5

Group Editors

ATUL PRAKASH
University of Michigan

ABSTRACT

This chapter focuses on group editors, an important class of collaborative tools that allow multiple users to view and edit a shared document simultaneously. Building group editors requires solving non-trivial problems such as providing adequate response time for edit operations, ensuring consistency with concurrent updates, providing adequate per-user undo facilities, and providing collaboration awareness. Design choices are presented for implementing these facilities as well as examples of implementations from several group editors.

5.1 INTRODUCTION

A group editor is a system that allows several users to simultaneously edit a document without the need for physical proximity and allows them to synchronously observe each others' changes. Group editors are a way to enhance collaboration by providing a shared workspace in which users can organize ideas, work jointly on papers, do brainstorming, etc. A group editor should have most of the functionality of single-user editors, such as being able to open, edit, and save documents.

In addition, group editors must usually be designed to have the following features:

- *Collaboration awareness:* A group editor should provide sufficient context information so that users are aware of other active participants in the group session. It should also facilitate sharing of views and sufficient idea of the work each participant is doing so as to encourage communication and avoid conflicting work.
- *Fault-tolerance and good response time:* A group editing session should continue to run smoothly despite machine crashes and people joining or leaving a session. Also, the editors should provide interactive response time for frequently done operations, such as browsing and a sequence of updates by a particular user.

Computer Supported Cooperative Work, Edited by Beaudouin-Lafon

- *Concurrency control:* Concurrency control is needed to ensure consistency of data being edited when parallel editing is going on. Concurrency control protocols should be designed to minimize the impact of network latency on response times experienced by users, so that group work using the editor is not an inconvenience.
- *Multi-user undo:* A group editor should allow users to individually undo their changes. This is important because users may use a group editor to work in parallel on different parts of a document. Users should be able to use an undo command to reverse their own mistakes even if their change was not the last one carried out in the editor.
- *Usable as a single-user editor:* Group editors should provide good support for single-user use. Users should not have to switch to a different editor when they are working alone or asynchronously.
- *A rich document structure to serve as a medium of collaboration:* Some group editors use the document as a medium for brainstorming, organizing ideas, or as a means of communication among users. An important factor in the design of such group editors, as a result, is providing an appropriate document structure that facilitates group communication.

The rest of the chapter is organized as follows. It first gives examples of several group editors, illustrating how they can be used to support collaborative activities. Then, it presents the high-level architecture of typical group editors so that fault-tolerance and response time requirements can be met. Next, it suggests several approaches to addressing concurrency control requirements and supporting undo in group editors, and the tradeoffs between the approaches. Then, it discusses the collaboration awareness features that can be useful to provide in group editors. After that, it presents the structure of content in group editors that are designed to support specific collaborative tasks such as brainstorming activities. Then, it briefly highlights other design issues that arise in building group editors. Finally, it presents some directions for future work in group editors.

5.2 EXAMPLES OF GROUP EDITORS

5.2.1 Group Graphical Editors

Dolphin [Str94] is an example of a graphical group editing environment for supporting joint work and brainstorming by users who are not co-located. In Figure 5.1, four users are using a shared document in Dolphin as the medium to support brainstorming. Users can sketch, type, or create links to other pages to communicate their ideas. Audioconferencing tools, such as MBone's *vat*, are often used with Dolphin so that users can conveniently discuss the contents of the document.

The rich hyperlink-based structure of documents in Dolphin allows different kinds of collaborative tasks to be supported. Simple brainstorming tasks may use the system only as a graphical sketch pad. More involved, decision-making discussions can choose to take advantage of the hyperlink-support to organize the discussions into IBIS-like decision tree structures [Rei91]. The system does not have most of the formatting features of commercial (single-user) word-processing systems such as Word or LaTeX, though, in principle, it could be extended to support more extensive formatting features.

Many of the graphical operations, such as dragging or resizing objects, require high interactivity, independent of network latencies. Dolphin provides immediate feedback on such

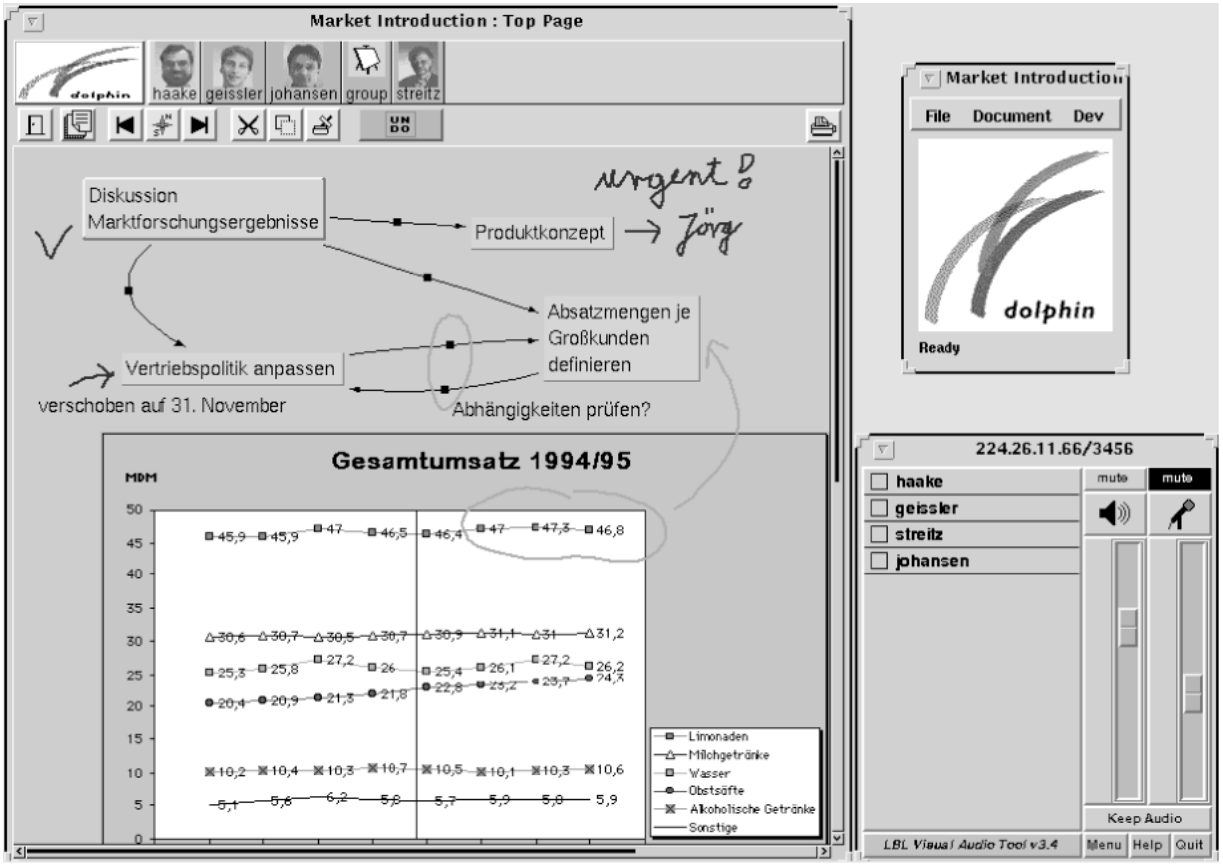

Figure 5.1 An example of a group editor. The left window shows a document being used to support collaboration among four users in a shared window. An audioconferencing tool (shown on the bottom right) is also used to facilitate interaction. ©1996 by GMD-IPSI, reprinted by permission.

operations. At the same time, Dolphin guarantees that all users will see a consistent state at quiescence, even if several users edit the document simultaneously.

Temporary anomalies may arise when several users attempt to modify the same object. For example, if two users drag the same object to different places simultaneously, they will initially see the object being dragged in their own direction. However, when the changes are propagated to other users, one of the operations is undone by the concurrency control algorithm. As a result, one of the users may see the object suddenly move back to its old location (undoing the user's change) and then move to the location selected by the other user (executing the other user's operation).

A Dolphin window shows the list of users who are looking at the same document. As users open or close the document window, the list is automatically updated. This is a form of group awareness that can be critical to the successful use of group editors. Users often need to know if other users are also looking at the same document, in order to have a meaningful discussion about the document. Audio communication among group members can provide additional context and awareness.

Dolphin allows users to have one shared public window and multiple private windows for a given document. Navigation in the public window (e.g. following links to other pages of the document) is visible to all the users. Navigation in a private window is private to a user. Editing changes to the document itself, however, are not private.

Multi-user whiteboards, such as MBone's *wb* and those in Netscape's Cooltalk and Intel's ProShare system, provide basic sketching facilities for brainstorming, as in Dolphin. These systems do not, however, provide the ability to create links to other parts of a document, or the ability to use both private and shared windows into a document.

5.2.2 Group Text Editors

All the above editors are primarily graphical editors. The support for text is generally limited to placing simple textual objects at a selected location in the graphical document. Text is usually treated as a graphical object that can be placed at a selected coordinate on the document canvas. Simultaneous update of a text object is usually not supported. Thus, these editors are not really appropriate for creating large text documents jointly.

Several group editors, such as GROVE [Ell88], DistEdit-based Emacs [Kni90], MACE [New91], and SASSE [Bae93] have explored issues in providing support for joint editing of text. In these editors, simultaneous editing of text objects is allowed, even within the same sentence or paragraph.

Allowing simultaneous editing of a related sequence of characters raises interesting concurrency control issues. Consider the following example:

> A document contains a string *ompute*. Suppose user A attempts to insert the character r after character e. In many text editors, this would be carried out using an operation $InsChar(7, r)$ on the document, inserting r at position 7 in the document. But, now suppose that between the time the operation is generated by A's input and the time it is executed, another user's operation $InsChar(1, c)$ is executed, in order to insert a c before the o. If the operations are simply executed in the order $InsChar(1, c)$ followed by $InsChar(7, r)$, the resulting string would be *computre*, rather than the intended result of *computer*.

Such a problem of unintended results rarely arises in graphical editors because most operations are with reference to absolute coordinates on a canvas. Intended results can usually

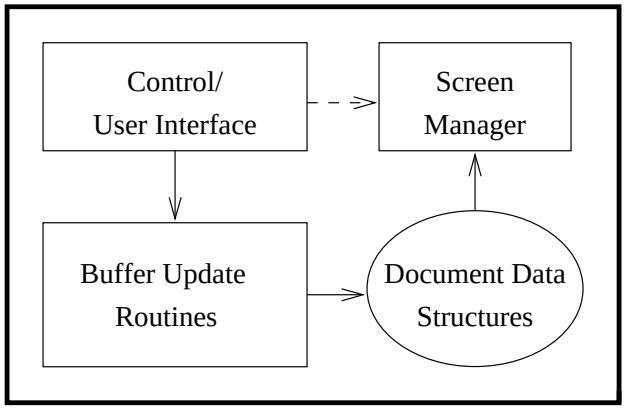

Figure 5.2 Typical structure of a single-user editor

be achieved by executing the operations in the same order at all sites. Furthermore, simultaneous operations often commute; when they do not, the differences in the results may not be significant enough for users to care [Gre94].

In a text editor, on the other hand, users usually intend their operations to be relative to positions of existing characters. However, internally, editors often represent operations using offsets from beginning of the text object. As illustrated in the above example, this can lead to unintended results. Some of the techniques for dealing with this problem are discussed later in the chapter.

Another class of group editing environments are those that support more asynchronous or non-real time styles of interaction. Examples are editors such as CES [Gri76], Quilt [Fis88], and Prep [Neu90]. Prep, for instance, introduced a novel interface in which multiple columns are used, with the first column displaying the editor's text, and subsequent columns showing the comments on the text by the collaborators in the group. These editors allow users to work on the same document but typically on different sections and at different times. As a result, interactions are over a much longer duration, even up to several days. Many of the issues of concurrency control, fault tolerance, and real-time propagation of updates are less relevant to such systems. This chapter does not discuss these systems.

5.3 GROUP EDITOR ARCHITECTURE

The high-level structure of a typical single-user editor is shown in Figure 5.2. A user interface and control section waits for input; when input is received, it is translated into a set of calls which update the document or update the interface.

Group editors, in order to provide interactive response times on browsing operations, usually use a fully replicated architecture in which the document state is replicated at each site (see Chapter 7 in this book [Dew99]). As an example, Figure 5.3 shows the replicated architecture of DistEdit-based group editors. DistEdit is a toolkit that allows existing text editors to be converted to group editors with minimal changes to their code as well as to ease development of new group text editors. Several editors, including MicroEmacs, Xedit, and Gnu Emacs, have been modified to make use of DistEdit. In DistEdit, modifications to an editor's document state are done using a set of standardized update primitives. Each editor's update

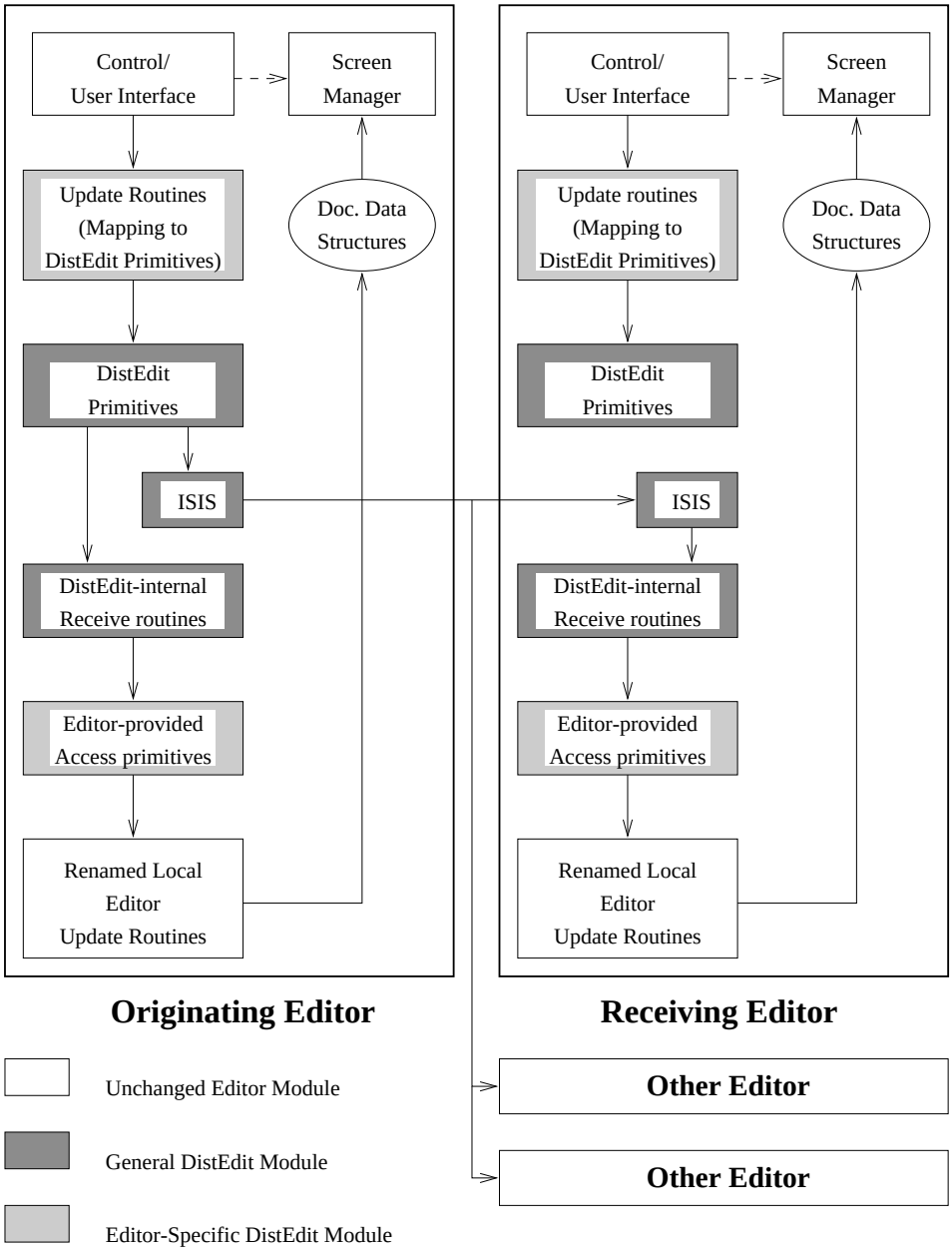

Figure 5.3 Replicated architecture of group editors built using the DistEdit toolkit

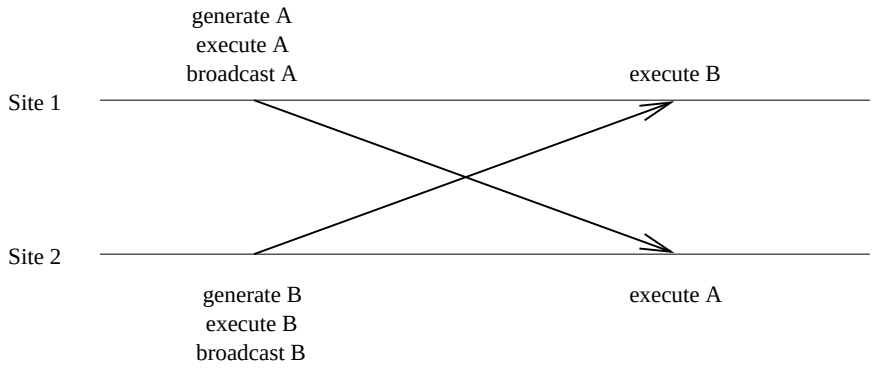

Figure 5.4 A scenario of the document state becoming inconsistent at different sites. Without concurrency control, the two operations A and B are executed in different orders at different sites, possibly leading to different states

operations are mapped to one or more calls on the DistEdit primitives. Those DistEdit primitives are multicast to all the editor copies, using the ISIS communication package[Bir90]. All the editors then apply the updates to their local copies. This general idea of replicating state and propagating changes is used in many group editors, though different editors differ in the choice of update primitives, document type, user-interface features, and algorithms for concurrency control.

As we will see in Sections 5.4 and 5.5, selection of a core set of update primitives that directly update the document state in a group editor is helpful in the implementation of concurrency control and multi-user undo. Usually, implementing concurrency control algorithms and undo is simpler if this core set of update primitives is kept small. Additional update operations can be defined in terms of the core set of update primitives, without substantially complicating concurrency control and undo algorithms.

5.4 CONCURRENCY CONTROL

Concurrency control techniques are required to ensure that a document's state in a replicated architecture remains consistent even when users attempt to modify the document simultaneously in a group editing environment. Consider a case where the state, S, of the document is initially consistent (identical) at the various sites. Let us consider the simple case that two users attempt to modify the document simultaneously via operations A and B. If each operation is executed locally first and then broadcast for execution at other sites (Figure 5.4), the operations would be applied in different orders at different copies of the document, potentially leading to inconsistent states — an undesirable situation in general.

One solution to the data consistency problem is to use ordered broadcast protocols to ensure that all broadcasts are received in the same sequence at all sites [Bir87, Cha84]. However, in this case, the sender of a broadcast has to wait to receive its own message from the network before it can execute the operation. In fact, it may receive other sites' messages prior to receiving its own message owing to message ordering requirements (Figure 5.5). This waiting can lead to poor interactive response-times in multi-user use. Unfortunately, it can also lead to poor interactive response times when one user is primarily interacting with the editor because

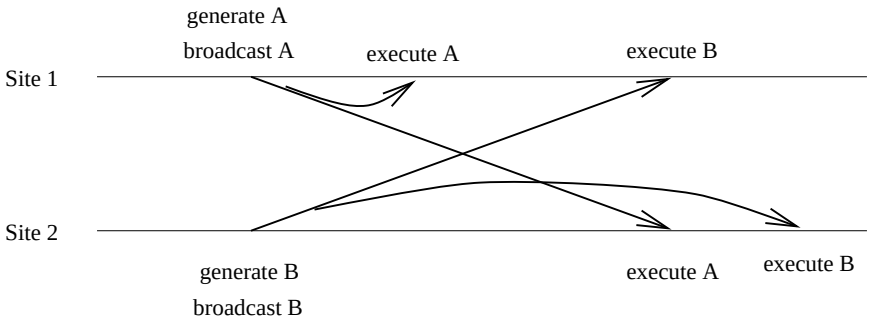

Figure 5.5 The use of ordered broadcast protocols to achieve data consistency. Delivery and execution of message B must be delayed at site 2 so that delivery/execution order is consistent with that at site 1. Also, B is executed in a different state than when it was generated, possibly leading to an unintended document state

ordered broadcast protocols usually rely on a central site to determine the order of delivery of messages. Another problem with this solution is that operations received from other sites may be done on the application's state between the time that the operation was generated and the time it is actually executed; executing the operation in the modified state may not lead to correct or intended results, as we will see later.

Another solution to the data consistency problem is to use a centralized data architecture, where the copy of the data resides only at one site. Without the use of any other concurrency control scheme, this is essentially equivalent to the use of a replicated architecture that uses ordered broadcast protocols. In particular, it has similar response time problems. It can have much worse performance for browsing if sites do not cache the document state locally. Also, the possibility of unintended results remains because the centralized site may execute parallel operations in an arbitrary order, perhaps leading to unintended results.

There are thus two key aspects of ensuring consistency that group editors must deal with:

- consistency of state among the various document copies, assuming that a replicated architecture is used in order to enhance response time
- consistency of the resulting state of a document with respect to a user's intention when doing an update, in cases where other participants' operations are applied to the document between the time the user's operation is generated and the time it is executed.

Techniques for maintaining consistency among copies of a document are largely based on algorithms that have been proposed in the work on replicated databases. However, performance tradeoffs are different between group editors and database systems, and that leads to different choices of concurrency control algorithms. Group editors must provide interactive response times; impact of network latencies on response time to user input must be minimized. The concurrency control strategies in databases, on the other hand, must usually maximize transaction throughput, rather than the response time of individual transactions.

Another difference from database systems is that group editors usually do not use a transaction-based approach to updates because of the complexities involved in developing support for transactions in a general-purpose programming language and for systems requiring highly-responsive, interactive graphical interfaces. However, some recent systems, such as COAST [Sch96] and DECAF [Str97], provide transaction-based support for building groupware systems.

There are two broad classes of concurrency control techniques: *pessimistic* and *optimistic*. Pessimistic techniques ensure that inconsistencies among copies do not arise by requiring that any update operations acquire appropriate locks to prevent conflicting updates from occurring. Optimistic techniques do not prevent inconsistencies from occurring, but use mechanisms to detect and correct inconsistencies if they occur.

Almost all practical databases use lock-based pessimistic techniques because they usually provide a better transaction throughput. Many group editors, on the other hand, use optimistic or special pessimistic techniques, with the goal of reducing interactive response times. The following first discusses strategies for applying pessimistic and optimistic strategies to group editors and then discusses enhancements to the strategies to avoid inconsistency among documents with respect to users' intentions.

5.4.1 Pessimistic Concurrency Control

To ensure consistency among copies of a shared document, one strategy is for operations to acquire network-wide locks before updating the various document copies. Thus, in the earlier example of two users doing operations A and B in parallel, the operations can be executed on all the copies in the order in which the operations acquire locks.

If acquiring or releasing locks requires going over the network, users may still perceive substantial increase in interactive response times because each user's update operation will involve acquiring some locks over the network, doing the operation, and then releasing the locks. In fact, if in a group session only one user is interacting with the application, the same overhead could occur. Such a situation is clearly undesirable.

To improve performance of lock-based schemes, one technique is to use a token-based locking scheme. DistView [Pra94b], a general-purpose toolkit for building groupware applications, uses a token-based locking scheme. When a site acquires a lock, it gets a lock-specific *token*; only one site can have the token at a time. When a site releases the lock, it is treated as a *hint* that the lock is no longer needed. The site still retains the token, but marks it as *available* for other users. If the same site wishes to reacquire the same lock, the lock can be granted immediately without going over the network by simply marking the token as *unavailable*. If another site wishes to acquire the lock, it sends a message out to the entire group, requesting the lock. The sites without the token ignore the message. The site with the token transfers the token if it is marked *available*; otherwise it denies the lock request.

The performance impact of the above token-based scheme is that network latencies in acquiring locks occur only if the lock has to be acquired from some other site. If only one user is repeatedly acquiring and releasing locks, no network messages need to be sent except for the first lock request (Figure 5.6). This can be an efficient locking strategy in practice because usage patterns in group editors are often such that one user does most of the interactions, while others observe the changes. A complication in the token-based scheme is that, for fault-tolerance, a distributed token recovery algorithm is needed to deal with situations where a site crashes while holding a token.

Another technique to reduce impact on response time is to support multiple, fine-grain locks on the document. Different users may hold different locks, so the likelihood of waiting on locks can be reduced. In DistEdit-based text group editors [Kni90], the granularity of locks can be as small as one character. A lock covers any contiguous region of text. Inserting a string requires obtaining a lock on the character which precedes the point of insert. Deleting a string requires a lock covering the characters of the string.

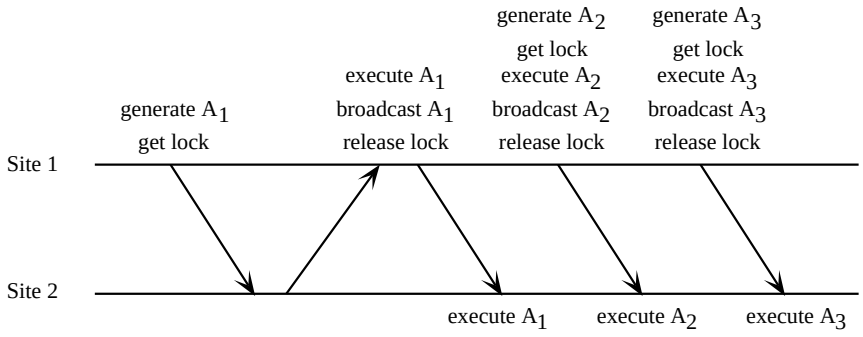

Figure 5.6 The DistView toolkit uses a token-based locking scheme to help improve response time for typical usage patterns. Site 1 needs to acquire a token from a remote site for the first operation A_1. For subsequent operations that require the same lock, no token needs to be acquired over the network, leading to good response times

As insertions or deletions are performed within a region, the associated lock in DistEdit expands or shrinks automatically, without re-acquiring fresh locks over the network. Thus, for insert operations (perhaps the most frequent operation in text editors), network latencies impact response time only for the first insert at a new position; subsequent characters are inserted with similar response times as in single-user editors.

Browsing operations in group editors normally do not need to acquire any locks (unless the editor supports synchronized browsing). Thus browsing can be done interactively, independent of updates initiated at other sites.

Locking, as described above, is largely intended to be hidden from users. Locks are automatically acquired or released as users interact with the editor. Several group editing systems, including MACE [New91] and DistEdit, also support *explicit* locking, where a user deliberately selects and locks a region of the document. A user may chose to acquire an explicit lock to work on a region in order to indicate to others that they should not work on that region until the lock is released. Techniques for handling explicit locks are similar to those used for handling automatic locks, except that explicitly-acquired locks are kept until released by the user.

5.4.2 Optimistic Concurrency Control

Another technique to ensure consistency is to use optimistic concurrency control. An operation is executed on the local copy immediately and then broadcast to other sites for execution. All update operations are first time-stamped so that any two operations can be consistently reordered at all the copies, even if they are received in different orders. To reorder operations, each site has to maintain a *history list*. The history list is a sequence of operations that have been performed on the document. The operations on the history list are stored in the order in which they were performed. For instance, if the history list is

$$B\ C\ D$$

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

Consider the situation now if operation A is received by the above copy, where A has lower

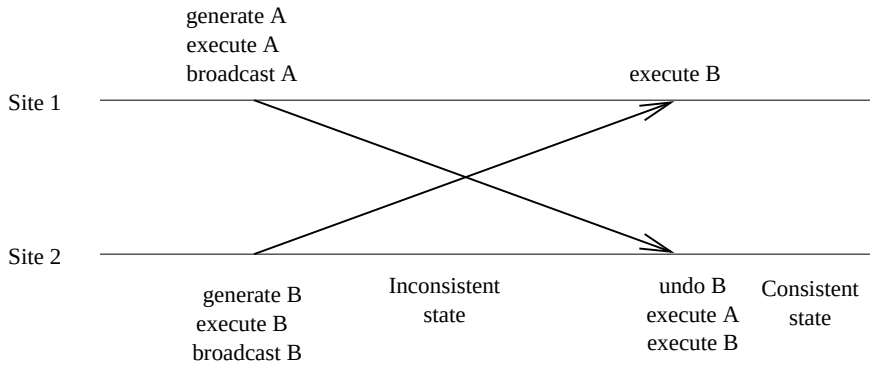

Figure 5.7 The use of undo/redo strategy for optimistic concurrency control. At site 2, execution order is made consistent with that at site 1 by undoing execution of B and then executing A and B. Site 2 is temporarily in an inconsistent state with respect to site 1

time-stamp than B, C, and D. In such a case, the operations can be reordered by *undoing* D, C, and B (in that order), performing A, and then performing B, C, and D. The resulting history list will be

$$A\ B\ C\ D$$

For this scheme to work, inverses need to be defined for all operations so that they can be undone. Figure 5.7 shows the use of optimistic concurrency control to achieve data consistency for the example in Figure 5.4.

Karsenty and Beaudouin-Lafon [Kar93] describe an algorithm that improves the performance of the above undo/redo scheme by taking advantage of commutativity among operations, when the commutativity information is provided to the editor. The following example illustrates the use of undo/redo in their scheme. Suppose that two operations A and B should be executed in the order A followed by B. However, one of the sites in a groupware system 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 site if A and B commute. The algorithm will in fact not execute A at all if B *masks* the effect of executing A; i.e. executing A followed by B gives the same results as just executing B. If commutativity or masking does not occur, the algorithm will undo A, execute B, and then redo B to correct the execution order. This algorithm is used in several systems including Dolphin and COAST [Sch96].

Note that the undo/redo in the above optimistic concurrency control scheme 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 an operation that is executed in the correct order is not addressed by the above scheme.

Unlike in lock-based pessimistic schemes, optimistic schemes suffer from a window of opportunity where a user can interact with the editor while the user's copy of the document is in an inconsistent state. For instance, in Figure 5.7, the user at site 2 could issue editing operations immediately after broadcasting B. These operations would also execute optimistically at site 2. The problem is that these operations may be generated based on a state that is later going to be undone. Unlike in databases, the optimistic (inconsistent) state is visible to the users, since the goal of using an optimistic algorithm is to reduce response time. Currently,

there does not appear to be a good solution to this problem. Most editors that use optimistic schemes simply assume that such a possibility does not arise too often, and when it does, users can deal with any unintended effect.

5.4.3 Consistency with Users' Intentions

A group editor must not only provide a consistent document state at each site, but must attempt to perform operations with effects that are consistent with users' intentions. The algorithms, as discussed above, need to be enhanced to address this. Consider the example given in Section 5.2.2, which is repeated below:

> **Example 1**
> A text editor's document contains the string *ompute*. User A attempts to insert the character r after character e. In many text editors, this would be carried out using an operation $InsChar(7, r)$ on the document, inserting r at position 7. But, now suppose that between the time the operation is generated by A's input and the time it is executed, another user's operation $InsChar(1, c)$ is executed, in order to insert a c before the o. If the operations are simply executed in the order $InsChar(1, c)$ followed by $InsChar(7, r)$, the resulting string would be *computre*, rather than the intended result of *computer*.

The above problem is not a replicated data inconsistency problem because all the copies of the data will have the same string. However, it is an inconsistency with the user's intention of inserting the character r after the e in the string. The inconsistency arose because another user's action was carried out on the document between the time the user initiated the action and the time it was executed. Thus the user's operation, which used positional offsets, was applied in a different location than intended, leading to unintended results.

The reader may think that the above scenario of two users modifying the same word is not very likely. However, the same problem arises if users are modifying different parts of a long document (a more likely scenario) as long as references used in the operations change as a result of other editing operations.

The solution commonly used in group text editors to address the problem is to *detect* the possibility of an undesirable result, *modify* the operation so that it leads to the intended, desirable result, and then execute the modified operation.

In Example 1, this scheme would transform the second operation from $InsChar(7, r)$ to $InsChar(8, r)$ so that it leads to insertion at the correct point, given that the first operation has already been executed.

The use of transformations requires the definition of a *transformation matrix* Tr [Ell89]. $Tr(A, B)$ tells how an operation A should be transformed to give the intended effect, given that another parallel operation B has already been executed. For instance, if A and B are generated in parallel by different users and

$$Tr(A, B) = A' \text{ and } Tr(B, A) = B',$$

then A' should be executed instead of A if B has already been executed. Similarly, B' should be executed at a site instead of B if A has already been executed.

Transformations can be used in several ways in optimistic schemes. Figure 5.8 shows the original use of it in the GROVE editor [Ell88, Ell89]. Site 1 executes the parallel operations A, B as A followed by B', whereas site 2 executes the operations as B followed by A'. Data consistency obviously requires that the transformation matrix satisfy the *transform property* that executing A followed by B' results in the same state as executing B' followed by A.

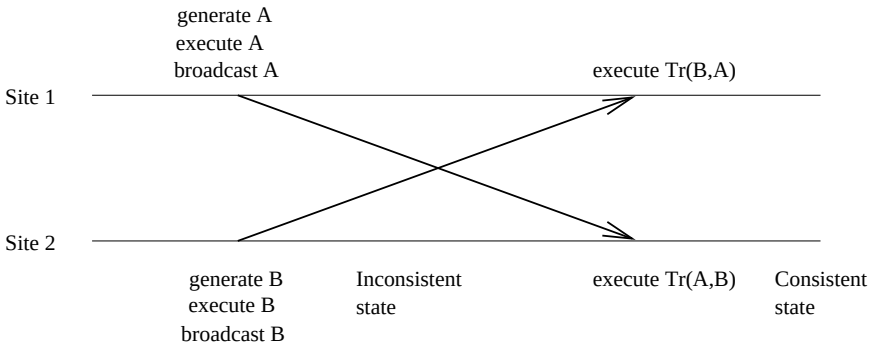

Figure 5.8 The use of transformations to achieve data consistency and consistency with users' intentions in an optimistic manner, without the use of undo/redo. Sites may do operations in different order and transformations must satisfy constraints that ensure consistency

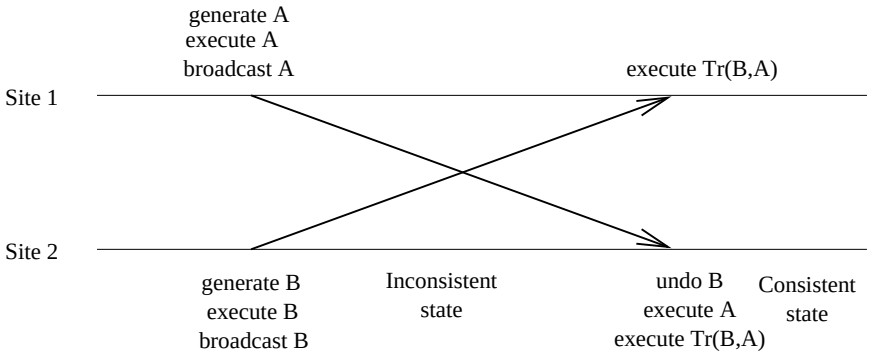

Figure 5.9 The use of transformations to achieve data consistency and consistency with users' intentions in an optimistic manner using undo/redo. In this case, all sites execute operations in the same sequence, if necessary after reordering operations

Figure 5.9 shows the use of transformations with the undo/redo concurrency control scheme. In this case, site 1 executes the operations as A followed by B'. Site 2 executes them in the same order, after undoing B to achieve the same order. This has the slight advantage that data consistency is achieved even if the transformation matrix fails to satisfy the transform property. The disadvantage is that the undo/redo-based strategy can be somewhat slower.

Transformations can also be useful in pessimistic schemes. For instance, if an ordered broadcast protocol is used, transformations can be applied to achieve the same results as in the undo/redo scheme shown in Figure 5.9.

With lock-based schemes, transformations can be used for mapping lock requests to correct regions in the document, even when parallel lock requests are made. Operations do not need transforms in that case, because operations can use positions that are relative to the positions specified in the locks. This simplifies transformations considerably, because the number of operations can be much larger than the types of lock requests.

A second approach to the handling inconsistency with user's intentions is to simply do nothing. The assumption is that users can correct any unintended results manually. This may

be an acceptable approach in some cases. It is not an acceptable solution, however, if the problem is likely to occur frequently during group editing, if it could have a hard-to-correct effect on the document state, or if the users are unlikely to notice the problem when it occurs. For simple graphical editors, such as shared whiteboards, this can be a reasonable approach, because most operations can be designed so that transformations are not required.

A third approach to the problem is to *detect* the possibility of an undesirable result and to *abort* the second operation on all copies of the document (i.e. not execute the operation, or undo it if it has already been executed). This is somewhat better than doing nothing because undesirable results are prevented and the user who initiated the aborted action can be notified (e.g. via a beep) that the operation was not carried out.

Designers of sophisticated group editors may find it practical to rely on all the three approaches for different aspects in the same editor. The approach of modifying operations may be used in the same editor to deal with commonly-occurring situations that are easy to detect and correct, such as changes in the positional offsets of an operation as a result of parallel operations.

The second approach of doing nothing may be best for situations where the system cannot easily determine, usually owing to difficult data semantics, whether the results will be considered to be unintended by users. For example, if given the word *helo* in a document copy, if one user attempts to insert an l after the e, and another user attempts to insert an l before the o simultaneously, the resulting string would be *helllo* in most text group editors — perhaps an unintended result if the users expected the result to be *hello*. However, this result is difficult to avoid if the editor treats its document state as simply a string of characters — the concept of words and perhaps even spellings would have to be introduced in the editor to result in the intended string *hello*. And even that could be an unintended effect to some users because they may have expected both letters to be inserted. A more practical solution in this case is to design the editor to provide *predictable* behavior that will be compatible with users' intentions in *most* cases, and to provide collaboration-awareness features so that users are less likely to make updates that lead to unintended results.

The third approach of aborting operations may be useful in the same editor when operations conflict so severely that only one can be done meaningfully. For example, in editors that provide locking, if two users attempt to lock an overlapping document region simultaneously — obviously, only one operation can be allowed to succeed, and the other has to be aborted.

5.4.4 Alternatives to Concurrency Control

Another way to deal with concurrency control is to ensure that parallel editing operations always commute. In that case, neither pessimistic, nor optimistic concurrency control is needed. Operations can be executed by sites as they are received.

The MBone-based whiteboard, *wb*, used such a strategy. In *wb*, users could only modify their own work on the whiteboard — not that of others. This restriction ensured that parallel updates from different users modified different objects on the canvas, thus removing the need for a concurrency control algorithm. Such a strategy was useful in *wb* because in an MBone session, thousands of participants were often expected. Attempting to do concurrency control among such a large group was likely to be inefficient.

In a graphical editor that does not use any concurrency control, it is possible to get some inconsistencies. For instance, if two users move different filled objects, say A and B, simultaneously to the same location on the canvas, it is possible that one user's display shows A

as being above B while the other user's display shows B as being above A. If such inconsistencies are acceptable to users, then the two operations can be considered to commute. Otherwise, a concurrency control algorithm is likely to be needed.

5.5 UNDO IN A GROUP EDITOR

The ability to undo operations is a useful standard feature in most interactive single-user applications. For instance, the availability of an undo facility in editors is useful for reversing erroneous actions [Han71]. It can also help reduce user frustration with new systems [Fol74], particularly if those systems allow users to invoke commands that can modify the system state in complex ways.

Compared with single-user applications, performing undo in groupware applications provides technical challenges in the following areas [Pra92, Pra94a]:

- *Selecting the operation to be undone.* In a group editing 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 the operation to undo based on who performed it.
- *Determining what operation will result in a correct undo.* 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.
- *Dealing with dependencies between different users' operations.* 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.

Supporting undo in group editors requires a *history list* — which was previously used in Section 5.4.2 for doing optimistic concurrency control. If the editor uses a replicated architecture, a concurrency control scheme should be used that results in the history lists being consistent (i.e. operations added to the history list in the same sequence) at all sites.

In addition to maintaining a history list, supporting undo requires that all operations that modify the state of the document are *reversible*; i.e. for every operation A, we can determine an inverse operation $\overline{A}$ that will undo the effect of A, assuming A was the most recent operation executed. For instance, in an editor, an *INSERT* operation can be undone by a *DELETE* operation.

We next look at design strategies to address the above three issues.

5.5.1 Selecting Operations

In a group editor, a user may wish to undo his last operation, but that operation may not have been the globally last operation executed on the document (other users may have done operations subsequently). We therefore need to allow undoing of a *particular user's* last operation from the history list.

To allow such selection of the operation to undo based on user identity, each operation

on the history list needs to be *tagged* with the *user-id* of the user who invoked the operation [Pra92]. 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. The selection mechanism would choose to undo A_2, the last operation on the list that is tagged with Ann's user-id.

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 selection on other attributes, tags could include additional information such as the time at which the operation was carried out or the document region in which it was carried out.

In DistEdit, we have found undo that is restricted to a particular region of a document (e.g. paragraph) to be particularly useful, in addition to an undo based on user identity.

The above scheme has been termed *selective undo* [Pra92, Ber94, Pra94a], since the operation to be undone is not necessarily the last one, but is selected using some attributes attached to the operation.

5.5.2 Executing the Undo

Once the correct operation to be undone is selected, the operation to execute to effect an undo has to be determined. We look at several strategies for executing the operation.

5.5.2.1 Direct Selective Undo Strategy

One potential solution for undoing any operation in the history list is simply to execute its inverse, provided the inverse is executable in the current state [Ber94]. So, given the following history list:

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

operation A_2 is undone by simply executing A_2's inverse, $\overline{A_2}$, resulting in the history list:

$$A_1\ B_1\ A_2\ B_2\ B_3\ \overline{A_2}.$$

This approach essentially assumes that any operation in the history list can be undone simply by executing its inverse from the current state (provided the inverse can be executed), irrespective of the other 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 example.

Example 2
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 the first user now issues a command to undo operation 1. To undo operation 1, using the above undo strategy, the inverse of operation 1 is executed, resulting in a circle of radius 2. Unfortunately, since the circle was never of size 2, this may be a result that is difficult for the users to understand.

Another problem with the strategy 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.

This strategy, despite the above problems, may be useful in some cases. First, if operations on the editor are carefully designed to always commute, then this strategy will give expected results. For instance, in the above example, if the second operation was replaced by an operation that reduces the radius of the circle by a factor of 3 (and its inverse being an operation that increases the radius of the circle by a factor of 3), then this strategy, as can be verified, would give expected results. Second, as suggested in [Ber94], if the users are presented with the list of operations that have been done and explicitly select one to be reversed with the understanding that the system will simply execute the inverse of the operation in the present state, then the results can be better understood by users.

5.5.2.2 Undo–Redo Strategy

Another strategy for undoing an operation is to bring the document to a state prior to an operation A by undoing all operations executed since A (in reverse order), then undoing A, and then redoing all the undone operations except A [Cho95]. This strategy is similar to that used in undo-skip-redo (US&R) strategy [Vit84] for single-user editors. For example, given the history list:

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

to undo A_2, first inverses of B_3 and B_2 are executed, then the inverse of A_2 is executed, and finally B_2 and B_3 are re-executed. This results in the history list:

$$A_1\ B_1\ A_2\ B_2\ B_3\ \overline{B_2}\ \overline{B_3}\ \overline{A_2}\ B_2\ B_3$$

or its equivalent, in terms of the effect on the document state:

$$A_1\ B_1\ B_2\ B_3\,.$$

For Example 2, if operation 1 is to be undone, the effect would be to undo both operations and then redo the second operation, resulting in a circle of size 4. This is a reasonable result in the sense that the circle would have been of size 4 if operation 1 had never been done. On the other hand, a problem remains that this may not be what the user intended to happen because the undo will appear to have no effect and no error will be reported [Ber94].

Both this strategy, as pointed out by its authors [Cho95], and the direct selective undo strategy, do not account for the need for transformations. Consider the following example from a text editor.

Example 3
Let's say that a text document contains only the string *omputr*. The following two operations are done in sequence by two users, leading to the string *computer*:

- *Operation 1:* $InsChar(1, c)$ to insert *c* before *o*, the first position in the string, resulting in the string *computr*.
- *Operation 2:* $InsChar(7, e)$ to insert *e* between *t* and *r*, the seventh position in the string.

Reasonable inverses for the above operations are:

- $DelChar(1)$ and
- $DelChar(7)$.

If now, the first user attempts to undo operation 1, we would like the result to be *omputer*, the string that would have resulted if operation 1 had not been executed by the first user. However, the undo/redo strategy would first restore the string to *omputr* by undoing both operations and then re-execute operation 2, leading to the string *omputre*. The direct selective undo strategy works for this example, but does not work if operation 2 had been done prior to operation 1.

The problem that occurred with Example 3 is that the second operation would have executed as $InsChar(6, e)$ — at a different position — if the first operation had not occurred, assuming that the intended effect of the operation was to insert *e* between t and r. Unfortunately, this basic strategy does not take such needs of modifications to operations into account [Abo92, Pra92].

Despite the above limitations, the undo/redo strategy can be a useful one, especially when transformations are not required and users accept its semantics. Results after multiple undo commands, unlike the direct selective undo strategy, are independent of the order in which they are carried out.

5.5.3 Transformation-Based Selective Undo

The basic problem illustrated by Examples 2 and 3 is that to undo an operation other than the last one on the history list, one cannot simply execute the inverse of the operation (or use the undo/redo strategy) because subsequent operations could have shifted the location at which the operation was originally performed.

Another problem with implementing selective undo is the the possibility of *dependencies*, or *conflicts*, between operations. Suppose an operation B has modified the same region of the document as an earlier operation A. It may then not be possible to undo A without first undoing B. A general solution to undo needs to be able to detect when an operation cannot be undone because of later conflicting operations that have not been undone.

A general solution to the problem of dealing with transformations and conflicts is presented in [Pra94a]. Here we examine an intuitive description of the solution used. To allow an arbitrary operation on the history list to be undone, the solution in [Pra94a] requires that the application supply functions which can detect *conflicts* between operations, *re-order* non-conflicting operations, and create *inverse* operations. More specifically, besides the inverse function, the application must provide the following two functions:

- $Conflict(A, B)$ that returns $true$ if the operations A and B performed in sequence cannot be reordered, and $false$ otherwise.
- If A and B do not conflict, a function $Transpose(A, B)$ that returns (B', A'), a reordering

of operations A and B such that executing A and B in sequence has the same effect as executing B' and A' in sequence. Also, B' must be the transformed operation that should have been executed by the editor if A had not been executed earlier.

The notion of conflict is just a formal way of capturing the requirement that the operations should not be reordered because of semantic dependencies — typically the operations modify the same objects or region in the document. For instance, if operation A inserted a string and operation B modified the inserted string, there would be a conflict between the two operations. Also, note that the Transpose function above applies to operations that have already been executed with correct results, unlike the transformation function for the Transformation matrix, which is used to determine the operation to execute for getting correct results.

If an operation A is undone, we assume that the users want their document to go to a state that it would have gone to if operation A had never been performed, but all the following non-conflicting 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. Let's assume that $Transpose(A, B) = (B', A')$. Therefore, by the definition of the Transpose function, if A had never been performed, the system would have performed operation B' in place of B. Therefore, after undoing A, the selective undo algorithm should result in the document's state being as if only B' had been performed in state S.

The basic idea behind the algorithm is to *shift* the operation to be undone to the end of the history list by transposing it with subsequent operations. If the operation cannot be shifted to the end of the list owing to a conflict, then the operation cannot be undone without also undoing the conflicting operation. If the operation can be shifted to the end, then it can be undone by simply executing its inverse. As an example, suppose 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 of A will fail, as it should, because A cannot be undone unless B is also undone. If A does not conflict with B, the result after one iteration of shifting 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 the definition of the Transpose function 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:

$$B'\ C'.$$

Performing $\overline{A''}$ in the present state therefore correctly cancels A, giving the same document state as executing B' and C' in the original state — the operations that would have executed had A never been performed; the undo has succeeded.

The Transpose and Inverse functions need to satisfy several formal properties for a correct undo algorithm. For more details on the properties the reader is referred to [Pra94a]. The paper also presents a generalization of the above scheme to handling undo of previously undone operations and undo of operations restricted to a region. Below, we only illustrate the Conflict and Transpose functions that would be defined for Examples 2 and 3 and the resulting behavior on undo.

In Example 2, $Conflict(Operation1, Operation2)$ is best declared to be true because they both change the same attribute of the circle; also there is no simple way to reorder the two operations with the same resulting effect on the state and satisfying all the properties that are given in [Pra94a]. Thus, $Operation1$ cannot be undone by this algorithm without also undoing $Operation2$. An implementation can either report a conflict error, undo both operations automatically, or give an option to the user to either undo both operations or to leave the document state unchanged, in view of the subsequent change by another user.

In Example 3, operations 1 and 2 need not be declared to conflict. The history list would contain:

$$InsChar(1, x)\ InsChar(5, y).$$

To undo the first operation, the algorithm would shift it to the end of the list by (temporarily) reordering the list as follows:

$$InsChar(6, e)\ InsChar(1, c))$$

It will then execute its inverse, DelChar(1). This results in the string *omputer*, the intended result that deletes e from the first position, leaving the effect of the second operation in the correct place.

5.5.4 Undo–Redo Strategy with Transformations

Another possible algorithm for selective undo is to use the undo/redo strategy for selective undo, augmented with transformations. This does not appear to have been described in the literature, so we only sketch the ideas here. To undo an operation A, one can assume that its inverse $\overline{A}$ is a late arriving operation that should have executed immediately after A. To execute $\overline{A}$, we can reverse all the operations that were done after A, then execute $\overline{A}$, and then redo the operations after A after transforming all the operations, using the Transformation Matrix described in Section 5.4.3.

5.5.5 Relation of Concurrency Control and Undo

An interesting question is whether the choice of concurrency control algorithm and the undo algorithm are dependent. There are some obvious similarities between the schemes, such as the use of transformations, undo/redo, etc. There are also some differences. Undo, like other operations, should behave identically at all sites in a replicated architecture and provide intended results.

Ensuring consistent behavior of the undo operations can be challenging for several reasons. First, the history list may not be identical at each site, particularly if the transformation-based scheme illustrated in Figure 5.8 is used. In Figure 5.8, undoing the command B at site 1 may not necessarily have the same effect as undoing the command B' at site 2, unless additional requirements are placed on the transformation matrix. And, in general, it is not clear if it

is always possible to find a transformation matrix that satisfies the requirements for both consistency and undo. An elegant discussion of the properties that transformations need to satisfy so that operations can be selectively undone can be found in the work by Ressel et al [Res96].

Second, conflicting operations may be issued in parallel and the system may pick an execution order that achieves consistency but makes it later difficult to undo one of the operations with reasonable results.

Third, with optimistic concurrency control, the undo command itself may be issued and executed optimistically in an inconsistent state; it is not obvious what operation, if any, should be undone by an undo command that is issued from an inconsistent state.

In DistEdit [Kni93, Pra94a], some of these problems are addressed as follows. First, the undo commands are not broadcast, only the operation executed. Thus, even if history lists are not identical (but equivalent), data consistency is maintained. Second, locks are used so that only non-conflicting operations are allowed to be executed in parallel. Third, because of pessimistic concurrency control, the undo commands can only be issued in consistent states.

In [Cho95], several other issues in the design of an undo framework are considered, including the problem of undoing commands that are executed only at a subset of sites and undoing commands that affect more than one site.

5.6 SUPPORTING COLLABORATION AWARENESS

Collaboration awareness features can be critical in a group editor in order to provide better context regarding the environment in which collaborative activity is taking place. Below we look at examples of collaboration awareness features from various group editors. Additional examples can be found in Section 6.4 (page 150) and some implementation issues can be found in Section 7.4.2 (page 177) in this book [Gre99, Dew99].

5.6.1 Participant Context

Group editors often display the list of users in the group editing session, so as to provide context regarding the participants when group editing is not face-to-face. Examples of this can be found in the user interface provided by DistEdit [Kni90] and by Dolphin [Str94] (Figure 5.1). The list of participants is updated as participants leave or join a session.

The list of participants can be useful in several ways. It can be used to allow users to send electronic mail to individual participants (for example, by clicking on their name or icon). It can also be used to give additional information about the participants — such as the contact information from their business card, and their role in the session (observer, participant). Some recent systems, such as Habanero from NCSA, use the list of participants in the above ways.

Keeping the list automatically updated in the presence of network failures requires some support from the underlying communication system. In particular, if a user's editor crashes or the connection to the editor is lost, other editors need to be able to determine that and drop the user from the list. It is well known, though, that distinguishing a crash or lost connectivity from a very slow connection is not possible in typical networks. The standard solution in such cases is to drop a very slow connection, treating the editor at the end of the connection as effectively being out of the collaboration session. If the user's editor later attempts to communicate, it is forced to rejoin the session as a new member.

One difficulty with providing participant context is that showing a user as a member in a membership list normally only shows that the user has the group editor open. It does not guarantee that the user is paying attention to the group editing session. In face-to-face meetings, eye contact and other bodily cues indicate whether a particular participant is paying attention.

Several potential solutions exist or are being tried out to provide more information than just membership lists. One solution is to show idle time for each user — the period for which they have not interacted with the group editor. This is not a perfect indicator either because it could be that a user is idle but is paying attention; or perhaps the user could have stopped paying attention very recently.

Another solution is to use additional media, such as video, to provide awareness (see also Chapters 3 and 4 in this book [Mac99, Ish99]). Use of video can provide relevant participant context more rapidly than idle time. However, this solution also has limitations: 1) bandwidth and computing cycles may be limited to provide good quality video; 2) screen real-estate can be an issue if the group consists of more than two people; and 3) video is potentially more invasive of privacy than other solutions.

In general, providing good participant context in a non-obtrusive way and in a scalable manner is an open research problem. A more in-depth discussion of various aspects of awareness can be found in [Ben93, Rod96, Tol96].

5.6.2 View Context

People often find it natural to use references such as "top-line of the window" or "the node in the top right corner" to refer to objects being edited. Unfortunately, such references can be confusing in a group editing environment if users do not have their windows or views of the document synchronized. Many group editors, thus, usually attempt to provide an ability to synchronize their views of the document.

Group editors, however, differ in the extent to which they provide synchronized views. In DistEdit-based text editors, for instance, support is available for synchronization of cursor positions and highlighted selections, but no support is provided for synchronization of window sizes, position of lines within a window, etc.

In Suite [Dew91], facilities are provided for closer synchronization of views, including selection of fonts, scrolling, etc. — application designers are provided substantial controls on the editor attributes that they wish to synchronize, but the programmers must implement the attributes.

Supporting view synchronization is usually done by introducing additional shared state variables, besides the document itself. As an example, for implementing synchronized scrolling, the position of the scrollbar can be made a shared variable, with a copy at each site, and updated with the concurrency control techniques described earlier. Response time can be even more critical for operations such as scrolling that update views — since users expect browsing to be fast — so judicious use of concurrency control techniques is essential.

The above strategy of using shared state variables to capture the view state can sometimes be non-trivial to use for programmers. Graphical views of documents often consist of multiple user-interface widgets (e.g. scrollbars, windows, buttons, canvas), each of which can require a large number of state variables to represent completely. For example, state variables required to share the visual representation of a simple button can include its label, font of the text, size of the button, its shape, whether it is active or disabled, etc. It can, in general, be quite tedious for programmers to determine what state variables are required to be shared for a particular

widget and then doing the programming to keep the state variables consistent with each other and with the visual state of the button.

In DistView-based groupware tools [Pra94b] on the NeXT systems, the task for implementing synchronized views is considerably simplified. *Groupware-enabled widgets* corresponding to each of the standard GUI widgets, but with built-in replication support using state variables, are provided by extending the standard set of NextStep widgets. These groupware-enabled widgets are made available in the NextStep's Interface Builder so that users can build applications with window replication and sharing using the standard NextStep's drag-and-drop graphical environment. More recently, a Java-based version of DistView is attempting to provide a similar drag-and-drop functionality for building groupware applications using the Java Beans component model.

5.6.3 Activity Context

While using a group editor on a large document, a participant may need to know the regions of the document that other participants are working on. Such information can help avoid conflicting work and facilitate interactions. SASSE [Bae93] (and its earlier version SASE) are examples of group editors that provided this information particularly well. In SASE, continuous feedback was provided to users about other collaborator's working locations in the document with color-coded text selections and multiple scrollbars (one per user). In SASSE, multiple scrollbars were replaced by two scrollbars in order to save screen real-estate: the normal scrollbar of the local user and another scrollbar with multiple color-coded indicators to show the locations of other users.

Activity context is also often provided by the use of audioconferencing tools or multi-user chat tools. This additional conferencing channel can be used by users to coordinate or discuss the document contents while it is being edited. Figure 5.1 shows the MBone audioconferencing tool, *vat*, being used along with the Dolphin group editor.

Use of other generic tools for communication to provide activity context is an attractive strategy because such tools can be useful for a variety of group editing environments. A key challenge however is providing a seamless integration among these tools and the group editing system. If each of the tools and the group editor requires its own set-up and provides its own interface for joining/leaving a group session, then the system can become tedious to use. Several strategies for integrating multiple tools seamlessly in a single system are centered around the *room* metaphor; participants join an editing session by entering a room. The rooms contains various tools, such as editors, chat, and audio, and all these automatically become available to a new participant upon entering the room. Systems that integrate multiple groupware tools based on the room metaphor include wOrlds [Tol95], Collaboratory Builder's Environment [Lee96], and TeamRooms [Ros96b].

One significant challenge with providing activity context is determining what is the appropriate context that users need. A system could show to each user what everyone else is looking at. However, that has screen real-estate implications (besides privacy concerns which we ignore here). In addition, it could probably overload users with too much unnecessary context information.

In general, there is a tradeoff between the extent of common view context and the extent of need for activity context. If views of the document for all users are synchronized, then less activity context may be needed — actions of one user are going to be visible to all other users because of view synchronization. On the other hand, if views are not synchronized, actions

of one user may not be visible to other users. Thus, more *a priori* and on-going coordination among users may be needed in order to avoid conflicting or overlapping work.

5.6.4 Telepointing

Telepointing can be a useful collaboration awareness feature in group editors that provide synchronized views. In telepointing, a user's mouse movements can be tracked by the system and displayed on everyone's synchronized window.

Different editors provide varying levels of telepointing capability. DistEdit-based text editors only provide synchronized cursor capability. A user's cursor is tracked by cursors of other users when they are in a *lockstep* editing mode. Selections of regions of text are also tracked. No mouse-based telepointing is supported, primarily because close synchronization of views in a window is not supported owing to heterogeneity of the user-interfaces and platforms of DistEdit-based editors. Mouse-based telepointing makes little sense unless the pointer can be displayed in the same position with respect to the data being viewed in all the windows.

Editors such as Dolphin and SASSE support mouse-based telepointers. Both systems use a standard underlying platform (Smalltalk in Dolphin's case and Macintosh in SASSE's case) so that they are able to provide *group windows* that are identically-sized and have the same contents to all the users. This facilitates displaying a mouse-based telepointer at the same position with respect to the data in the group window.

Supporting multiple telepointers can also be useful, with a different telepointer assigned to each user [Hay94]. If multiple telepointers are provided, they should be assigned different colors or shapes so that users can identify who is manipulating a particular telepointer.

The main challenge with implementing telepointers is dealing with performance. Moving a pointer can generate a large number of mouse-move events. To reproduce the pointer movement at other sites with low latency, these events have to be broadcast over the network *as they are generated*. In low-bandwidth situations, the originating site can potentially be slowed down by the network bottleneck, leading to jerky mouse movement at the originating site. The movement of telepointers at receivers can also be unsatisfactory because of jitter introduced by the network in delivering the broadcast messages. The behavior can be worse than trying to use a window system such as X over a slow network.

Potential strategies to deal with the above performance problems include only broadcasting a subset of mouse-move events. Recent studies show that transmitting ten mouse-events per second is usually adequate to get continuity in pointer movements [Ste96]. With judicious sampling of pointer movements, by using non-blocking protocols, and by incremental painting of screen when remote pointers move, group editors can be designed to support telepointers adequately, even in low-bandwidth situations.

5.7 DESIGN OF DOCUMENT STRUCTURE

Some group editors have focused on not just supporting simultaneous editing of documents, but on the design of document structures that support collaboration activities such as brainstorming. The assumption is that brainstorming is a major use of group editors, and thus group editors need to provide appropriate document structures to support brainstorming.

The simplest kind of group editors to support brainstorming are simply little more than group drawing editors. They provide a graphical canvas on which users can draw shapes (such

as rectangles, arrows, lines, etc.) to represent objects of conversation, type in text for labeling, and use one or more telepointers to draw attention to objects represented on the canvas. The shared whiteboard tools such as MBone's *wb*, Cooltalk in Netscape 3.0, and several public-domain programs are examples of such editors.

A much richer document structure for group editing is provided by the Dolphin system [Str94], whose interface is shown in Figure 5.1. Dolphin provides a *hypermedia* document structure, in which users can not only draw shapes and type in text, but also create *nodes* and *links*, where nodes can represent substructures within a document and links can be used to jump from one part of a document to another related part of the document. Recent experiments with Dolphin have shown that such hypermedia-based document structure can lead to more effective brainstorming than the standard shared whiteboard tools [Str94].

Supporting richer document structures is facilitated by richer support for object replication because a document may be represented using a large number of objects with distinct types (e.g. nodes, links, text, graphics), and not all objects may be shared among the entire group at a given time. The use of object replication for view synchronization is discussed in Section 5.8.3. In the case of Dolphin, the COAST system [Sch96] provides the necessary object replication support.

5.8 OTHER DESIGN ISSUES

5.8.1 File Management

Several problems arise when users share document files in order to do group editing. First, when a user requests a file be opened for editing, a group editor must determine whether anyone else is currently editing that file and, if so, load from the active group session rather than from the file. Second, a user should not be allowed greater editing access rights using a group editor than the file system would allow. Third, care must be taken should several users attempt to save a shared file at the same time.

In determining whether several users wish to edit the same particular file, it is not possible to simply examine the path names of the files; because of network file systems, a file can potentially be referenced by different paths. In DistEdit, this problem is solved as follows. When a user attempts to open a file from within an editor, DistEdit searches in the directory containing the file for an auxiliary file of the same name prefixed by '#de.'. For instance, when opening */aprakash/de/testfile*, DistEdit will search for the auxiliary file *#de.testfile* in the directory */aprakash/de*. This auxiliary file contains a unique identifier to be used as the group session name for the particular file. If no such file exists, DistEdit creates it so other users will be able to join the session. If the file exists, DistEdit attempts to join the session identified in the file.

Another solution to the file path problem is for the group editor to provide its own small file-server where files belonging to groups are stored. The group editor can then ensure that their file system presents a common view of files to all users. Several PC-based editors, such as ShrEdit [McG92], use this solution, primarily because network file systems were not commonly available in PC environments.

A group editor needs to be designed to enforce access control. Enforcing access control is more important than in single-user editors because a user can potentially modify not only his own documents, but also documents owned by other users in the same group session. The

access control permissions may be inherited from the file system or the group editor can be designed to provide its own access control policies.

The normal file save routines of single-user editors can basically be used as-is in group editors. There is, however, a potential problem. If multiple users were to save slightly different versions (due to network message latency) at approximately the same time, care has to be taken that the resulting file saved is not corrupted owing to parallel save operations.

5.8.2 Screen Updates

The screen update code in group editors needs to be carefully designed with the following goals:

- *Minimize full screen redraws on updates:* Full screen redraw of the user-interface is the simplest strategy for displaying updates to the document, but can be annoying to users. We experienced this problem when converting *xedit* to a group editor using the DistEdit toolkit. Xedit's screen update code handled local updates well, but remote updates caused flickering because an assumption was made in the original Xedit code that updates can only occur at the user's cursor position. Changing the cursor position, applying a remote update, restoring the cursor position, and then displaying the screen caused flickering. Xedit's assumption was acceptable for a single-user version of Xedit, but caused us problems when making it a group editor. A similar problem occurs in most shared-X systems when *expose* events on one client's window cause the X server to redisplay the windows of everyone in the group [Abd91]. We did not have this problem with converting Emacs to a group editor using DistEdit, primarily because the single-user version of Emacs was already well-designed to handle updates from multiple windows into the same document buffer.

- *Manage cursor/pointer position and scrolling:* To the extent possible, applying updates from remote sites should not cause a user's cursor/pointer position to change or the display to scroll. If a user's window starts scrolling because of updates in earlier parts of a document by other users, the user is likely to find the behavior annoying — especially if the document parts that are changing are not even in the user's display.

5.8.3 Use of Object-Replication

The shared state of some group editors can sometimes be naturally represented using multiple encapsulated objects such that not all objects are necessarily of interest to all users. Toolkits such as DistView [Pra94b] and COAST [Sch96] provide support for managing multiple replicated objects in such editors. These toolkits require applications to be built using techniques similar to the Model–View–Controller paradigm; the application consists of *model* objects and *view* objects. Model objects represent the underlying application data, such as the document state, that must always be kept consistent at all sites. View objects usually correspond to the visual representation of the model objects using user-interface widgets, such as windows, scrollbars, etc. Using these toolkits, an application, such as an editor, can provide both synchronization of document state and that of the views of that state. To allow simultaneous work on the same model object, the model object is replicated at the various sites. If a subset of users wish to get identical views of model objects in their windows, they can also replicate a view object so that their views are consistent.

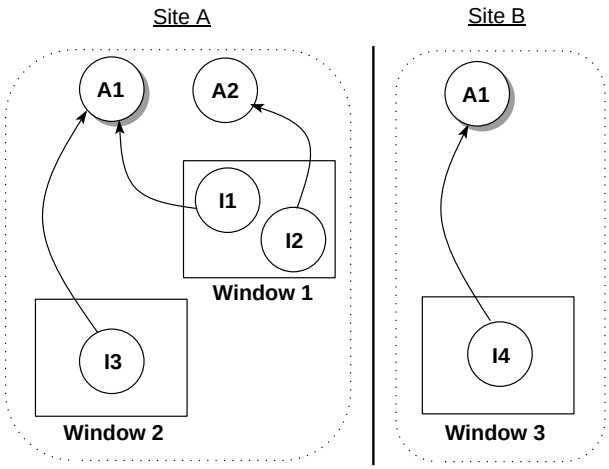

Figure 5.10 The state of a multi-user application at sites A and B before sharing of Window 1 in a DistView-based application. $A1$ and $A2$ are application objects, and $I1$, $I2$, $I3$, and $I4$ are interface objects

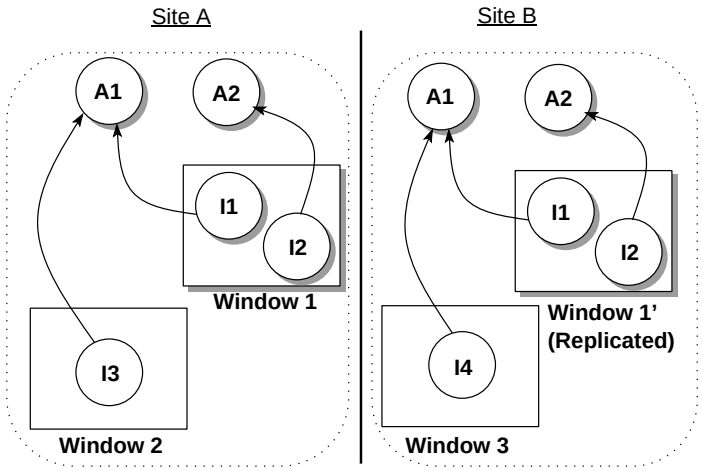

Figure 5.11 The state of the multi-user application at sites A and B after the user at site B imports Window 1 in the DistView-based application. Appropriate objects necessary for sharing windows efficiently are replicated and maintained consistent on subsequent updates

Figures 5.10 and 5.11 illustrate the use of an object replication infrastructure in DistView to facilitate synchronized views of the underlying document. Figure 5.10 shows the objects at two sites when they are not sharing any window (though they are sharing data corresponding to object $A1$). Figure 5.11 shows the objects that are automatically replicated upon demand after Window 1 is *exported* by the user at Site A and *imported* by the user at site B for exact view synchronization. Concurrency control techniques, discussed in Section 5.4, are then used to keep object copies consistent as windows/data are interacted with by the users.

5.8.4 Transactions

Concurrency control, transformations, and undo algorithms are greatly simplified if a group editor uses only a small set of core update primitives. However, any other editing action to be provided by an editor then must be mapped to a sequence of these operations.

In single-user editors, treating a group of simple operations as one larger, user-level action is important primarily for implementing undo; a user, upon doing an undo operation, usually expects all the changes associated with the last single-user-level action to be undone, rather than just some of them.

In a group editor, grouping operations into a larger action is also important for undo. In addition, it is an important issue from the perspective of atomicity because users may expect the operation to behave as a single atomic (indivisible) action even when concurrent updates are being applied by other users.

Supporting the undo of multi-operation actions requires that operations on the history list be tagged with a *transaction-id* so that all operations belonging to a transaction can be undone together. DistEdit uses this scheme. An issue remains as to whether to allow partial undo of a transaction when a complete undo is not possible (say, due to inability to acquire locks or due to conflicts with subsequent updates), but this is largely a policy issue and either choice can be implemented in a straightforward manner.

Supporting atomicity of actions in group editors requires addressing several problems. One problem is that transaction boundaries can sometimes be difficult to determine. The simplest solution is to treat each user's interaction that generates an update as a request for a transaction. However, consider a case where a user is doing a free-form drawing by pressing and moving a mouse. In such a case, the updates are generated continuously on every mouse-move event. However, the user may reasonably expect the entire action of drawing while the mouse is pressed to be a single action from the perspective of undo and atomicity. Another example is an interactive global find and replace string operation in a text editor. Find/replace commands are generated between every interaction, but the user may expect the entire sequence to be a single action for undo purposes. A good design principle is to normally treat each user's interaction that generates a command as a transaction, but if a different choice of transaction boundaries is made for the purpose of undo, to use the same choice for the purpose of defining atomicity.

Another problem in implementing atomicity is that the operations that constitute the transaction may have to be executed before the transaction is complete, for example, in order to provide feedback to a user who is doing free-form drawing in a graphical document or a text-search and replace operation throughout a text document. This can be a problem because the editor may not be able to determine in advance whether the transaction will successfully complete. An operation in a transaction could fail owing to the inability to acquire locks, for instance. A solution to the problem is to execute the transaction optimistically on the local copy first, determine all the locks that are needed as the transaction is executed, and then do a local undo in the normally rare case that the transaction fails for concurrency control reasons [Kni90]. The undo required for this is simpler than a group undo because it applies only to the user's local copy of the document.

5.9 FUTURE WORK

Group editors are likely to continue to evolve in the future. Many of the basic concepts in the design of group editors, such as doing concurrency control and undo, and providing collaboration awareness, have been explored in various editors. However, they have usually been explored in different editors, many of which are research prototypes. Group editors still have not yet evolved to the point where any single group editor provides all the features described in this chapter. Nevertheless, successful use has been reported from even prototype group editors, which indicates that group editors can become widely used, once they are available as standard tools on common computing platforms and sufficiently robust for everyday use.

Group editors need to better support both synchronous and asynchronous collaboration in the future. The support for persistence of shared objects and sessions in the DistView/CBE system [Lee96] and in the TeamRooms system [Ros96b] helps support asynchronous activity. For asynchronous collaboration, additional collaboration awareness may need to be provided to late joiners, so that they know what activities have taken place since they last participated. Providing support for on-line session recording and replay [Man95] are steps in that direction, but need to be abstracted out so that users can get a higher-level context about the work they have missed.

We believe that group editors will be much better integrated with other tools used by users. In particular, group collaboration environments are likely to consist of not only group editors, but also other groupware-oriented versions of applications such as Internet browsers, audio/video/text conferencing tools, data visualization tools, etc. The various groupware tools need to be provided in an integrated environment with a shared session management, global context information, and a seamless transfer of information from one tool to another. We are beginning to see some trends in that direction in several recent university projects such as wOrlds [Tol95], Collaboratory Builder's Environment at the University of Michigan [Lee96], and the GroupKit and TeamRooms work at the University of Calgary [Ros96a, Ros96b]. Commercial efforts by Microsoft and Netscape to integrate groupware tools in their browsers, by IBM/Lotus to extend Lotus Notes to support synchronous collaboration, and by JavaSoft to provide a collaboration toolkit based on Java, are steps in a similar direction.

REFERENCES

[Abd91] Abdel-Wahab, H.M. and Feit, M.A., XTV: A framework for sharing X window clients in remote synchronous collaboration. In *Proceedings, IEEE Tricomm '91: Communications for Distributed Applications and Systems*, April 1991.

[Abo92] Abowd, G. and Dix, A., Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.

[Bae93] Baecker, R.M., Nastos, D., Posner, I.R. and Mawby, K.L., The user-centered iterative design of collaborative software. In *INTERCHI'93 Conference Proceedings*, pages 399–405. Addison-Wesley, 1993.

[Ben93] Benford, S.D. and Fahlén, L.E., A spatial model of interaction in large virtual environments. In *Proceedings of the European Conference on Computer-Supported Cooperative Work (EC-SCW'93)*, pages 109–124. Kluwer, 1993.

[Ber94] Berlage, T., A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Computer-Human Interaction*, 1(3):269–294, 1994.

[Bir87] Birman, K.P. and Joseph, T.A., Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, pages 47–76, February 1987.

[Bir90] Birman, K. et al, *The ISIS System Manual, Version 2.0*, April 1990.

[Cha84] Chang, J.M. and Maxemchuck, N.F., Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Aug. 1984.

[Cho95] Choudhary, R. and Dewan, P., A general multi-user undo/redo model. In *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work*, pages 231–246. Kluwer Academic Publishers, September 1995.

[Dew91] Dewan, P., Flexible user interface coupling in collaborative systems. In *Proceedings of the ACM CHI'91 Conference on Human Factors in Computing Systems*, pages 41–48, April 1991.

[Dew99] Dewan, P., Architectures for collaborative applications. In Beaudouin-Lafon, M. (Ed.), *Computer Supported Cooperative Work*, Trends in Software Series 7:169–193. John Wiley & Sons, Chichester, 1999.

[Ell88] Ellis, C., Gibbs, S.J. and Rein, R., Design and use of a group editor. In G. Cockton (Ed.), *Engineering for Human–Computer Interaction*, pages 13–25. North-Holland, Amsterdam, September 1988.

[Ell89] Ellis, C., Gibbs, S.J. and Rein, R., Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD '89 Conference on Management of Data*, pages 399–407. ACM Press, 1989.

[Fis88] Fish, R., Kraut, R., Leland, M. and Cohen, M., Quilt: A collaborative tool for cooperative writing. In *Proceedings of ACM SIGOIS Conference*, pages 30–37, 1988.

[Fol74] Foley, J.D. and Wallace, V.L., The art of natural graphical man–machine conversion. *Proceedings of the IEEE*, 62(4):4622–471, April 1974.

[Gre94] Greenberg, S. and Marwood, D., Real-time groupware as a distributed system: concurrency control and its effect on the interface. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 207–217, 1994.

[Gre99] Greenberg,S. and Roseman, M., Groupware toolkits for synchronous work. In Beaudouin-Lafon, M. (Ed.), *Computer Supported Cooperative Work*, Trends in Software Series 7:135–168. John Wiley & Sons, Chichester, 1999.

[Gri76] Grief, I., Seliger, R. and Weihl, W., Atomic data abstractions in a distributed collaborative editing system. In *Proceedings of the 13th Annual Symposium on Principles of Programming Languages*, pages 160–172, 1976.

[Han71] Hansen, W.J., User engineering principles for interactive systems. In *AFIPS Conference Proceedings*, Vol. 39, pages 523–532. AFIPS Press, 1971.

[Hay94] Hayne, S., Pendergast, M. and S. Greenberg, S., Implementing gesturing with cursors in group support systems. *Journal of Management Information Systems*, 10(3):43–61, 1994.

[Ish99] Ishii, H., Integration of shared workspace and interpersonal space for remote collaboration. In Beaudouin-Lafon, M. (Ed.), *Computer Supported Cooperative Work*, Trends in Software Series 7:83–102. John Wiley & Sons, Chichester, 1999.

[Kar93] Karsenty, A. and Beaudouin-Lafon, M., An algorithm for distributed groupware applications. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 195–202. IEEE Press, 1993.

[Kni90] Knister, M. and Prakash, A., DistEdit: A distributed toolkit for supporting multiple group editors. In *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, pages 343–355, Los Angeles, California, October 1990.

[Kni93] Knister, M. and Prakash, A., 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.

[Lee96] Lee, J.H., Prakash, A., Jaeger, T. and Wu, G., Supporting multi-user, multi-applet workspaces in CBE. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 344–353, 1996.

[Mac99] Mackay, W.E., Media Spaces: Environments for informal multimedia interaction In Beaudouin-Lafon, M. (Ed.), *Computer Supported Cooperative Work*, Trends in Software Series 7:55–82. John Wiley & Sons, Chichester, 1999.

[Man95] Manohar N.R. and Prakash, A., The session capture and replay paradigm for asynchronous collaboration. In *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work*, pages 149–164. Kluwer Academic Publishers, September 1995.

[McG92] McGuffin, L. and M. Olson, G., ShrEdit: A shared electronic workspace. Technical Report CSMIL No. 45, University of Michigan, Ann Arbor, 1992.

[Neu90] Neuwirth, C.M., Kaufer, D.S., Chandhok, R. and Morris, J.H., 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.

[New91] Newman-Wolfe, R.E. and Pelimuhandiram, H.K., 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.

[Pra92] Prakash, A. and Knister, M., Undoing actions in collaborative work. In *Proceedings of the Fourth ACM Conference on Computer-Supported Cooperative Work*, pages 273–280, Toronto, Canada, October 1992.

[Pra94a] Prakash, A. and Knister, M., A framework for undoing actions in collaborative work. *ACM Transactions on Computer–Human Interaction*, 1(4):295–330, December 1994.

[Pra94b] Prakash, A., and Shim, H., DistView: Support for building efficient collaborative applications using replicated objects. In *Proceedings of the Fifth Conference on Computer Supported Cooperative Work*, pages 153–164, Toronto, Canada, October 1994. ACM Press.

[Rei91] Rein, G.L. and Ellis, C.A., rIBIS: A real-time group hypertext system. *International Journal of Man–Machine Studies*, 34(3): 349–367, 1991.

[Res96] Ressel, M., Nitsche-Ruhland, D. and Gunzenhäuser, R., An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 228–297, 1996.

[Rod96] Rodden, T., Populating the application: A model of awareness for cooperative applications. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 87–96, 1996.

[Ros96a] Roseman, M. and Greenberg, S., Building real time groupware with GroupKit, a groupware toolkit. *ACM Transactions on Computer–Human Interaction*, 3(1):66–106, March 1996.

[Ros96b] Roseman, M. and Greenberg, S., TeamRooms: Network places for collaboration. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 325–333, 1996.

[Sch96] Schuckmann, C., Kirchner, L., Schümmer, J. and Haake, J.M., Designing object-oriented synchronous groupware with COAST. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 30–38, 1996.

[Ste96] Steinmetz, R., Human perception of jitter and media synchronization. *IEEE Journal of Selected Areas in Communications*, 14(1):61–72, January 1996.

[Str94] Streitz, N.A., Geißler, J., Haake, J.M. and Hol, J., DOLPHIN: Integrated meeting support across local and remote desktop environments and liveboards. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 345–357, Chapel Hill, North Carolina, October 1994.

[Str97] Strom, R., Banavar, G., Miller, K., Prakash, A. and Ward, M., Concurrency control and view notification algorithms for collaborative replicated objects. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 194–204, Baltimore, MD, USA. IEEE Computer Society Press, 1997.

[Tol95] Tolone, W., Kaplan, S. and Fitzpatrick, G., Specifying dynamic support for collaborative work within wOrlds. In *Proceedings of the 1995 Conference on Organizational Computing Systems*, pages 55–65, August 1995.

[Tol96] Tollmar, K., Sandor, O. and Schömer, A., Supporting social awareness @work: Design and experience. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, pages 298–307, 1996.

[Vit84] Vitter, J.S., US&R: A new framework for redoing. *IEEE Software*, pages 39–52, October 1984.