operations may very much depend on the order in which the operations are undone. State-based schemes are probably closer in semantics to being “state restoration” schemes rather than “operation undo” schemes. In our experience, the selective undo algorithm with the optimizations discussed above is efficient enough for almost all practical purposes and also provides cleaner semantics with respect to consistency with the history list.

7 Implementation and Experience

The per-user Emacs-style history undo has been implemented in the DistEdit toolkit [22]. This undo facility automatically becomes available to all editors that use the DistEdit toolkit. At present, *GNU Emacs* and *xedit* are the two editors that we have modified to use DistEdit.

Figure 2 shows the user interface of a DistEdit-based group editor. DistEdit provides a status/control window that informs the user about the state of the group session and functions specific to group-editing. To allow experimentation with group undo schemes, two very different styles of undo were provided — local undo, which undoes the invoking user's last action and global undo, which undoes the last action on the user's history list, irrespective of the owner. 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.

Below are some of the implementation issues we faced and the solutions adopted in DistEdit.

7.1 Replication of History List

In the DistEdit toolkit, a copy of the editor buffer is maintained at each users' site, to keep response times short and provide fault-tolerance. Replication of the editor buffer raises several questions with respect to the implementation of undo :



Figure 2: Sample screen display of a DistEdit-based group editor. The window on the right allows experimentation with the various undo facilities of DistEdit, such as per-user undo and global undo.

- Should the history list also be kept at each editor or should it be maintained in a central “undo” server?
- If multiple copies of history list exist, what messages need to be broadcast to maintain them consistent with each other?

In the DistEdit toolkit, for fault-tolerance and response time reasons, we chose to maintain a history list with each editor. If storage availability were a concern, maintaining a single history list at a special “undo” server would probably have been a reasonable strategy too, at the expense of fault-tolerance and an increase in response time for undo operations.

If a history list is maintained by each editor, as in DistEdit, a decision must be made whether the undo operation or just the results of the undo operation would be broadcast to the group. If the undo operation were to be broadcast, each editor would have to run the selective undo algorithm. In such a case, care has to be taken to ensure that the undo operation behaves consistently at all sites. Two problems can arise in ensuring consistent behavior of the undo operation. First, the history list may not be identical at each site. In several group editing systems such as DistEdit [22] and Grove [9], to ensure good response time, operations are done locally first and then broadcast. Thus, concurrent operations can end up in different orders on different history lists. This can cause a problem if upon receiving the broadcast message containing an undo request (such as “global undo”), different editors end up selecting different operations for undo. Second, unless atomic multicast protocols are used, an undo operation may not be received in the same order with respect to other concurrent operations by all editors. Therefore undo may succeed at some sites and fail at other sites due to conflicts or lock failures.

In DistEdit, to avoid the above two problems, we chose to broadcast the results of the undo operation. An undo operation is executed locally first, and if it succeeds, the resulting primitive operations are broadcast. To other editors, since the operations received look just like other editing operations (INSERT, DELETE, etc.), consistency is easily achieved. Each broadcast message due to an undo also contains a globally unique reference to the undone operation so that every site can create a consistent view of *do-undo* pointers.

7.2 Length of the History List

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

7.3 Multi-operation actions

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

- One user-level action (e.g. *IndentParagraph*) could result in numerous primitive operations (a bunch of *INSERTs*). Users would expect to be able to undo the high-level action in entirety using one undo operation rather than having to undo the primitive operations one by one.
- Undoing many steps at once could be useful for returning to a known previous state. For example, a user may wish to revert chapter 15 of a paper back to the way it was at 5PM last Tuesday (i.e., undo all operations done on chapter 15's region with time-stamps after 5PM last Tuesday), assuming sufficient history with appropriate tags is kept.

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

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

- The history list needs to be extended to keep sufficient information around so that the set of operations that constitute a high-level action can be determined.
- When undoing a high-level action, all the primitive operations that constitute the high-level action need to be shifted to the end and then undone collectively. If conflicts arise during shifting, the undo should not be permitted without first undoing the conflicts.
- *Do-undo* pointers need to go between corresponding operations, which could be high-level.

7.4 Local vs. global undo

In DistEdit, we have provided support for two kinds of undo — local undo and global undo. Local undo undoes the user's last action. Global undo undoes the last action irrespective of the owner of the action. An undo command can fail if either conflicts arise or locks cannot be obtained for executing the inverse commands. In that case, the user gets feedback about the reason for the failure of the undo command.

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 is of questionable use and often confusing for users. Two key problem 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. 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 useful and having predictable effect are:

- 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 *lock-step* editing mode in which views of all participants in a group session are synchronized. Based on above usage experience, it probably would have been sufficient to provide a *single* undo button in DistEdit — which acts as local undo in the normal (non-lock-step) editing mode and as global undo in the lock-step editing mode.

7.5 Dealing with Conflicts

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 handle 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 give the user an option to undo all the conflicting operations along with the requested operation. The user probably would have to be shown the effect of undoing the conflicting operations and this is an interesting user interface research issue. Some possibilities include allowing the user to undo/redo the conflicting operations several times while highlighting the affected regions in the document.

The second approach, the one used 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, is quite reasonable for many usage patterns of group editors where users work on different parts of a document and thus are unlikely to do overlapping tasks. If conflicts do arise occasionally during undo, they could be dealt with by normal editing rather than by the use of undo. Overlapping changes are much more likely to occur in closely-coupled editing, such as in lock-step mode; however in such cases, the use of global undo is quite reasonable, avoiding the need for handling conflicts.

8 Future Work and Conclusions

We have presented a framework for group undo which is simple and generally applicable to a variety of documents. The techniques proposed in this paper have been implemented in the DistEdit toolkit [21]. The techniques are presented in the context of history undo; however, many aspects of the techniques, such as the notions of *Transpose* and *Conflict*, are also applicable to implementing undo based on the linear and US&R models. The focus of the paper was on developing a general framework for group undo and reporting our initial experience in implementing and using such undo facilities in the Distedit toolkit. More systematic studies are needed to determine appropriate interfaces for supporting undo in groupware applications, particularly for dealing effectively with conflicts.

The behavior of undo in a group environment is dependent on defining a conceptual model of a document that matches users’ view of the document. Design of a conceptual model is, to some extent, up to the discretion of the editor designer. For instance, an editor designer may choose to treat read operations (e.g., yanking, browsing) as null operations for the purpose of designing undo facilities, because they do not modify a document. A user-level copy operation may be thought of simply as an INSERT operation or can be thought of as a READ followed by an INSERT operation. An even richer conceptual model could be defined that takes semantic contents of the document

into account. The selective undo scheme proposed in this paper does not dictate the choice among several possible conceptual views. It does allow an editor designer to provide selective undo behavior at the interface level that matches the chosen conceptual model. In general, we believe that the conceptual model chosen should be sufficiently simple so that it can be easily implemented and so that the undo behavior is understandable and predictable for the users irrespective of the document being edited.

The selective undo framework presented in this paper is applicable to implementing a more general class of undo algorithms such as those that undo the operations in a particular region of document or changes made within a certain time. Such undo schemes could be useful not only in groupware systems but also in single-user editors. It will be useful to incorporate such schemes in single-user editors and group editors and evaluate their use in practice.

The history list could also be applicable to other tasks, such as seeing a trace of the evolution of the text [8]. The mechanisms for transposing operations could be useful in providing the additional capability of seeing a *selective* evolution of the text, for instance the evolution history of a particular section of the document. In the future, we plan to explore such uses of the history list.

REFERENCES

- [1] H. M. Abdel-Wahab, S. Guan, and J. Nievergelt. Shared workspaces for group collaboration: An experiment using Internet and Unix inter-process communication. *IEEE Communications Magazine*, pages 10–16, Nov. 1988.
- [2] Gregory D. Abowd and Alan J. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.
- [3] J.E. Archer and R. Conway. COPE: A cooperative programming environment. Technical Report TR-81-459, Cornell University, June 1981.
- [4] 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.
- [5] Thomas Berlage. From undo to multi-user applications. In *Vienna Conference on Human Computer Interaction*, September 20-22 1993.
- [6] Rajiv Chaudhary and Prasun Dewan. Multi-user undo/redo. Technical Report Technical Report TR125P, Computer Science Department, Purdue University, 1992.
- [7] P.P. Chen. The entity-relationship model: toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, January 1976.
- [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*, pages 38–51, January 1991.
- [12] M. Elwart-Keys, D. Halonen, M. Horton, R. Kass, and P. Scott. User interface requirements for face to face groupware. Technical Report CMI-89-020, Center for Machine Intelligence, Ann Arbor, MI, December 1989.
- [13] 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. *CACM*, 8(11):624–633, 1976.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] J.N. Gray. *Notes on Database Operating Systems*, pages 394–481. Springer-Verlag, 1978.
- [18] 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.
- [19] 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.
- [20] W.J. Hansen. User engineering principles for interactive systems. In *AFIPS Conference Proceedings*, volume 39, pages 523–532. AFIPS Press, 1971.
- [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] 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.
- [26] Alex Mitchell. personal communication, University of Toronto, 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] Rajeev Rastogi, Henry F. Korth, and Abraham Silberschatz. Strict histories in object-based database systems. In *Proc. of ACM Conference on Principles of Database Systems*, 1993.
- [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.