Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work,
September 10–14, Stockholm, Sweden
H. Marmolin, Y. Sundblad, and K. Schmidt (Editors)

A General Multi-User Undo/Redo Model

Rajiv Choudhary
Intel Corporation (rajiv_choudhary@ccm.jf.intel.com)

Prasun Dewan
University of North Carolina (dewan@cs.unc.edu)

A general multi-user undo/redo model must satisfy several requirements It must be compatible with an existing single-user undo/redo model, give individual users autonomy in executing undo/redo commands, support undo/redo of remote commands and the remote effects of local commands, be independent of the coupling, multicast, and concurrency control model, and allow undo/redo of arbitrary commands We have developed a multi-user undo/redo model for meeting these requirements The model constructs the command history of a particular user by including all local commands and those remote commands whose results were made visible to that user It allows a user to undo/redo corresponding commands in the command histories of all users of a program Moreover, it allows a user to undo/redo both symmetric user-interface commands and asymmetric collaboration commands We have implemented the model in a collaboration system called Suite In this paper, we motivate, describe, and illustrate these requirements and our model

Introduction

Undo/redo is an important interactive feature whose absence seriously degrades the usability of an interactive program. It provides automatic support for recovery from user errors and misunderstandings as well as a mechanism for exploring alternatives [11]. It is offered in some form or another by most popular single-user programs [2, 12, 13, 14]. But few of the multi-user programs known to us offer this feature although it is crucial in a group setting for

several reasons. First, features available to users in the single-user case must also be available in the multi-user case. Otherwise users hesitate to use and adopt new environments. Moreover, in the multi-user case, the potential cost of an individual user's mistake is multiplied many times because it can adversely affect the work of a large number of collaborative users. Furthermore, in a collaborative setting, the number of alternatives to be explored increases due to the presence of many users. Thus undo/redo is even more useful in a multi-user application.

One reason for the absence of undo/redo from most multi-user applications is the lack of general requirements and models for multi-user undo/redo. We have identified an initial set of these requirements and designed an undo/redo model that satisfies them. The model determines how command histories of the users of a multi-user program are constructed, which commands are undone/redone by an undo/redo request from a particular user, and which users can undo/redo a command. It is an extension of the linear single-user undo/redo model [2, 14]. It constructs the command history of a particular user by including all local commands and the remote commands whose results were made visible to that user. It allows a user to undo/redo corresponding commands in the command histories of all users of a program. Moreover, it allows a user to undo/redo both symmetric user-interface commands and asymmetric collaboration commands.

This model is applicable to multi-user programs offering a variety of functionality, coupling, concurrency control, and multicast schemes. In particular, it is applicable to multi-user text/graphics editors, spreadsheets, mail programs, and code inspectors; coupling schemes offering WYSIWIS (What You See Is What I See) and WYSINWIS (What You See Is Not What I See) interaction; concurrency control schemes offering floor control and concurrent interaction; and multicast schemes supporting both atomic and non-atomic multicast.

We have implemented this model as part of a system called Suite [5]. In this paper, we motivate, describe, and illustrate our undo/redo model using the concrete example of Suite. The remainder of the paper is organized as follows. Section 2 outlines the set of multi-user undo/redo requirements we have identified. Section 3 explains the single-user undo/redo model used as the basis of our work. Section 4 motivates, describes, and illustrates how we have extended this model to meet the requirements. Finally, Section 5 presents conclusions and directions for future work.

Undo/Redo Model Requirements

We have identified a number of requirements that a multi-user undo/redo model should satisfy:

Compatibility Multi-user undo/redo should behave like single-user undo/redo when only one user is interacting with the system, thereby reducing the overhead of learning a new undo/redo model, which in turn hinders usability. This is especially true for a feature such as undo/redo that is expected to be used when the user makes errors and is thus not in a relaxed frame of mind to try new things. Compatibility allows the user to incrementally learn the new features of a multi-user undo/redo model.

Independence Multi-user undo/redo model should not require attention or intervention of all users in a collaborative session. This allows a number of users to participate in a collaborative interaction without being overloaded with the requirement of synchronizing with other users their use of recovery commands. It also allows each user to concentrate on the work he is performing.

Semantic Consistency When multiple users interact with an application simultaneously, a user's actions can affect other users. The undo/redo model must ensure that the effect of undoing a command is equivalent to the state the system would be in if the undone command had never been executed, thus guaranteeing that the degree of consistency ensured by the concurrency control policy adopted by an application is not compromised by the use of undo/redo.

Collaboration The model must allow users to undo/redo commands issued by other users, thereby allowing the users to collaborate on their undo/redo. This is a direct analogy of the ability of users to collaborate by executing 'do' commands on behalf of other users. The model must also allow users to undo the remote effects of command they issued, thereby allowing them to recover from mistakes made in their collaboration. This follows from the general semantic consistency requirement given above.

Genericity The model should be generic enough to allow its use in a number of applications with a variety of coupling, concurrency control, and broadcast schemes, thereby promoting uniformity and automation.

Undo of Arbitrary Commands The model must not be restricted to commands that manipulate the user-interface state. It must also support undoing of collaboration commands.

We describe below the model we have designed to meet these requirements

Single-User Interactive Undo/Redo

Our multi-user undo/redo model is based on a minor variation of the linear single-user undo/redo model [13], which is provided in some popular tools

such as the InterViews `idraw` editor. The model maintains a history list of executed commands and provides undo/redo/skip commands. These commands are metacommands, that is, they are themselves not added to the list. Each command in the command list has a status associated with it, which can be *executed*, *undone* or *skipped*. In addition, the model defines a *current command pointer* to point to a command in the list. When a new command is executed, it is inserted in the history list after the current command pointer and the pointer is then set to the new command. The undo metacommand undoes the command pointed to by the current pointer(if it has been executed) and moves the current command pointer to the previous command. The redo metacommand executes the command after the current command pointer(if such a command exists) and sets the pointer to the command just redone. The skip command marks the command after the current command pointer as skipped if it is currently undone and moves the pointer to the skipped command. In addition, each metacommand also sets the status of the command on which it operates.

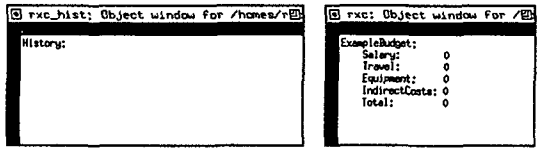

Figure 1: Initial display with the history list empty

To illustrate this model, consider a single-user interactive session with Suite that edits a simple budget. Figure 1 shows the initial display, which consists of an edit window (right window) showing the budget, and a history window (left window) displaying the history of commands executed in the edit window. The history list is initially empty and the fields of the budget are initialized to 0. [1] Figure 2 shows the contents of the two windows after the user edits the `Salary` and `Equipment` fields by moving the cursor to each field and inserting text. [2] [3] As each command is executed, it is appended to the history list. In the history list display, the last command is displayed in detail while all other commands are elided. The detailed display of a command shows a unique label for the command (motivated in Section 4.2 and 4.3), which user executed the command (a user, such as `rxc`, is indicated by the name of the Suite object, such as `/homes/rxc/objects/rxc`, that issued commands on his behalf), the

[1] In Suite, users do not edit simple lines of text. Instead, the display consists of a number of labeled fields and the user edits the text of these fields.

[2] It is not conventional to treat MoveCursor as an undoable history command in traditional editors. However in Suite, a number of other actions such as broadcast of a change can be tied to the moving of the cursor Thus, as discussed later, Suite must treat MoveCursor as a history command.

[3] Successive `InsertChar` commands are combined together in to a single `InsertChar` command

status of the command (Executed, Undone, Skipped), the name of the command (e.g. MoveCursor and InsertChar), and arguments of the command (e.g. field in which a character string was inserted, the insertion position, and the inserted string). [4] [5]

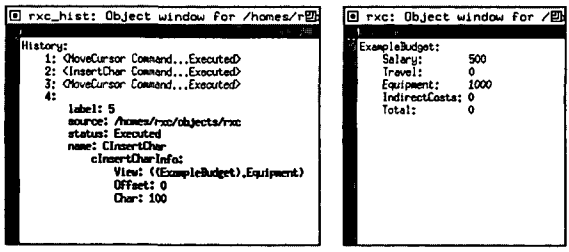

Figure 2: User edits Salary and Equipment fields

Figure 3 shows the editing window and history list after the user undoes the last two commands. As each undo is executed, the last executed command in the history list is marked undone and the effects of the command execution are removed from the display. Executing two more undo commands would return the edit window to the state represented by Figure 1 with all four commands undone, while issuing two redo commands would return the window to the state depicted in Figure 2.

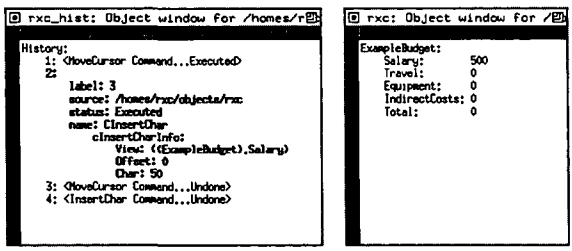

Figure 3: After two undos the Equipment field is reset

Designing Multi-User Undo/Redo

In this section, we incrementally motivate, describe and illustrate the major components of our model.

[4] We expect most users would not be interested in these details. They have been expanded here to give the reader an idea of how the system works

[5] We use a Suite dialogue manager to display the history list. The Suite dialogue manager provides facilities for eliding, and displaying different portions of the display in different fonts.

Basic Multi-User Undo/Redo

Here is a definition of a basic multi-user undo/redo model, which is a simple extension of the single-user undo/redo model above. All commands are shared by every user and commands appear in the same sequence in all command histories. An undo/redo command can be executed by any user and causes the last command in every history to be undone/redone

More formally, we provide four operators *execute, undo, redo,* and *skip* which operate upon command lists. We describe the model in terms of the effect of these functions on the command lists. Let CL_i be the command list associated with the ith user interacting with a program. Let CL be a command list such that $CL = c_1, \ldots, c_{n-1}, c_n$. Then $execute(CL, c_{new})$ sets $CL = c_1, \ldots, c_{n-1}, c_n, c_{new}$, $CL_i = CL$ and $CurrentCommand_i = c_{new}$ for each user i. $undo_i$ executed by the ith user uses the same rule as the single user model for determining the target command. Thus $undo_i(CL = c_1, \ldots, c_{n-1}, c_n)$ results in setting $CurrentCommand_i = c_{n-1}$ and $c_n.status = Undone$ for each user i. We have ignored above undone and skipped commands. The redo and skip commands can be similarly defined by considering these commands.

We illustrate the model by using the example of a multi-user editing session in Suite. Figure 4 shows command lists and edit windows of two collaborating users, users `rxc` (top two windows) and `pd` (bottom two windows), after user `rxc` has edited the `Salary` and `Equipment` fields. When user `pd` autonomously issues an undo, the resulting state is shown in Figure 5. Note that the last command is undone in both the command histories, thereby resetting the `Equipment` field in both edit windows.

The model described above meets the requirements of compatibility, independence, consistency, and collaboration. We describe below several extensions to this design to meet the remaining requirements of genericity and arbitrary commands.

Corresponding Commands

The model above assumes that all command histories have commands in the same order. This requires the availability of an atomic broadcast facility or assumes the floor control model of interaction. Certain systems do not have(or use) an atomic broadcast facility. Some of these systems also allow simultaneous execution of commands by multiple users. In such systems, when two commands are executed by different users, the system can not guarantee that they are received in the same order by all users. Thus execution of commands c_p and c_q by other users may be received by users i and j in different order resulting in $CL_i \neq CL_j$. If we still assume that an undo/redo always operates on the last command in each command history, then an undo/redo invocation may operate on different commands in different command lists, since the last command in different lists is not guaranteed to be the same.

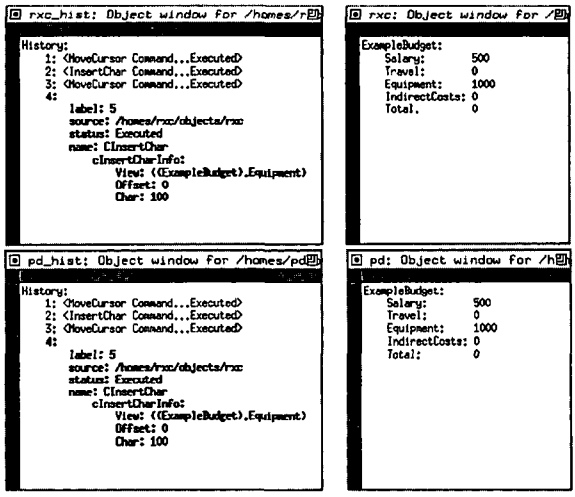

Figure 4: Local commands appear in remote histories

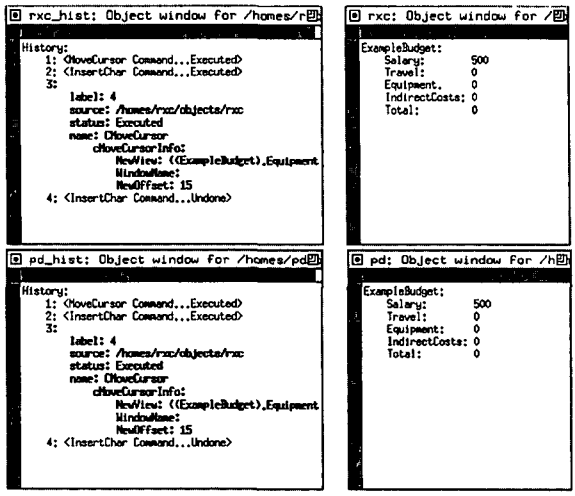

Figure 5: An undo undoes local and remote effects

More precisely, when the *ith* user issues an undo/redo command, it will result in the $CurrentCommand_j$ to be undone for each user j. But note that $CurrentCommand_i \equiv CurrentCommand_j$ can not be guaranteed and thus different commands gets undone for different users.

Our solution to this problem is to provide a way to identify all copies of a command in different command histories with a unique labels. Then semantics of undo/redo no longer need to depend upon all command histories having the same last command. When an undo/redo is executed, the last command in the local command history is undone and a request is sent to all other users to undo/redo the *corresponding* command in all the other command histories. The correspondence between commands in different command histories is established by using the unique label associated with each command. Thus $undo_i$ invoked by the ith user undoes the current command in the local command list CL_i. However the target command for the jth user is determined by $TargetCommand_j$ = Search c_k in CL_j such that $c_k.id = CurrentCommand_i.id$. We discuss and illustrate below the semantics of undoing a non-last (that is, non-current) command in a command history.

Undo by Reference

Now consider a system that also allows execution of undo/redo commands concurrently with other command execution. In such a system, when a user issues an undo to request the undoing of the last command in his command history, between the time the decision to invoke undo is reached and the undo is invoked, a new command may be executed by another user and become the last command in the command history. According to our model so far, this is the command that is undone when the undo is invoked. Thus the approach of always undoing/redoing the last command is not entirely satisfactory in such a system. For illustration, consider the multi-user editing session depicted in Figure 6. Suppose user `pd` decides to undo an accidental modification of the `Total` field. Suppose also that before he can execute the undo command, user `rxc` executes two new commands to select and elide the `IndirectCosts` fields, as shown in Figure 7. Now if user `pd` executes the undo, the `Elide` command would get undone instead of the desired command `InsertChar`.

We rectify this problem by modifying our undo/redo model to allow a user to explicitly mark the command to be undone. At most one command can be marked by a user, though different users can mark different commands. When an undo/redo command is invoked, it checks if a command in the command list has been marked. If a command is marked, that command is undone/redone. Otherwise, as before, the last command is undone/redone. Thus even if the referenced command is not the last command, this is the command that is undone/redone. To maintain the correctness of the interface state with respect to its command history, when a non-last undo/redo is requested, first all intervening commands in the command list are undone, then the referenced

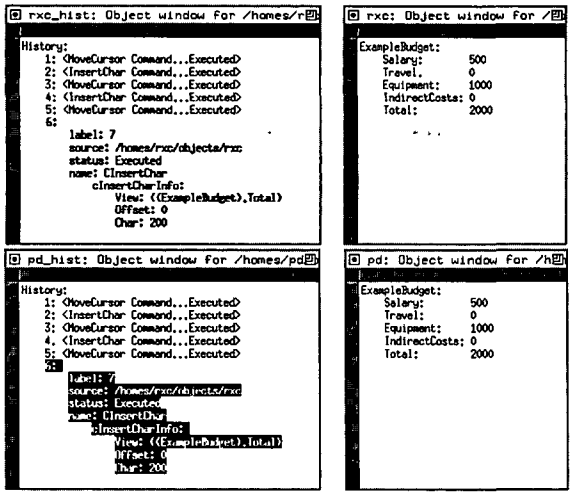

Figure 6: User pd decides to undo change to Total

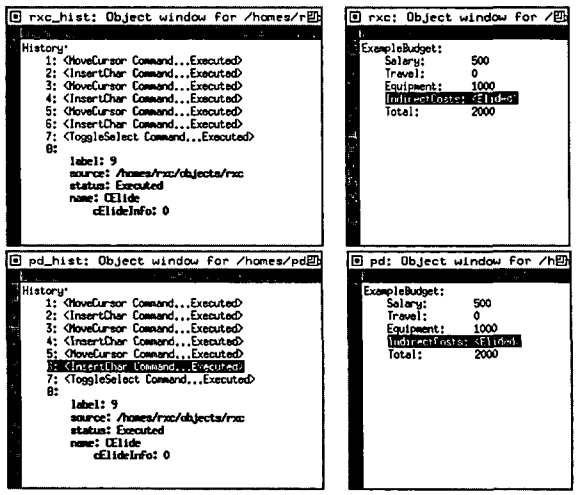

Figure 7: User rxc selects and elides IndirectCosts

command is undone/redone and skipped, and finally all other commands are restored to their previous status. In general, as discussed in detail in [10], before undoing/redoing a non-last history command, subsequent commands in the history that depend on it must be considered. Our system, currently, does not attempt to detect conflicts.

We illustrate our modified undo/redo model with a continuation of the previous example. In Figure 6, user pd indicates the command to be undone by selecting it. Then even if new commands are appended to the command list(Figure 7), it is the desired command that is undone by the undo command (Figure 8). As shown in the history windows of Figure 8, the target `InsertChar` is undone and skipped, and then the succeeding `Select` and `Elide` commands are redone.

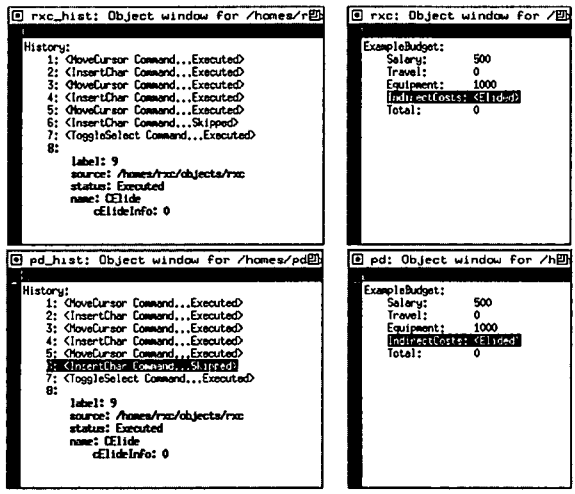

Figure 8: The referenced command is undone

Undo by reference increases the overhead of using undo/redo since it requires the user to perform the extra step of selecting the desired command. However, it is necessary only in highly concurrent collaborations. If such collaborations are rare, then the default approach of undoing the last command would be sufficient in most collaborations. We did not choose the alternative approach of requiring a user to lock the system before an undo/redo since that would increase the overhead of executing all undos/redos.

Selective Command Sharing

Now consider a system that provides non-WYSIWIS coupling [5, 9], that is, does not require the results of all commands to be shared immediately with all users. In these systems, our current approach of inserting all commands in every history is not applicable, since users expect that if their command histo-

ries are identical then so should be the state of their user interfaces. Therefore, we change our model to one that inserts into users' command histories only those commands that are shared with the users. More precisely, we introduce the concept of a *sharing* function. Given two users i and j and a command c_n executed by user i, the sharing function determines if the command will be (immediately) executed by the user-interface of user j. Different sharing functions can represent different collaboration schemes. When user i executes the command c_n, we insert c_n in CL_j for each user j if and only if $shared(i, j, c_n)$ returns *true*.

To illustrate undo/redo in non-WYSIWIS systems, consider the interaction shown in Figure 9. In this figure, user pd has executed a command to change the Equipment field. The results of this command are not currently shared with user rxc. (As we shall see in the next section, they will be shared later when an asymmetric collaboration command is executed). Therefore, according to our new command sharing policy, the InsertChar command does not appear in the command list of user rxc. Our undo/redo policy must now define the semantics of undo/redo when command sharing results in different histories. Consider what should happen when user rxc executes an undo. The last

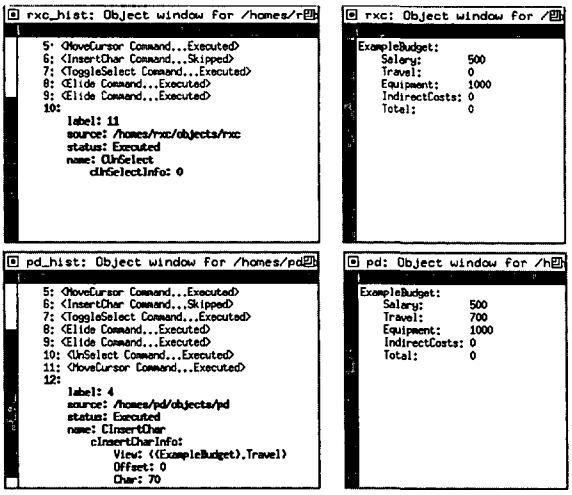

Figure 9: Only shared commands appear in a remote history

command in his history is an UnSelect command, which should be undone. But the UnSelect command is not the last command in the command list of user pd. Fortunately, using the unique labels assigned to commands, our existing undo/redo model can easily determine corresponding commands in command histories of all users and undo all of them. This is illustrated in Figure 10. An undo by user rxc undoes the last command in his interface and the corresponding (non-last) command in the interface of user pd.

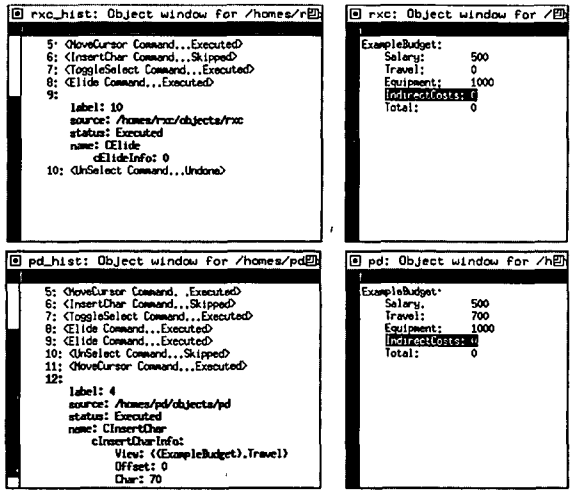

Figure 10: Undo with selective command sharing

Undo/Redo of Asymmetric Commands

So far, our model has addressed undo/redo of only symmetric commands - commands that have the same effect on local and remote user interfaces. Some non-WYSIWIS systems such as Suite [5], also provide asymmetric commands, commands that trigger the execution of previous commands in the local history in the user-interfaces of other users. These commands are asymmetric since the previous commands are not (re)executed in the local user interface.

To illustrate the nature of these commands, consider the Suite `Transmit` command, which transmits to other users commands in the local history that have not been shared with these users. The set of local commands transmitted to a remote user is determined by the Suite coupling scheme [5]. Continuing with the example, if user `pd` executes this command, his change to the `Travel` field, which has not been shared so far with user `rxc` (Figures 9 and 10), is now transmitted to `rxc` (Figure 11). This command is is an asymmetric command because its effects on local and remote user interfaces are different. In particular, a remote execution of it would yield results that have no correlation with the results of a local execution, since the remote execution would transmit pending changes of the remote user which have no correlation with the pending changes of the local user. Our model must now define the command sharing and undo/redo semantics of asymmetric commands.

According to our current command sharing scheme, an asymmetric command would not be inserted into remote command histories since it is not a (strictly) shared command. But that would not allow remote users to return changes they have received from other users. Therefore, we can modify our model to allow a user's history list to contain all commands, both local and

remote, that modify that user's interaction state. But that would not allow a command such as `Transmit` to be inserted into the local command history (since it does not change the local interaction state), thereby not allowing users to withdraw changes they have sent to others. Therefore, we change our command sharing scheme to one that adds a command to the local command history and the histories of all users whose interaction states were modified by the command.

More precisely, we replace the sharing function described earlier with a more general *coupling* function. Given two users i and j and a command c_n executed by user i, the coupling function, determines the set of previous commands to be shared by user j. When user i executes the command c_n, we insert c_n in CL_j if and only if i is equal to j or $coupled(i, j, c_n)$ returns a non empty set.

The commands whose results are transmitted to remote users by a collaboration command are not added by our model into the histories of these users. Figure 11 illustrates our command sharing model. As shown in the figure, the `Transmit` command is added to both histories, but the `InsertChar` command that changed the `Travel` field is not added to the remote history. The remote user must undo the `Transmit` command to undo the effects transmitted by it.

Figure 12 demonstrates the effects of undoing the `Transmit` command in Suite. Recall that the effect of the `Transmit` was to transmit the result of the previous `InsertChar` command to a collaborating user (Figure 11). Thus, undoing of the `Transmit` command removes the effect of sharing the command. As before, either of the two users could have undone the command. Note that upon undoing, although the effects of the collaboration command are undone, the command sharing effects(namely the presense of the command in the remote command list) is not undone. This is the desired result since in single-user undo mode, the undo of the command removes the effects of the command, but does not remove the command from the command list. The presence of the command allows either user to redo the command. The `Transmit` command is an extreme form of an asymmetric command in that it has no effect on the local user-interface state. In general, an asymmetric command may change both local and remote states. For instance, the Suite `MoveCursor` command moves the cursor in the local user-interface and, depending on the coupling, may also transmit previous commands to other users. Our command sharing scheme described above accommodates such commands by including them in the histories of users who see their effects. It is for this reason that the `MoveCursor` command was included in the history windows of our example. Single-user applications do not usually put this command in the history, considering it a relatively insignificant command that users would seldom want to undo. In the multiuser case, several 'insignificant' single-user commands increase their significance since they can transmit significant changes to other users, and thus must be put in undo histories.

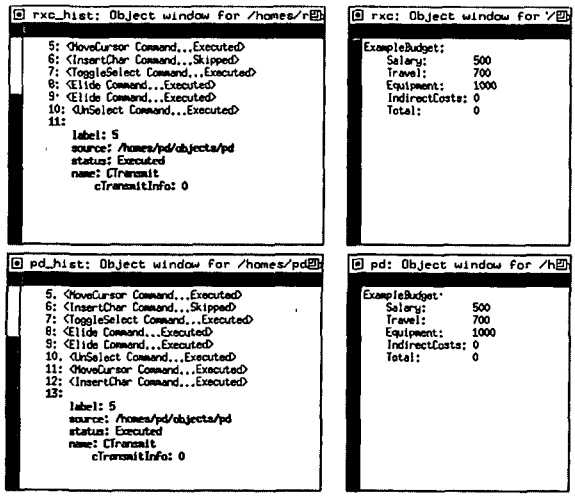

Figure 11: Transmit sends change to Travel

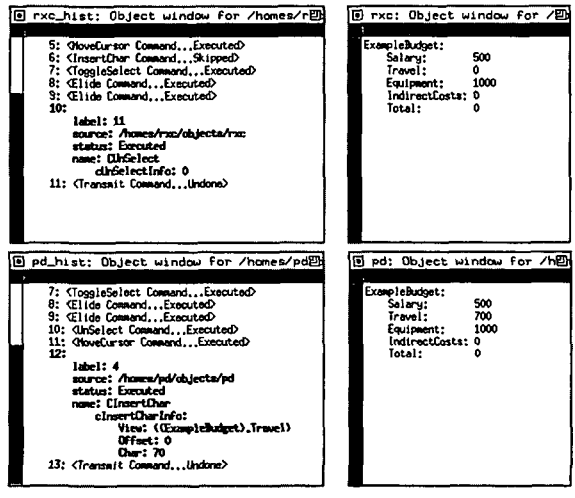

Figure 12: The transmission is undone

Conclusion

The contributions of this paper are a set of undo/redo requirements, a design that meets these requirements, and several detailed examples to motivate and illustrate the requirements and the design.

Multiuser undo/redo has also been addressed by other works [10, 3, 1]. These works and our model, which were developed independently and simultaneously, address mostly different issues arising in the design of multiuser undo/redo. In particular, none of these works addresses the undo issues that arise in a collaborative system allowing the histories of different users to diverge. The bulk of this paper presents these issues and an approach to resolve them. On the other hand, our model does not address some difficult issues raised by other works. In particular, it does not attempt to find conflicts that arise in the undo of a non-last command between the command be undone and subsequent commands in the history that are not be undone [6, 10].

The design described in this paper is only a first step towards a general multi-user undo/redo model and we propose to address the conflict issue in our future work. In particular, we propose to address the consequence of undoing a command that conflicts with only a subset of the histories that share the command. Moreover, we plan to consider extensions to single-user undo models other than the one assumed here. We plan to also explore how availability of a multi-user undo/redo facility may affect design of the access control and concurrency control facilities of collaborative systems. In particular, a multi-user undo/redo model must specify how a user's right to execute undo/redo is derived from his access rights to the data operated upon by these commands. Finally,, due to lack of space, we have not discussed here undo of computation commands: commands that commands that request carrying out of computation. For instance, we have not discussed what happens if a user asks the application to compute the indirect costs of the budget and then wants to undo this computations. We address this issue in [4].

Acknowledgments

We would like to thank the reviewers for their detailed comments, which helped us improve the presentation of the paper. We would also like to thank the reviewer who pointed out the problem of a command conflicting with only a subset of the histories that share it. This research was supported, in part, by a grant from the Software Engineering Research Center at Purdue University, a National Science Foundation Industry/University Cooperative Research Center (NSF Grant ECD-8913133), and by National Science Foundation Grants IRI-9208319, IRI-9496184, and IRI-9408708.

References

[1] G. D. Abowd and A. J. Dix. Giving Undo Attention *Interacting Comput.*, 4(3):317–342.

[2] James E. Archer, Jr., Richard Conway, and Fred B. Schneider. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages and Systems*, 6(1):1–19, January 1984.

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

[4] Rajiv Choudhary and Prasun Dewan. Multi-user undo/redo. Technical Report SERC-TR-125-P, Purdue University, 1992.

[5] Prasun Dewan and Rajiv Choudhary. Flexible user interface coupling in collaborative systems. In *Proceedings of the ACM CHI'91 Conference*, pages 41–49. ACM, New York, 1991.

[6] W. D. Elliot, W. A. Potas, and A. van Dam. Computer assisted tracing of text evolution. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 533–540, 1971.

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

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

[9] C.M. Neuwirth, D.S. Kaufer, R. Chandhok, and J. H. Morris. Computer support for distributed collaborative writing: defining parameters of interaction. In *Proceedings of CSCW'94*, pages 145–152, October 1994.

[10] Atul Prakash and Michael J. Knister. A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction*, 1(4):295–330, December 1994.

[11] Herold Thimbleby. *User Interface Design.* ACM, 1990.

[12] Jeffery Scott Vitter. US&R: A new framework for Redoing. *IEEE Software*, 1(4):39–52, Oct. 1984.

[13] Haiying Wang and Mark Green. An event-object recovery model for object-oriented user interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 107–115, Nov. 1991.

[14] Xerox PARC, Palo Alto, CA. *INTERLISP Reference Manual*, December 1975.