Giving undo attention

Gregory D Abowd & Alan J Dix *
HCI Group, Dept. of Computer Science
University of York
Heslington, YORK YO1 5DD
UK

Abstract

In this paper, we investigate the problems associated with the provision of an undo support facility in the context of a synchronous shared or group editor. Previous work on the development of formal models of undo has been restricted to single user systems and has focused on the functionality of undo, as opposed to discussing the support that users require from any error recovery facility. Motivated by new issues that arise in the context of computer supported cooperative work, we aim to integrate formal modelling of undo with an analysis of how users understand undo facilities. Together, these combined perspectives of the system and user lead to concrete design advice for implementing an undo facility. The special issues that arise in the context of shared undo also shed light on the emphasis that should be placed on even single user undo. In particular, we come to regard undo not as a system command to be implemented, but as a user intention to be *supported* by the system.

Keywords: *Undo support, CSCW, formal methods*

1 Introduction

In this paper we discuss the issue of design options for undo in a group editor, both in terms of what is wanted by the users and what is meaningful and possible for the system to provide. We explore the extent to which the system supports the users' intention for undo within the wider context of recovery.

The importance and problem of undo

Few people would argue about the importance of undo. As Yang [29] points out:

*Gregory Abowd is a Research Associate with the HCI Group and Dependable Computing Systems Centre at York. Alan Dix is funded by a SERC Advanced Research Fellowship B/89/ITA/220 with the HCI Group at York.

> "Most sophisticated interface systems should be provided with an undo support facility."

Similar quotes can be found in the literature dating back at least 10 years; indeed, Shneiderman [24] regards the ability to easily and incrementally undo one's actions as a characteristic of direct manipulation systems which makes them so appealing. Despite the general agreement of the importance of undo, few systems supply more than the simplest single-step undo command and, even then, the effect of command and when it can be applied is often far from obvious. In a recent study of the use of Microsoft Word [27], it was found that even expert users were unable to both predict the effect of the undo command or recognise its behaviour. Undo support for single user systems is regarded as essential, but recognised to be fraught with potential pitfalls.

Users who have come to expect effective undo support in their single user applications will carry forward this expectation to their groupware. Extending single user notions of undo into a multi-user context raises many new issues and even calls for a re-examination of the understanding of conventional single-user undo. However, the authors have only found one brief reference to this issue in the CSCW literature [22]. We therefore believe that multi-party undo is an area which is urgently due attention.

Undo—a system function?

Most analyses of undo consider it as a function, that is, the system undoes the user's last action in response to the undo command invoked by a keyboard button press or a menu selection. From a computing perspective, the system function invoked by the user's command can be extended to look at repeated undo or redoing previously undone commands. There are also questions about how to efficiently implement undo—which is why most systems supply undo in such a restricted form. Even psychologists, who take a more user-centred approach to undo analysis, still remain confined to the idea of undo as a system function. Examples of such psychological analysis are the above-mentioned study of undo in Microsoft Word [27] and the knowledge analysis of Young and Whittington [30]. which has also been applied to the question of multi-user undo [21].

Our analysis will take this dominant theme of undo as a system function as far as it will meaningfully go in the multi-user case. Even though we are able to suggest principles for the support of the multi-user undo function, the issues of group undo necessarily lead us to rethink this stance. Rather than ask, "What does the undo button do?", we instead ask "What is the undo button for?" The former question leads to an analysis of undo as a system function, but the latter to an interpretation of undo as a *user intention*. Th user wants to recover from a recognised erroneous state and may achieve this intention by use of any system function, not necessarily the undo function. This perspective suggests that designers must focus on supplying tools to facilitate this recovery.

Overview

In Section 2, we will describe a scenario for the design of a multi-user undo for a shared text editor, summarize the results of a knowledge analysis of undo, define the scope of the problem we are considering and the particular formal approach we adopt in our analysis.

Our analysis in Sections 3–5 will push the 'undo as a function' perspective as far as possible. Section 3 will look at existing models of single user undo and how they extend to the multi-user case. It is here that the conflict arises between the user preference for local undo and the system preference for global undo. A more formal discussion in Section 4 will characterise conditions under which the conflicting system and user preferences for undo can be reconciled. Section 5 will then discuss more concrete mechanisms which a designer can use to satisfy local user undo with a global system undo. Unfortunately, the majority of these mechanisms work only be decreasing the possibility of interference between users, which in the context of the shared editor means reducing the possibility for cooperation.

Though we do present some more fine-grained mechanisms in Section 5 for resolving conflict whilst maintaining cooperation, there still remain situations in which the system cannot provide a sensible interpretation for local undo. The only person who knows what to do is the user. Given this, Section 6 takes a look at the more general issue of recovery, in which undo is but one form. We conclude that recovery is more properly understood as a user intent and not a system function, and we discuss mechanisms which can be used to support that intention.

Section 7 summarizes the recommendations for undo in a group editor, both as a function and as user support.

2 The problem

Scenario

A team of designers are developing a group text editor intended for cooperative document preparation. Early versions of the editor allow for only one insertion point into the shared document, so that simultaneous editing requires the users to "pass" control of the insertion point between them. It is possible that future versions of the system will allow multiple insertion points so that truly concurrent and synchronous editing can occur. The designers feel it is important to provide support for undoing, so that the users could reverse the effects of the most recent editing operations if they produced undesirable effects on the shared document.

They are not sure, however, how that undo facility should work. Some feel it should work relative to the document, that is, allowing one user to undo the most recent operation regardless of whether they were the initiators of that operation. The others believe that a single user should only be able to undo operations that they themselves initiated. None of the designers can appeal to anything other than intuition as to which is the best solution, and in this case there is no consensus. The designers recognise that an undo facility is not something that can be developed as an add-on component to an existing system but rather requires consideration from the very early stages of design. Therefore, developing systems which manifest the two undo schemes implies the parallel development of two systems, and there are not the resources for such a luxury. In short, they are unable to give undo attention.

The user's perspective

A reasonable requirement for the construction of an undo support facility is an understanding of what a user must know about undo in order to make effective use of it. A *knowledge analysis* of undo by Young and Whittington [30, 21] highlights four questions the user must be able to answer about undo:

1. *What stream of activity is relevant?* For a single user system, this is normally the user's stream of actions. For multi-user systems we must also consider whether the stream is the user's own actions or the intermingled stream of all the users' actions.

2. *How is the stream articulated into units?* As suggested by the language model of Foley and van Dam [12], we may consider the relevant stream of activity at the lexical level (mouse clicks, key presses, etc.), the syntactic level (gestures or command strings), or the semantic level (operations on the underlying data structure). Even so, determining the articulated units may be complicated by a macro facility in which the user is able to tailor the units of activity in a way that may not be recognised by the undo function.

3. *Which unit is affected by an undo?* The user must also know which articulated unit is relevant at any given invocation of undo. For example, the undo may operate at the level of individual user actions, but may undo only the last such destructive action.

4. *What is the definition of undo?* How is the underlying data structure affected by the undo and is this apparent to the user? For example, a selected block of text is removed by pressing 'delete'. The undo function should return the block of text, but is it also returned as the selected text?

We are interested in addressing these questions only as far as they are affected by multi-user undo. The first question is important, since it differentiates the two options for undo that the designers are considering—*local* attention to individual user activity and *global* attention to document activity. Questions 2 and 3 raise issues of granularity which are no more special in multi-user undo than for single-user undo. In fact, Wright's study of Word [27] which we have already discussed showed how difficult in practice it can be for users to know how their stream of activity is divided into units and which unit undo affects. In the multi-user case, the situation is made even more difficult since all one user can know about any other user's activity must be determined by the perceivable effect of their actions—on the display screen or through some audio or visual communication channel—which may not provide enough information. The last question is of particular concern in mapping out the options for multi-user undo. It is the purpose of a formal analysis to clarify the option space.

Scope

As we have indicated, many of the probelms of multi-party undo, such as the granularity issues, are no different from those for single user undo. These problems are important, but we will confine ourselves to the *new* issues posed by multiple users. For the purpose of exposition, we will usually consider examples where the user actions are at the level of words and selections.

We will not consider the problems of true distribution. We will assume that the group editor is effectively centralised and that the effects of one user's actions are instantly available to all other users. This means that we consider problems of interleaving, but not true concurrency. Real group editors are likely to be distributed and replicated; however, the problems posed by distributed undo are similar to those for distributed editing. There are known algorithms for giving the impressision of centralised editing (e.g., Ellis and Gibbs [10] described later). Though we do not consider the problems of concurrency control as special for multi-user undo, it is interesting to note that the techniques which help make group undo possible are almost exactly the same as those for distributed editing.

We will be focusing on fine-grained synchronous group text editors, both those with a single shared insertion point and with multiple insertion points. Much of the discussion will be relevent to other fine-grain shared tools, such as spreadsheets or drawing surfaces – the important thing being that user's actions may be interleaved. However, text is easiest to demonstrate with examples.

Some of the discussion will also be relevent to coarser-grained cooperation, but this will not be the thrust of the paper. Undo in such contexts is probably best seen as a problem connected with versioning or backup. However, some of the mechanisms proposed towards the end of the paper could seamlessly operate between fine-grained and coarse-grained cooperation.

Using formal methods

One of the problems we will face is simply defining the options for the meaning of undo for a group, and for this a formal model is invaluable. It also helps us to see that informal descriptions of what we want in such an undo may be meaningless in some situations. Consequently, *any* system built for group undo which behaved properly in typical situations would sometimes misbehave terribly. The formal model thus helps us avoid solutions which are impossible to implement solutions. It might be impossible to implement more than one undo system (undo systems are both difficult to program and to user test), but we can discuss the properties of alternatives given a suitable description.

The formal models that we use are designed to model the system properties *from the user's perspective.* They obviously do not employ the language of the user, but nevertheless still represent a user-oriented design methodology. In this respect, they have prompted solutions and clarified problems. Also, the models show that some problems of group undo are not just difficult but fundementally intractable. It was this insight which liberated us from seeking a functional solution to undo and prompted a broader perspective to encompass support for the user's intention to to recover from error.

3 Existing models of undo

Single user undo

Plenty of work has been done to describe various models for undo. For general interactive systems, formal models of several undo options are available due to the work of Archer, Conway and Schneider (the ACS model [3]), Vitter (the US&R model [26]), Yang (various history-based and circular models [29, 28]) and Dix (in the context of the PIE model and multi-windowing systems[5, 6]). Leeman described the semantics for basic undo facilities in a programming language [18]. In this work, there is at least the implicit assumption that the interactive system involves only a single user. Furthermore, there is also the assumption that the user is the initiator of all changes to the state, so that the actions that are undone are actions that the user at one time did do. Though both of these assumptions are violated in the shared editor, we will still begin with the single user undo models and try to extend them. Our justification for this is partly because the single user undo models are well understood and relatively simple and partly because the complications involved in the shared case may also inform the single user case. Thimbleby [25] provides a very good synopsis of the ACS and US&R models, and it is his description of these models which motivates the approach of this section.

The undo facility can vary in functionality from the familiar "undo my last command" to an extremely sophisticated system of commands for manipulating the entirety of the previous interaction history. To understand undo in the more general sense, it is best to describe the user's interaction in terms of commands that perform two separate functions. One class of commands is domain related and directly associated to the task that the user is performing. In our example, the domain commands perform text editing tasks. The other class of commands perform the support for undo. In Thimbleby's terms, the domain commands provide a *script* which determines the resultant text document and the undo commands are a means of editing that script. The separation into these two command classes is relevant from a user's perspective. Furthermore, Dix has shown [6] that attempts to regard the 'undo' commands as similar in kind to the 'ordinary' commands, though theoretically appealing, soon lead to inconsistencies for all but the simplest undo schemes.

The current state results from the commands in the *active script*. Usually, when a user issues a command it would be appended to the end of the active script, but the undo related commands would perform some more complicated manipulation. For instance, the simplest undo would chop off the last command from the active script. Once commands are removed from the active script, the user may want to reissue them at a later stage, and so the model also keeps track of removed commands as *pending scripts* in addition to the active script.

We provide a more concrete example of undo as script editing. Assume a single user is editing a text file. We will concentrate on user-issued commands at the level of abstraction of the word. The user inserts words and can select words and perform operations on the selected text. Table 1 shows a typical short session of interaction. The first column represents the user's interaction history. The second column details the active script after each command issued by the user. The third column gives the state which results from interpreting the active script. The fourth column represents the one (and only in this simple example) pending script. When the user issues an `undo` command, the last command is removed from the active script and placed at the beginning of the pending script. The new state is then determined from the new active script. The purpose of the pending script is demonstrated by the ensuing `redo` command, which removes the most recently undone command from the pending script and appends it to the end of the active script.

User history	Active script	State	Pending script
`ins(hello)`	⟨ *ins(hello)* ⟩	hello|	⟨ ⟩
`ins(world)`	⟨ *ins(hello), ins(world)* ⟩	hello world|	⟨ ⟩
`sel(hello)`	⟨ *ins(hello), ins(world), sel(hello)* ⟩	$\boxed{\text{hello}}$ world	⟨ ⟩
`del`	⟨ *ins(hello), ins(world), sel(hello), del* ⟩	|world	⟨ ⟩
`undo`	⟨ *ins(hello), ins(world), sel(hello)* ⟩	$\boxed{\text{hello}}$ world	⟨ *del* ⟩
`redo`	⟨ *ins(hello), ins(world), sel(hello), del* ⟩	|world	⟨ ⟩

Table 1: A simple example of the script model

The undo and redo commands can be generalized so that the user can undo an arbitrary number of previously issued commands or redo any number of previously undone commands. In Vitter's US&R model, the user can `skip` any command that is at the head of the pending script. A skipped command is then placed at the head of another pending script. When more than one pending script is available, the user must choose which pending script a subsequent `redo` command affects. The additional complexity of `redo` and `skip` are not considered further in this paper as they do not pose any particular new problems to group work over and above those of plain undo.

Extension to multiple users

We extend the script model depicted by Table 1 to cover the case of more than one user. We will assume there is only one insertion point, or cursor, and the overall user history is an interleaving of commands issued by each user. As motivated by the scenario description, there is a question as to whether the script editing commands should apply to the entire multi-user history of interleaved commands (the global undo) or whether they should be apply to the single user's history (the local undo). Table 2 presents an example of the global undo for two users, whereas Table 3 presents an example of the local undo for two users.

In the first case, with a global undo, the effect on the state is identical to the single user case. This is because the system effectively does not distinguish between the users. For the local undo, however, the identity of the user becomes important. The `undo` command now refers to the last command of user1, `ins(world)`, rather than the last command overall, `del`.

One point which arises immediately from this simple exercise is how very difficult it is to present any moderately complex user history in the tabular form, and so the examples

User history		Active script	State
1	2		
`ins(hello)`		⟨ *ins(hello)* ⟩	hello|
`ins(world)`		⟨ *ins(hello), ins(world)* ⟩	hello world|
	`sel(hello)`	⟨ *ins(hello), ins(world), sel(hello)* ⟩	$\boxed{\text{hello}}$ world
	`del`	⟨ *ins(hello), ins(world), sel(hello), del* ⟩	|world
`undo`		⟨ *ins(hello), ins(world), sel(hello)* ⟩	$\boxed{\text{hello}}$ world

Table 2: Multi-user global undo

User history		Active script	State
user1	user2		
`ins(hello)`		⟨ *ins(hello)* ⟩	hello|
`ins(world)`		⟨ *ins(hello), ins(world)* ⟩	hello world|
	`sel(hello)`	⟨ *ins(hello), ins(world), sel(hello)* ⟩	$\boxed{\text{hello}}$ world
	`del`	⟨ *ins(hello), ins(world), sel(hello), del* ⟩	|world
`undo`		⟨ *ins(hello), sel(hello), del* ⟩	|

Table 3: Multi-user local undo

we have provided are not very general. We will rectify this by using a more abstract and mathematical version of the script model in Section 4.

However, even this representation has clarified somewhat the different design options for undo – global vs. local, and so before we move on to the formal model, we shall discuss some of the conflicting claims of the two approaches.

Global vs. local undo

The global undo appears easier to implement from the system perspective. It does not matter to the system how many users are issuing commands because the script editing commands behave exactly as they do in the single user case. In the local undo case, the semantics of the script editing commands are complicated because the system must keep track of which user issues which commands. Furthermore, it is not difficult to find pathological examples for local undo. What happens when a user wants to undo a text deletion only after another user has removed the whole paragraph where the text originally resided? Is local undo not only hard to implement, but fundamentally meaningless? We will leave it to the formal model in the next section to show exectly where local undo is and is not a meaningful concept.

We saw in Section 2 that users must know what stream of activity is relevant to the undo operation. In the shared case there are two options. There is one stream of activity corresponding to each user's actions—the local option—and there is the composite stream of all the users' actions ordered, say, by timestamp—the global option. Are these different streams apparent to the user? It is reasonable to assume that each user is aware of their own actions, but are they aware of those of other users?

In the case of a single insertion point, each user's attention is focussed on the cursor which is the locus of all actions. Thus, a user is likely to be aware of the composite stream. This suggests that global undo, which is based upon the composite stream, will be acceptable in single cursor applications. Often such applications are obtained using standard applications and shared screen or window systems such as Timbuktu [11] or Shared-X [14]. As the application is unaware that there is more than one source of commands, any undo supported will, by necessity, be global. Fortunately, this concurs with the model that the users are given, namely of a virtual shared computer where the individual users pass the keyboard and mouse between one another.

There are some potential problems due to 'race conditions' with a single insertion point.

> ... if two individuals' actions occur close in succession, the author of the action to be undone may not enter the undo command until another group member's action had taken place—a situation likely to create enormous confusions. *Olson et al.* [22]

Arguably, this is a somewhat contrived situation, especially if one assumes that the users are communicating by some additional channels—surely they would tend to say "Oops!" as they did the undo. However, it does suggest that the undo in shared insertion point editor should be not operate across changes in floor-holder. We see that even though global undo appears to be the most viable solution for the single insertion point case, there are still situations in which it will cause unavoidable problems.

In the case of multiple insertion points, the users are far less likely to be aware of one another's actions. It is only when a clash or conflict arises that one is forced to notice the other's actions. In general, there is no reason to assume that users will even notice whether the last command was their own or not. So, we must conclude that the stream of activity implied by global undo is not salient to such users and thus that local undo is the only meaningful mechanism to supply.

The position so far

To summarise, there are two options for shared undo:

local undo operating on the user's own actions

global undo operating on the combined actions of all users

From the system's perspective global undo is easier, and it is not even clear whether local undo is even meaningful. However, the users will clearly normally expect local undo except (possibly) in the case of a shared insertion point editor. The design choice between single insertion point and multiple insertion point is, therefore, a critical one as far as undo is concerned. The situations in which local undo is preferred by the users will be our main focus in the remainder of this paper. We turn to a formal model to help resolve the conflict between the system's preference for global undo and the users' preference for local undo.

4 A formal model for multi-user undo

In this section, we want to formalize the script model in terms of the handle space model, first given by Dix [7, 5, 6]. This model was originally proposed to model multi-window interfaces. However, it used the metaphor of a multi-user system. Windows were associated with different tasks for the user, and it was assumed that the single user, in a rather schizophrenic fashion, took on different personnae whilst attending to different windows. Windows tend to be associated with different tasks, so it was assumed that while operating in one window, the user would forget actions in all other windows. It is not surprising then that given its multi-user origins, this multi-window model encountered exactly the same issues of global vs. local undo as our multi-user analysis.

There are some differences, however, between the emphasis of this paper and that focused at multi-window systems. One difference is that windows are created and destroyed relatively frequently, whereas, although users may log on or off the system, they are rarely completely destroyed! Thus the multi-user case is not structurally dynamic—a considerable simplification. On the other hand, the sort of race condition described earlier, in which two users enter commands simultaneously, does not occur in the single-user multi-window situation.

Another difference is the level of cooperation assumed. When modelling multiple windows it was assumed that the user desired *no* sharing between windows for different tasks. The metaphor was therefore closer to the traditional timesharing computer system. With the group editor we want the users to be able to share a common focus and yet not interfere too much with one another's actions. This is similar to a single user with several windows representing different parts of the same task—the earlier analyses assumed one window per task, thus sidestepping the cause of interference. Despite the differences in emphasis, however, the handle spaces model is very useful in understanding the options for multi-user undo.

The model

The set of possible user commands is denoted by the set C. In the earlier examples, this would include `ins(`*`any word`*`)`, `sel(`*`any word`*`)` and `del`. In the multi-window work, each command is tagged with a *handle* which identifies the window to which the command is issued by the user. In the multi-user case, the handle indicates which user issued the command. The set of user handles is denoted by U. In the example with two users we would have $U = \{ \ user1, user2 \ \}$. A history of user actions, denoted by H, is a sequence of pairs from $U \times C$.

$$H = (U \times C)^*$$

In the example, a valid history would have been

$$h = \langle (user1, ins(hello)), (user1, ins(world)), (user2, sel(hello)) \rangle.$$

The set of possible system states is denoted by S. Beginning with any initial state of the system, s_0, the current state is obtained using a state transition function *doit*.

$$doit : (S \times H) \to S$$

In the example, with the history h given above and initial state s_0 being the empty document we would have

$$doit(s_0, h) = \boxed{\text{hello}}_2 \ world|_1.$$

The subscripts attached to the cursor and selection box above indicates that the system state knows the ownership.

Each user sees only a part of the whole state of the system. This is denoted by a display function for each user $display_u$ where u is the specific user's tag. The display functions map from the system state S to some different set, say D.

$$\forall\, u \in U \bullet display_u : S \to D$$

In the earlier examples, we did not distinguish the current state from the users' displays. In a large document these would be some small part of the document which would fit on the screen, and may be (depending on the form of groupware) different. For example, given the history h, we would not expect that the selection of text by the second user would be visible to the first.

$$display_{user1}(doit(s_0, h)) = hello \ \ world|$$
$$display_{user2}(doit(s_0, h)) = \boxed{\text{hello}} \ world$$

As well as identifying the individual views, the model distinguishes that part of the state which corresponds to the permanent result of the interaction. In the example, this *result* function would extract the contents of the document but would discard the selection and insertion point information. Since we are dealing with a shared document, the result is not local to the user, and so we do not represent individual result mappings as we did for the display. The result is, therefore, a mapping from the system state S to the result set R.

$$result : S \to R$$
$$result(doit(s_0, h)) = hello \ \ world$$

Independence properties and commutativity

The main purpose of the handle space model was to investigate interference or unintentional sharing between windows. This lead to several definitions of independence based on non-interference of results of the completed interaction or of the displayed effects. These independence properties are intimately linked to the analysis of multi-window undo and thus shed light on global/local undo for multiple users. As we have discussed, however, we do not want total independence in a cooperative editing environment—the whole purpose of interaction is collaboration and sharing.

One form of result independence says that, no matter what order commands are submitted by different users, the ultimate effect is the same. The formal statement says that for any state s, two commands—a issued by $user1$ and b issued by $user2$—are *result commutative* if

$$result(doit(s, h_{ab})) = result(doit(s, h_{ba})),$$

where

$$h_{ab} = \langle (user1, a), (user2, b) \rangle$$
$$h_{ba} = \langle (user2, b), (user1, a) \rangle.$$

Even if two commands are result commutative, the individual displays may betray a difference in the ordering. We say that command a issued by $user1$ is *display independent* of $user2$ if the effect of a is not perceivable by $user2$, that is,

$$display_{user2}(s, \langle (user1, a) \rangle) = display_{user2}(s).$$

Similarly, command b issued by $user2$ can be display independent of $user1$. Two commands issued by separate users are *display commutative* if their relative order does not affect the final display, in which case we write

$$display_{user1}(doit(s, h_{ab}) = display_{user1}(doit(s, h_{ba}))$$
$$display_{user2}(doit(s, h_{ab}) = display_{user2}(doit(s, h_{ba})).$$

We can now apply these formal principles to the multi-party undo situation in order to seek a compromise between the favoured system and user perspectives on the problem.

Rewriting history

Consider again the case in which $user1$ submits command a and then $user2$ submits command b, denoted by h_{ab}. Now $user1$ changes his mind and decides to undo the effects of command a. The two options for the effect of undo are straightforward.

Local undo The undo acts on his own command, resulting in an active script of $\langle (user2, b) \rangle$.

Global undo The undo acts on his partner's command, reversing her b action, leaving an active script of $\langle (user1, a) \rangle$.

Let us further assume that a and b are result commutative.Then, although a was submitted first, the system could rearrange the order to yield a history where b was instead first,i.e., the history would be h_{ba}. In this revised history, if $user1$ does an undo, local and global undo concur. Having seen that $user1$ was doing an undo, the system can effectively *rewrite history* and pretend that the commands came in an order which is convenient.

In fact, the implementation mechanisms can be even easier as the rewriting can be notional. The system can calculate the actions it would have performed on the state arising from h_{ba} in order to undo $user1$'s action. As the results of h_{ab} and h_{ba} are identical, these actions can be performed on the actual system state arising from h_{ab}.

Of course, in most systems there will be pairs of commands which are not commutative. In this case an individual (local) undo does not seem appropriate and the undo must be to some extent cooperative. An undo system can distinguish commutative and non-commutative commands, performing local undo for the former and warning the user (possibly suggesting alternative actions) for the latter. With suitable representations of commands (see Section 5), it is realtively easy to check for commutativity, and to ensure that *all* commands of certain classes (e.g., character oriented commands) commute with one another.

To summarise the position, situations in which result commutativity holds allows for local undo to be implemented as global undo. This can be used to implement undo where it is meaningful and also detect when it is not.

What the user sees

Result commutativity does not take into account the intermediate displays and their effect on the separate users. Command b from $user2$ may have been directly in response to seeing the effects of command a by $user1$. There could then be a problem if $user2$ did not then notice that a was later undone. There may be no dependence on the order of a and b within the system, but there could be between the users.

Imagine now that we have a similar situation but that a and b are instead display commutative. Thus, the result of following a by b could be different from b followed by a, but the users *cannot see the difference.* In this case, the system, when faced with $user1$'s undo, can again notionally reorder a and b before making the undo act on a. This may require the system to reverse the actions of both b and a and then redo b, or it may have some more efficient mechanism. The reversal in the order of a and b has no visible effect, but it may make a big difference to the later behaviour of the system. However, *assuming there is no direct communication* between the issuing of the two commands a and b, the users cannot know which was originally executed first because one could not see the displayed effect of the other's command. The users would not have planned their actions based on the ordering of the commands. We can express this behaviour as a principle for design.

> **The principle of reordering**: *Any ordering of the history with the same visible effect for each user is acceptable.*

Implicit in this principle is the caveat of no intermediate direct communication.

We now have a revised display-based statement which allows the system to implement local undo in terms of global undo. However, there are still some problems. First of all, implementation considerations make the use of result commutativity easier, so we may want to only allow automatic undo when both result and display independence hold. This is not quite as stringent as it sounds as an examination of real examples shows that for most modern interfaces, when commands are display commutative they are also result commutative.

Cooperation?

More seriously, display independence may be far too strong for cooperative work. If the two users are editing the same paragraph, but one is acting at the top and the other at the bottom, it seems reasonable to allow undo. The users' commands in this case would be result independent (with suitable representation), but not display independent (they can see each other's edits). It is certainly possible that the second user bases her actions on the visible effects of the first user's command a so that the undoing of a may interfere with her work. However, it is equally possible that $user1$ may simply have *done* something else which was equally disruptive for $user2$.

Measures of independence are interference avoiding, *not* collaboration supporting. Collaboration implies interference—it depends on it. Can undo cause problems for other users? Of course it can, but *do*ing something can also cause problems. That is the nature of cooperation. This leads us to another principle.

> **The principle of action/reaction**: *If users can accept interference connected with do, then they can accept a similar level of interference with undo.*

In other words,

> *What's good for the do is good for the undo.*

We will return in Section 6 to issues arising from the interference of displays and what this teaches us about the meaning of undo. For the time being, we will focus on design mechanisms for supporting the user's local undo as a system global undo through commutativity.

5 Mechanisms for undo

We have seen that commutativity of commands allows the system to perform local undo as a form of global undo. This gives us both an implementation strategy and a way of understanding local undo. Furthermore, in cases where commands are not commutative, there is usually no clear meaning for local undo. We will consider various mechanisms which ensure commutativity, or at least increase its likelihood. These mechanisms fall into two broad categories. Coarse-grained methods largely finesse the problems of group undo by restricting collaboration. They follow, on the whole, the principle of reordering, satisfying either full display commutativity, or something close. Fine-grained methods follow the weaker principle of action/reaction. They allow collaboration at the level of character insertion.

Coarse-grained collaboration—exclusion

The following three mechanisms—locking, roles and copying—all operate by excluding all but one user from updating an object or part of an object. They ensure comutativity, but do so by reducing collaboration.

Locking

If we can ensure by locking that each application object can only be updated by one user, then the updates they perform will commute with other updates. This is the traditional database approach to ensuring commuting updates. It is also the basis of several group editors [13, 4, 17, 23, 20], where various locking schemes are used.

The most prevalent form of lock is the *explicit lock*. A user will obtain a lock on a section of a document, or a whole file, perform updates and then release the lock. During the period the lock is held, other users cannot update the same section, but they may be able to look at it. If large sections are locked for long periods of time, however, collaboration becomes difficult. Systems which want to encourage low granularity cooperation, therefore, try to reduce the granularity of locking.

Implicit locks tend to operate at a much finer granularity. When a user starts to edit some portion of a document (paragraph, sentance or even character), the system implcitly obtains a lock. When the user moves elsewhere or after a timeout—a short period with no typing—the lock is automatically released. Thus the users are unaware that there is any lock in operation unless they attempt to edit the same area simultaneously.

Although locks ensure commutativity while the lock is imposed, there is no such guarantee when the lock is released. The period of the lock puts a bound on the ability to easily undo. There is a general tendency for long-term locks to be associated with large scale changes. Most mechanisms for fine scale locking release the locks almost immediately. Locking mechanisms, therefore, either limit the level of cooperation, or have locks of such fine granularity or short duration as not to contribute to the commutativity solution of the undo problem.

Roles, ownership and social protocols

Several group authoring systems (e.g., Quilt [19]) assign *roles* to users (author, co-author, commenter) with respect to each object in the system. Depending on their roles, users may be able to read, write or add annotations to the objects. This is similar to the traditional idea of file ownership, where the file's owner has greater rights than others. The restricted access is not sufficient on its own to prevent contention, as often several users still have write access to an object. However, by making the rights of different users explicit in the system and to the participants, the likelihood of clashes which prevent undo (amongst other, possibly more serious, problems) is reduced.

Users cooperating over the use of resources often develop social protocols to prevent clashes whilst doing things to the shared objects. One of the most common dynamic social protocols is a simple baton passing protocol. One user creates a first draft of a document. This is then passed on to the second user who does some changes, who then passes it on to the next user, and so on. These baton passing protocols have been built into some groupware systems. Their importance is that they ensure a single thread of updates and thus make the meaning of undo clear. Within a single user's 'go' at the document, any action may as easily be undone as in a single user application. The user may even go on to undo the updates of the previous user, although this may be considered impolite.

Copying

Most traditional single-user applications do not act on the system's data store directly, but make copies of the data object (the document, spreadsheet, or program, for example), manipulate the copy, and then save the updated object. The copy is private to the application and thus there is no possibility of concurrent access. All commands internal to the application commute with all similar commands from other applications. In such a program, it is possible to undo the internal actions, but not the external ones. We may regard such a system as having two sorts of operations: **U** commands, such as editing the text, find-replace or recalculating a spreadsheet, which are undoable; and **non-U** commands, such as saving a file, which cannot be undone.

As with locking and role assignment, one has the problem that either the granularity is large, preventing cooperation at a fine scale, or the granularity is small, but the system cannot undo beyond these small boundaries. Copying without locking has an added problem that users may edit copies of the same object concurrently. This results in multiple versions which must be merged. Some form of modified version control system is required to deal with this resynchronisation problem [8], and we will see in Section 6 that such version control systems can also help with undo.

Fine-grained collaboration—-choice of representation

The mechanisms examined so far increase commutativity by preventing different users from updating the same object. On the whole, the objects dealt with are rather large scale (e.g. whole documents or sections) and they usually represent complete structural units. For example, in hypertext environments, such as Quilt [19] or ACE [9], one would expect locking or roles to be defined at the level of individual hypertext frames.

The scenario which drives this paper assumes a finer grain of cooperation within a document, and even within the same part of a document. Even where techniques such as insertion point locking preserve independence for this level of granularity, they do so for only a short time. They do not help that much with the issue of undo, particularly if we required repeated undo. Imagine two users, one of whom is editing at the beginning of a paragraph while the other is editing at the end of the same paragraph. It is clear *to the users* that the edits are independent and it should be possible to undo either set without disrupting the other. Depending on the representation of the commands, this obvious interpretation of undo may be far from clear to the system.

Ellis and Gibbs' algorithm for Grove

Ellis and Gibbs [10] use an algorithm in their Grove group editor which allows just this sort of independence between user actions. Their purpose was to allow fast feedback at individual workstations and maintain a distributed architecture. For our purposes, the algorithm demonstrates well the requirements for low granularity undo. Imagine the following state which $user1$ and $user2$ wish to cooperatively correct.

$$s_0 = hekllo{\sqcup}wrld$$

First $user1$ removes the superfluous '`k`' and then $user2$ adds an '`o`' to '`wrld`'. Ignoring each other's actions the commands could be coded as:

$$a = (user1, del(3))$$
$$b = (user2, ins(8, o))$$

Either of these commands work in the context of the original line, and achieve their desired effects. In combination, the ordering matters and only one ordering has the desired effect.

$$doit(s_0, \langle ab \rangle) = hello\textvisiblespace wrold$$
$$doit(s_0, \langle ba \rangle) = hello\textvisiblespace world$$

The problem is that the representation of the commands is *static*—it does not take into account the structural change to the line that other commands may have. Ellis and Gibbs overcome this by modifying a command based on other commands which have been executed after it was intended to be executed but whose actual execution took place first. So, if a and b were as before, then if b is executed after a it is modified to b_a, defined to be

$$b_a = (ins(7, o), user2).$$

Now a followed by b_a has the desired effect:

$$doit(s_0, \langle ab_a \rangle) = hello\textvisiblespace world$$

The command a is similarly modified to a_b when it follows b, but this time there is no change necessary and $a = a_b$.

Ellis and Gibbs suggest having a rule for every pair of operators and give detailed rules for character level insertion and deletion. For n operators, this technique yields n^2 rules in all which must be calculated.

Dynamic pointers

A similar and more general approach is that of dynamic pointers [6]. The commands above could have been described in terms of pointers. The pointers in this case would represent the position of the individual insertion points and they would be dynamically linked to their semantic position within the text. For example, $user2$'s insertion point, p_2, as a dynamic pointer would point to the gap after the letter '`w`' rather than to the eighth position in state s_0.

$$s_0 = \mathtt{h\ e\ k} \underset{\substack{\uparrow \\ p_1}}{} \mathtt{l\ l\ o}\textvisiblespace\mathtt{w} \underset{\substack{\uparrow \\ p_2}}{} \mathtt{r\ l\ d}$$

The commands used earlier would be accordingly modified.

$$a = (user1, del(p_1))$$
$$b = (user2, ins(p_2, o))$$

Each command not only changes the text but updates the positions of the pointers. So if we do a first, we delete the character before p_1 and get

$$s_1 = doit(s_0, \langle a \rangle) = \mathtt{h\ e} \underset{\substack{\uparrow \\ p_1}}{} \mathtt{l\ l\ o}\textvisiblespace\mathtt{w} \underset{\substack{\uparrow \\ p_2}}{} \mathtt{r\ l\ d}.$$

If this is followed by b, we insert an ‘o’ at p_2 and obtain

$$s_2 = doit(s_1, a) = \mathtt{h}\underset{\substack{\uparrow \\ p_1}}{\mathtt{e}}\mathtt{llo\textvisiblespace w}\underset{\substack{\uparrow \\ p_2}}{\mathtt{o}}\mathtt{rld},$$

which is exactly what we wanted. In fact, the pointers will often be stored as numerical indices, as in Ellis and Gibbs’ scheme, and thus the difference for simple commands is negligible. However, the dynamic pointers, being a general mechanism, make it easier to add support for more complicated commands. Ellis and Gibbs demand that every pair of commands has a rule determining how one is affected by the other. With dynamic pointers we only need to know how commands affect the pointers, and then describe the users’ commands in terms of pointers. That is, the effort is linear rather than quadratic in the number of commands effected.

By way of example, dynamic pointers can handle quite complex sequences of operations such as those depicted in Table 4. Even though $user2$’s actions move the point at which $user1$’s deletion took place, the dynamic pointers still track this and the two sequences of actions commute. When $user1$ finally issues the undo command, the effect is as one would expect (although, grammatically speaking, the text should read “`undone, alas am I`”).

User history		State
1	2	
		Alas I am undone
`sel(I)`		Alas $\boxed{\text{I}}$ am undone
`del`		Alas am undone
	`sel(Alas am)`	$\boxed{\text{Alas am}}$ undone
	`move`	undone $\boxed{\text{Alas am}}$
`undo`		undone $\boxed{\text{Alas } \boxed{\text{I}} \text{ am}}$

Table 4: Complex undo with dynamic pointers

There are still pathological cases; for instance, overlapping (rather than nested) block operations for which dynamic pointers do not give easy answers. But these are precisely the cases in which it is not obvious as a user what the meaning of undo should be. The advantage of dynamic pointers even in these pathological cases is that they highlight precisely when such confusion occurs for the system as well.

Even in these pathological cases, dynamic pointers can suggest interpretations which can be presented to the user for confirmation. For example, if the user inserted a paragraph and then decided to undo the insert when a second user had begun to edit the paragraph, the system can prompt the first user “`Undo: Gregory is editing this paragraph, delete (Y/N)?`”. One would expect a similar sort of message if the first user tried to delete the paragraph using normal methods. That is, we are following the principle of action/reaction.

Factoring the state space

A third method of representation to increase commutativity is to factor the state space. This technique can be used in conjunction with any of the other mechanisms already discussed. We could readily formalize this technique to show how it promotes result commutativity, but we will only provide an informal description here.

Imagine one user edits a style sheet. Another user then begins to edit a paragraph to which that style applies. The first user then decides to undo the style change. If we view this imperatively, as the original style being reapplied to the paragraph, then it clearly interferes with the second user. However, if we view the style sheet declaratively, the state is factored into two components—the style sheet and the document. One user is editing the style sheet, the other the document. Thus, there is no conflict. The formatted document as presented on the users' screens is derived from both of the components of the state. The change in the display for the second user might be disconcerting and as a matter of politeness users may refrain from updating styles when others are working, but there is no fundamental conflict.

State factoring is as much a matter of the user's model as the system's. By seeing the system's state as factored, the user can predict and comprehend their actions and the effect of undo.

Summary of mechanisms for undo

Most of the mechanisms we have presented allow commutativity at a low level of granularity effectively by preventing the users from cooperating during this period. There are arguments for and against low level cooperation. However, even assuming that low level cooperation is not required, some form of facility for supporting undo at the large scale is required. At this level of granularity, quite extensive user intervention is acceptable, and version control mechanisms allowing multiple version threads, as described in [8], become the preferred option. These mechanisms have the added advantage that they can handle asynchronous as well as synchronous work. Such mixed mode working is likely with large granularity cooperation.

However, the thrust of this paper is towards synchronous low-granularity cooperation. The options here are more restricted and depend on a judicious choice of representation to facilitate undo whenever the user can reasonably expect it to be meaningful. There are still bad scenarios, not because of difficulty in implementation but because there is no obvious interpretation of the meaning of undo for the user. The duty of the system should be to detect such circumstances and consult the user, rather than make guesses.

6 A broader definition of undo

We have seen that undo can be given a useful meaning which unites the user's preference for local undo and the system's preference for global undo whenever actions commute. However, there were still some problem cases with which the system could not sensibly cope. Is an undo facility that only works some of the time worth having?

This section takes a step back and looks at the meaning of undo. Previously we have been effectively asking, "What should the undo button do?" Instead, we should be asking, "How can we support the user's own undo?" After all, it is sensible that only the user can assess the appropriate meaning of undo in some contexts.

Return to the display

Recall that in Section 4 we were considering whether commands which affected another user's display should be candidates for undo. We noted that although the first user's undo might interfere with the second user, so too might the user's other actions. This gave rise to the principle of action/reaction—users can accept a level of interference with undo similar to that which they accept for other commands. But how does this interference appear to the other user?

Take as an example, two co-authors working on the same document (Gregory and Alan). Alan decides that two paragraphs would be better in reverse order and moves the first to be after the second. Gregory notices that the paragraph which is now first does not make sense as it uses an acronym defined in the original first paragraph. He proceeds to alter the two paragraphs to be consistent with the new order. Meanwhile, Alan changes his mind and hits the undo button. The paragraphs switch back, and presumably Gregory is rather miffed. But what exactly does Gregory see? It is not necessarily obvious that undo has been used at all. Originally, Gregory saw the paragraphs change order, now he sees them change back. For all Gregory knows, Alan may have simply selected the second paragraph by hand and moved it back to the top. However, Gregory will probably *interpret* Alan's actions as an undo, whether or not he believes the action to have been carried out by use of the undo button. Alan did something, and then did something else to reverse the effect of his first action.

We can abstract from this example. What is undo? It is an *intent* on the part of the user to reverse some previous action. The user may achieve that intent by way of an undo button, or by use of normal system commands. The undo button can then be seen as a rather sophisticated, context-sensitive macro key which will perform whatever action is necessary to achieve the user's intent.

Backward and forward error recovery

We begin to have a new emphasis; rather than focus on the functionality of undo, we should focus on error recovery. The idea of an undo button is derived from closed systems, in particular single-user computer applications. A multi-user system is not closed, the actions of any one user can have repercussions on other users. Error recovery in an open system has a different set of solutions.

In dependable and open systems, a distinction is made between backward and forward error recovery. With *backward error recovery*, the system detects when an error has occurred, and restores the system to the state just prior to the error before retrying the failed operation. This is exactly analogous to the effect of the undo button. However, in an open system the erroneous command may already have had an effect on the environment (the operators at Three Mile Island would have loved to have had an undo button, but it was not possible). Similarly in a real-time system one cannot turn back the clock and try again. Even if it were possible to return the system to a previous checkpoint, time has passed and some different operation may have to be used to accommodate the now closer deadlines.

In these circumstances a different approach is necessary, *forward error recovery*. This recognises that the system has some goal which it is trying to achieve. The system has performed some action which was not optimal in achieving that goal. Instead of moving back, the system moves on towards the goal from the present, albeit sub-optimal position. For example, the navigator in an airplane cockpit who has realized that the plane has just veered off in the wrong direction, cannot proceed to instruct the pilot to back the plane up to the past turning point and then bank right instead of left! In some circumstances, backward and forward error recovery may concur—the best way to make progress may be to go back. However, in general, forward error recovery is more flexible.

The attempts to handle undo via commutativity are similar to those used in database systems. These systems strive to maintain the illusion of being closed, even in the case of multiple clients. Any error on the part of one client can then be resolved by *rolling back* that client's transaction, that is, through backward error recovery. Much of our preceding discussion can then be seen in a similar vein, to give the individual users the impression that they can simply roll-back the clock when they do something wrong. But at the same time we want them to cooperate with one another. No wonder it's difficult!

Forward error recovery gives us a different perspective, in keeping with the idea of undo as a user's intention. Users interact with a system in order to achieve certain goals within some work domain. This goal achievement is a necessarily forward-seeking activity. Since users are (for the most part) human, they are fallible creatures; mistakes do happen along the way and things do not turn out as intended. Once the user discovers an error, there are two ways that they can go about rectifying the error in an effort to attain the original goal. They can retrace their steps and reverse the effects of past actions, or they can determine a new course of action which will take them forward from their current situation toward their original goal. That is they can engage in backward or forward error recovery.

In a single-user application, either forward or backward error recovery is possible. Shneiderman's demand that a direct manipulation interface should allow easy and incremental undo [24] does not presuppose the existence of an 'undo' button. A direct manipulation interface is often characterised by goal-seeking as opposed to pre-planned activity. Even when an undo button is available, users often undo erroneous actions by simply performing the inverse action themselves. However, not only the inverse action is possible; often other forward recovery actions are just as easy.

With a multi-user interface, the openness makes backward recovery a less useful concept. At best users can attempt to mimic the effects of it. Even where we can define a suitable functionality, it is better to regard the effects of an undo button as a form of forward error recovery, that is, we regard undo as action rather than reaction.

Implications for undo support

In summary, we can phrase another principle.

The principle of intent: *Undo is the user's* intention*, not a system function.*

An undo button is then seen as an *action* which aims to satisfy that intention. However, the system will not always be able to act so as to perform exactly the intent of the user. Can we design a system which supports the user's own forward error recovery?

We have already discussed the way that direct manipulation systems often achieve this aim in a single user setting. Properties which support this include *visibility*—being able to see the effects of one's actions and thus judge them against one's goals—and *commensurate effort*—big changes require big actions. By identifying such features in effective single user interfaces, we can see how they can be applied to the multi-user case. For instance, in low granularity interaction, visibility of one's own and other people's actions makes it easier to judge appropriate corrective action after mistakes. Further, if the system does support an undo button and if its effects are readily visible, then the user can judge whether it performed the desired recovery and make further corrections if not. At a larger granularity, say blocks of text, visibility would demand that one is made aware if other users are overlapping one's selected blocks, and proportionate effort would demand that in such situations both doing and undoing one's actions should require more user care.

In an open system, it is not possible to simply restore a previous state, but undo support may take the form of records of those previous states. One example of this is the Recent card in a HyperCard stack [2]. This allows the user to easily return to recently visited cards—a form of undo in the context of browsing. In some editors, deleted text is put into a special buffer. The 'undo' button is then simply a copy from this buffer. Generalisations of this may allow a stack of such buffers (as in the '`vi`' editor under UNIX [16]), with selective 'undoing' of previous deletes. A similar feature is the Mac wastebasket. In many operating systems deleting a file is difficult or impossible to undo, involving backup tapes or disk doctor programs. However, on the Mac, a file put into the wastebasket can simply be picked out again. That is, rather than performing some undo or undelete function, the user does something (forward recovery) to restore the file. Of course, this forward recovery is only possible between periodic emptying of the wastebasket.

We can generalise from the previous examples. The proposals for complex undo facilities such as US&R [26] involve browsing the user's past commands. Surely it would be more useful to browse the past states. Imagine if Hypercard's Recent card allowed you not only to see where you had been but also to see what the cards were like when you last visited them. You could then cut and paste from (but not to!) the past into the present. Time rolls on, but we can learn from and use the past. The advantage of such systems for recalling past information is that they generalise easily to the multi-user case. The users of a multi-party `vi` (heaven forbid!) could browse their own buffers, and possibly their partners' buffers, in order to recover lost text. Multi-party version control systems, as described in [8] would form a suitable infra-structure for such a state browsing facility that would not be all that complex and would also provide larger granularity undo support.

If as a design discipline we aimed for systems with sufficient support for the user's own forward or backward recovery, the undo button, although supplied, would be redundant.

7 Recommendations for group undo

We can now summarise the recommendations for undo support in a shared editor, looking at both the undo function and undo support for the user. Not all these features would be present in a single editor, but we would at least expect undo facilities at both system and user levels.

Undo function

single insertion point Use *global* undo based on the composite stream of users' actions. For shared window systems such as SharedX, this can be the application's normal undo mechanism. Some warning may be required when users attempt to undo beyond the beginning of their turn.

shared insertion point For character level operations, the algorithms of Ellis and Gibbs, or dynamic pointers can be used to implement local undo.

For block level commands dynamic pointers will be able to succesfully manage many undo operations. When there is any likelihood of conflict this can easily be detected and the user warned. A dynamic pointer based undo mechanism can suggest a possible undo, leaving confirmation to the user.

User undo support

commensurate effort We must ensure that commands that are easy to *do* are easy to *undo*. This will include the storing of the text from the users' deletes—not just the last delete, but many if not all in the session. In a Mac-like interface these could be represented as objects in a special folder in the clipboard.

large scale undo To support undo over a long time scale (and shorter ones as well), the user should be able to browse the past states of the shared document, and, if desired, copy sections of the old state. This process can be further aided if the system provides a merge tool to help the user to compare and re-unite different versions of the same document.

8 Conclusions

We began our analysis of multi-user undo by posing a set of psychologically motivated questions about the user's interpretation of multi-user undo and by reviewing models of single-user undo. This lead to the contradiction that from a user perspective we should supply a local undo, but from a system perspective local undo may not always be meaningful and so a global undo is to be preferred. To address this conflict, we looked at a formal model which allowed us to express the options.

We saw that it is possible to generalise the meaning of the undo button to the multi-user context. When the user's actions are non-interfering, the actions commute and we find that local and global undo agree. Indeed, we can view local undo as simply a reordering of commands followed by a global undo. When different users' actions interfere there is no longer a sensible meaning to ascribe to the undo key and the recommendation would be to give some warning at that stage as more thought and active cooperation is required.

The fact that these problem situations persist lead us to re-examine the meaning of undo. We concluded that undo should be seen as an *intention* of the user, not an aspect of system functionality. We can support this intention either by backward error recovery—the traditional undo button, or by forward recovery—designing the system so that the users themselves can perform their own recovery. Various practical suggestions were made as to how this can be achieved in design, such as traces of past activity, soft delete (buffers or the wastebasket) and, in the extreme, the ability to browse past displays and objects. Of course, the precise details and balance of features would depend on the particular application.

These conclusions reinforce what has been a recurrent theme in our work, namely that systems ought to be designed to support users in what *they* want to do. In short, the object of the designer is not primarily to provide an undo button, but to support the user's own undo.

Acknowledgements

The authors would like to acknowledge the contribution of the collaborators from the ESPRIT BRA project AMODEUS [1] and from SERC project "A formal investigation of context in a generalised mail system" [15], especially Michael Harrison, Richard Young and Victoria Miles.

References

[1] AMODEUS Consortium. Assimilating models of designers, users and systems: Final report of Esprit BRA 3066. Deliverable D23, Esprit BRA project 3066 (AMODEUS), February 1992.

[2] Apple Computer Inc., Cupertino, CA. *HyperCard User's Guide*, 1987.

[3] J. Archer, Jr., R. Conway, and F. B. Schneider. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages*, 6(1):1–19, January 1984.

[4] Cognitive Science and Machine Intelligence Laboratory, The University of Michigan. *ShrEdit: A Multi-user Shared Text Editor: Users Manual*, 1989.

[5] A. J. Dix. *Formal Methods and Interactive Systems: Principles and Practice*. D.Phil. thesis, University of York, 1987.

[6] A. J. Dix. *Formal Methods for Interactive Systems*. Academic Press, 1991.

[7] A. J. Dix and M. D. Harrison. Principles and interaction models for window managers. In M. D. Harrison and A. F. Monk, editors, *People and Computers: Designing for usability*, pages 352–366. Cambridge University Press, 1986.

[8] A. J. Dix and V. C. Miles. Version control for asynchronous group work. Submitted for publication, 1992.

[9] E. Dykstra and R. Carasik. Structure and support in cooperative environments: the Amsterdam Conversation Environment. *International Journal of Man-Machine Studies*, 34:419–434, 1991.

[10] C. Ellis and S. Gibbs. Concurrency control in group systems. *SIGMOD Record*, 18(2):399–407, June 1989. 1989 ACM SIGMOD International Conference on Management of Data.

[11] Farallon Computing. *Timbuktu: the next best thing to being there*, 1987.

[12] J. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics*. Addison Wesley, 1984.

[13] I. Greif, R. Seliger, and W. Weihl. Atomic data abstractions in a distributed collaborative editing system. In *ACM Symposium of Principles of Programming Languages*, pages 160–172, 1986.

[14] P. Gust. SharedX: X in a distributed group work environment. In *Presentation at the 2nd Annual X Conference*, January 1988.

[15] M. Harrison, A. Monk, H. Thimbleby, and A. Dix. A formal investigation of context in a generalised mail system. SERC Research Project GR/F/01895, 1989.

[16] W. Joy. *Ex reference manual*. University of California, Berkeley, Cupertino, CA., 1980.

[17] M. J. Knister and A. Prakash. Distedit: a distributed toolkit for supporting multiple group editors. In *CSCW'90 Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 343–355, 1990.

[18] G. B. Leeman, Jr. A formal approach to undo operations in programming languages. *ACM Transactions on Programming Languages*, 8(1):50–87, January 1986.

[19] M. Leland, R. Fish, and R. Kraut. Collaborative document production using Quilt. In *CSCW'88*, pages 206–215, 1988.

[20] V. C. Miles, J. C. McCarthy, A. J. Dix, M. D. Harrison, and A. F. Monk. Exploring designs for a synchronous-asynchronous group editing environment. In *Proceedings of the Workshop on Collaborative Writing*. Springer-Verlag, in press.

[21] K. Myers and N. Hammond. M1.5 workshop presentation material. Technical Presentation WS3/PRES1, Esprit BRA project 3066 (AMODEUS), March 1991.

[22] J. S. Olson, G. M. Olson, L. A. Mack, and P. Wellner. Concurrent editing: the group's interface. In D. Diaper, D. Gilmore, G. Cockton, and B. Shackel, editors, *Human-Computer Interaction—INTERACT'90*, pages 835–840. Elsevier Science Publishers B.V. (North-Holland), 1990.

[23] G. L. Rein and C. A. Ellis. rIBIS: a real-time group hypertext system. *Int. J. Man-Machine Studies*, 34:349–367, 1991.

[24] B. Shneiderman. The future of interactive systems and the emergence of direct manipulation. *Behaviour and Informations Technology*, 1(3):237–256, 1982.

[25] H. Thimbleby. *User Interface Design.* Addison Wesley, 1990.

[26] J. S. Vitter. US&R: A new framework for redoing. *IEEE Software*, 1(4):39–52, October 1984.

[27] P. Wright, A. Monk, and M. Harrison. State, display and undo: a study of consistency in display based interaction. (in preparation).

[28] Y. Yang. A new conceptual model for interactive user recovery and command resuse facilities. In *Proceedings of CHI'88 ACM conference on Human Factors in Computing Systems*, pages 165–170. Addison Wesley, 1988.

[29] Y. Yang. Undo support models. *International Journal of Man-Machine Studies*, 28(5):457–481, May 1988.

[30] R. Young and J. Whittington. Using a knowledge analysis to predict conceptual errors in text-editor usage. In J. Chew and J. Whiteside, editors, *CHI'90 Conference Proceedings*, pages 91–97. Addison Wesley, 1990.