Multi-User Undo/Redo

Rajiv Choudhary and Prasun Dewan
Department of Computer Sciences
Purdue University
West Lafayette IN 47907
rxc@cs.purdue.edu and pd@cs.purdue.edu

June 3, 1992

Abstract

We have developed a multi-user undo/redo model by extending an existing single-user undo/redo model. The model, consisting of a semantic model and an implementation model, is applicable to general multi-user programs including programs offering both WYSIWIS and WYSINWIS interaction, floor control and concurrent interaction, and atomic and non-atomic broadcast. The semantic model constructs the command history of a particular user by combining all commands, including both local and 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 arbitrary commands in a command history including commands executed by other users and commands that explicitly transmit information to other users. The implementation model divides the task of implementing undo/redo between generic dialogue managers and application programs. It divides application programs into three increasingly complex classes and requires increasing levels of undo/redo awareness from these classes.

1 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 of alternatives [13]. It is offered in some form or another by most popular single-user programs [1, 4, 5, 7, 8, 10, 13, 14, 15, 16, 17]. But none of the multi-user programs known to use 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 presense of many users. Finally, users of a multi-user program can

make new kinds of errors, which we call "coupling errors", by sharing the results of their commands with the wrong subset of users.

One reason for the absence of undo/redo from previous multi-user user-interfaces is the lack of semantic and implementation undo/redo models for multi-user programs. Therefore, we have developed first-cut versions of these models. The semantic 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 [1]. It constructs the command history of a particular user by combining all commands, including both local and 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 arbitrary commands in a command history including command executed by other users and commands that explicitly transmit information to other users.

The implementation model offers programmers a framework for implementing our semantic model for a particular multi-user program. It divides the task of implementing undo/redo between generic dialogue managers provided by the system and application programs written by programmers. The model classifies application programs into three increasingly complex classes according to how they respond to user commands and requires increasing levels of undo/redo awareness from these classes of programs.

These models are applicable to multi-user programs offering a variety of functionality, coupling, concurrency control, and broadcast schemes. In particular, they are applicable to multi-user text and 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 broadcast schemes supporting both atomic and non-atomic broadcast.

We have implemented these models as part of a system called Suite. In this paper, we motivate, describe, and illustrate our (semantic and implementation) undo/redo model using the concrete example of Suite. The remainder of the paper is organized as follows. Section 2 defines and illustrates the single-user undo/model used as a basis for our model. It also introduces an example, which is used throughout the paper to illustrate the various properties of our model. Section 3 and Section 4 motivate, describe, and illustrate the semantic and implementation components, respectively, of our model. Section 5 presents conclusions and directions for future work.

2 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 [15]. 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. To illustrate this model and our extensions to

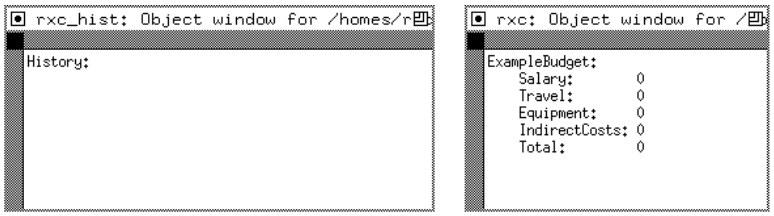

Figure 1: Single-user text editing: Initial display

it, consider an interactive session with a Suite dialogue manager. Figure 1 shows the initial display in a single-user text editing session in Suite. The history list is initially empty and the initial text is shown with the fields initialized to 0. [1] Figure 2 shows the dialogue manager display and the history list after the user edits two fields in the dialogue manager by moving the cursor there 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. [4]

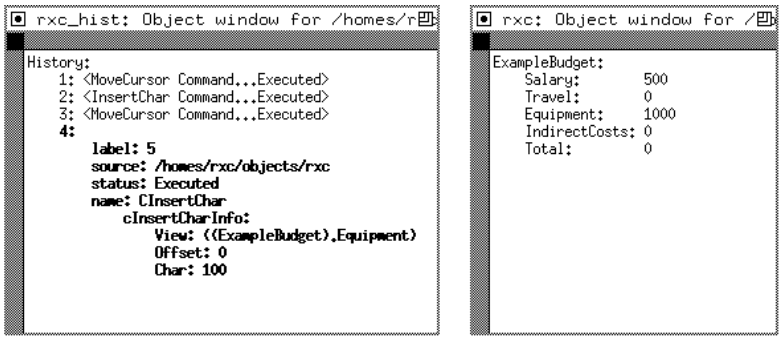

Figure 2: Single-user text editing: After editing

Figure 3 shows the dialogue manager and the history list after executing the undo command. As each undo is executed, the last executed command in the history list is

[1] In suite, users do not edit simple lines of text. Instead, the dialogue manager display consists of a number of labeled fields and the user edits the text of these fields. But this is irrelevant to our present discussion.

[2] It is not conventional to treat MoveCursor as a command in text editors. However in Suite, a number of other actions can be tied to the moving of the cursor. Thus, Suite must treat MoveCursor as a command.

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

[4] 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.

marked undone and the effects of the command execution are removed from the dialogue manager display. Executing two more undo commands returns the dialogue manager to the state represented by Figure 1 with all four commands undone, while issuing two redo commands returns the dialogue manager to the state depicted in Figure 2.

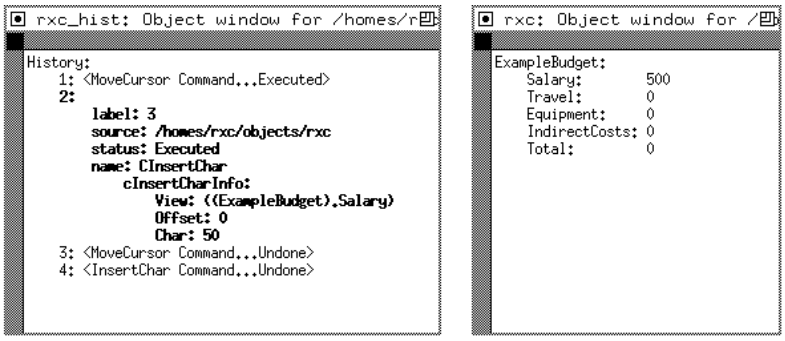

Figure 3: Single-user text editing: After undo

3 Designing Multi-User Undo/Redo

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

- construction of command histories using local and remote commands whose effects are visible.
- unique identification of commands in command histories for coordinated undo/redo.
- ability to undo/redo an arbitrary command in the command history.
- undo/redo of communication and computation commands.

3.1 Basic Multi-User Undo/Redo

Here is a definition of a multi-user undo/redo model, which is a simple extension of the single-user undo/redo model above: To build the command histories of users, all commands are shared by every user and commands appear in the same sequence in all command histories. When an undo/redo command is executed, the last command in every command history is undone/redone.

We illustrate the model by using the example of a multi-user text editing session in Suite. Figure 4 shows command lists and dialogue manager displays of two collaborating users. User `rxc` is the active users and executes all commands. When user `rxc` issues an undo, the resulting state is shown in Figure 5. Note that the last command has been undone in both the command histories.

3.2 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 floor control

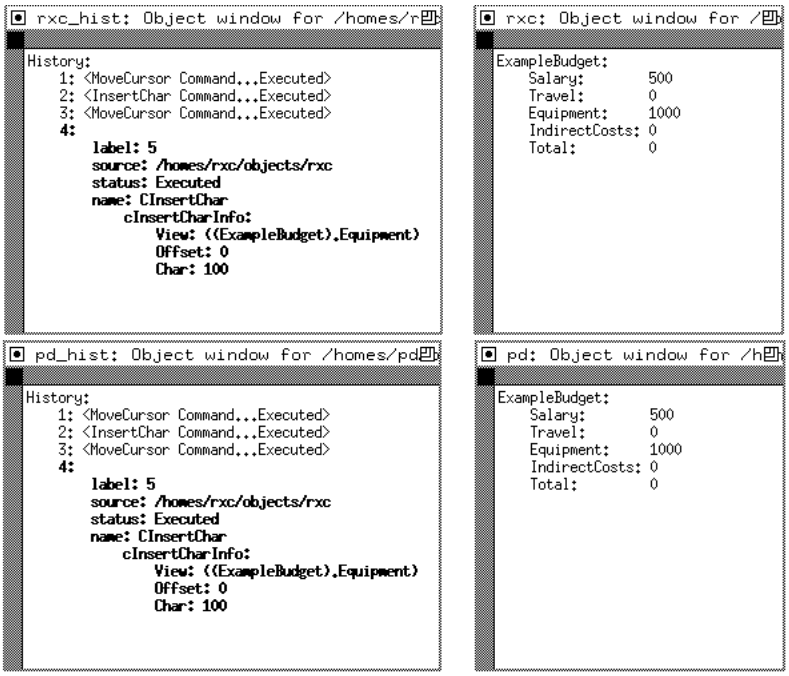

Figure 4: Command Sharing

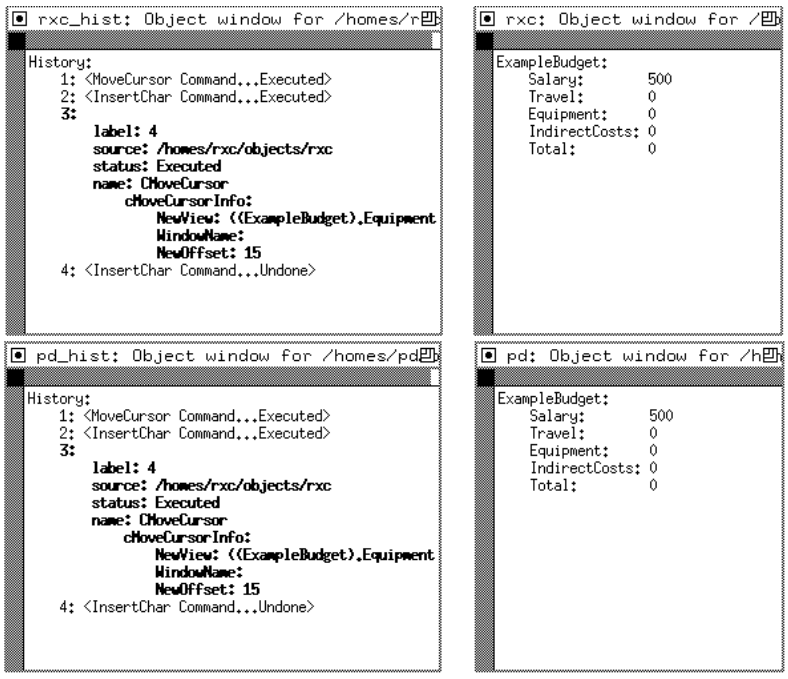

Figure 5: Undo in Multi-User Undo/Redo Model

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 a system, 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 commands may be ordered differently in different command lists and therefore the last command may be different in command histories of different users. Now, when a user issues an undo/redo command, it may operate on different commands in different command lists(since the last command in different lists is not the same). But this may not be the same command for all users. In short, chaos would result with different commands being undone/redone for different users in response to an undo/redo request.

Our solution to this problem is to provide a way to identify all copies of a command in different command histories with a unique identifier. Then semantics of undo/redo need no longer 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 identifier associated with each command.

3.3 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 becomes the last command in the command history. According to our model so far, when an undo is invoked, this is the command that is undone when the undo is invoked. Thus our current model of undo/redo that always undoes/redoes the last command is not entirely satisfactory in such a system. For illustration, consider the multi-user text editing session depicted in Figure 6. Suppose user `pd` decides to undo the last command in the command list, but before he can execute the undo command, user `rxc` executes a number of new commands as shown in Figure 7. Now if user `pd` executes the undo, the `Elided` command gets undone instead of the desired command `InsertChar`.

We rectify this problem by slightly modifying our undo/redo model so that a user can indicate by marking a command in the command list, which command he wants undone/redone. 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 command is undone/redone and skipped, and finally all other commands are restored to their previous status.

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), the desired command can be undone by invoking the undo operation while the appropriate command

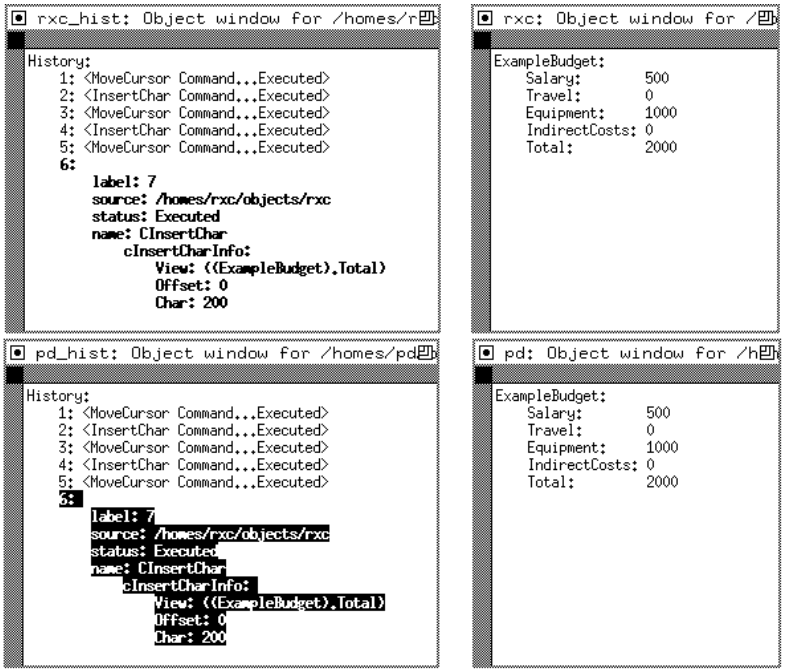

Figure 6: Marking a Command in the Command History

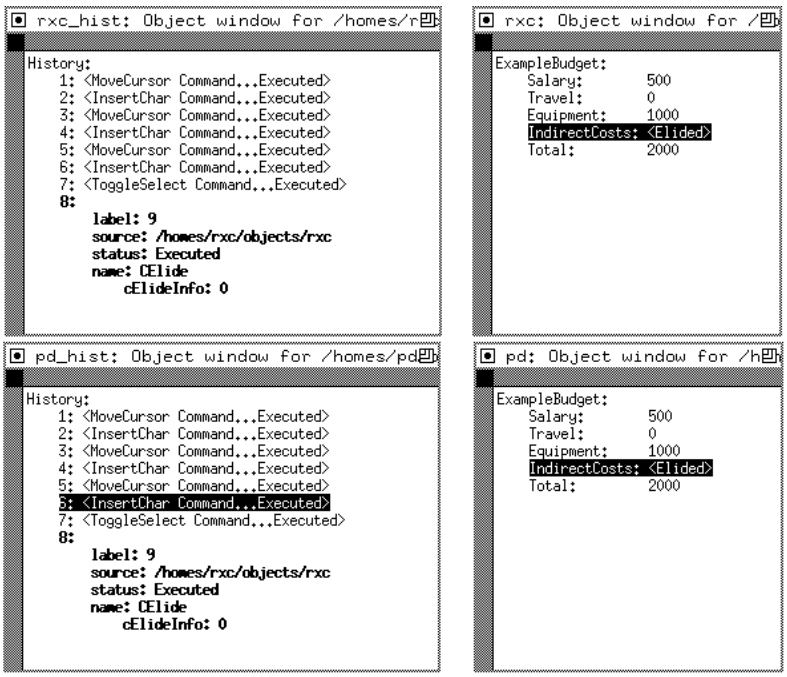

Figure 7: Marked Command after Remote Command Execution

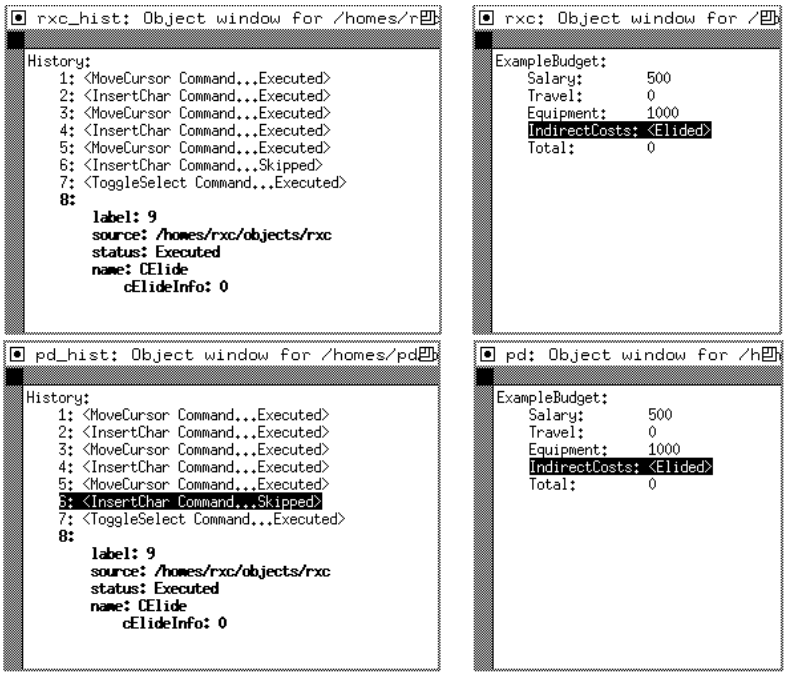

Figure 8: Undo-by-reference

is selected(Figure 8).

3.4 Selective Command Sharing

Now consider a system that provides the flexibility to choose with whom to share results of executed commands. In such a system, results of all commands are not shared with all users. If the system continues to insert all commands in every user's command list, the state of different user's interfaces would be different , whereas users expect that if their command histories are identical then so should be the state of their user interfaces.

In such a system, we construct command history of each interface by including only those commands whose results are shared with that user. Thus commands whose effects are shared with only a subset of users appear in the commands history of that subset of users only. One such interaction is shown in Figure 9.

In this figure, user `pd` executes a command whose results are shared with other users only upon explicit request. Thus the `InsertChar` command that edited the `Travel` field is not shared with user `rxc`. Therefore, using our new command sharing policy, the `InsertChar` command does not appear in the command list of user `rxc`. This semantics of command sharing results in different histories for users who do not share result of all commands. How can our undo/redo model be applied to this scheme?

Consider what happens when user `rxc` executes an undo. The last command in the history list is the `UnSelect` command which should be undone. But note that `UnSelect` is not the last command in the command list of user `pd`. Fortunately, using the unique identifiers assigned to commands, our existing undo/redo model can easily

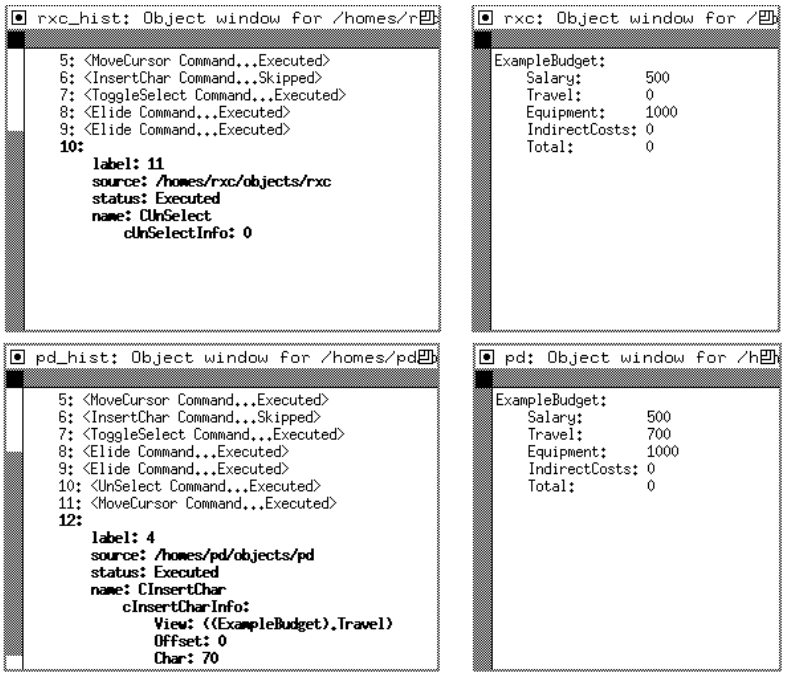

Figure 9: Selective Command sharing

determine corresponding commands in command histories of all users. By extending our undo/redo model to require coordinated undo of corresponding commands in different command histories is sufficient to provide the desired semantics. Specifically, the undo command undoes the last command(unless it is an undo-by-reference) in the command history of the issuer of undo and in addition requests undoing of the corresponding commands at the interfaces of all other collaborating users that share the command.

As illustrated in Figure 9, due to non-WYSIWIS nature of coupling, the command required to be undone at the interface of user `pd` is not the last command. To undo the desired command without affecting the semantics of other commands, the request is treated as semantically equivalent to a set of user actions where the local user successively undoes commands until the desired command is undone, then skips the specified command and redoes the rest of the commands in the interface. This scheme for undoing the not-the-last command is similar to the undo-by-reference command seen in the last section. This semantics of the undo command in the non-WYSIWIS coupling scheme is shown in Figure 10.

3.5 Undo/Redo of Collaboration and Computation Commands

So far, our model has addressed undo/redo of only the commands that change the user interface state. In general, multi-user programs also provide (a) collaboration commands : commands that request communication of values to other users, (b) computation commands : commands that request carrying out of computation. A general multi-user undo/redo model must define the semantics of undo/redo on these commands.

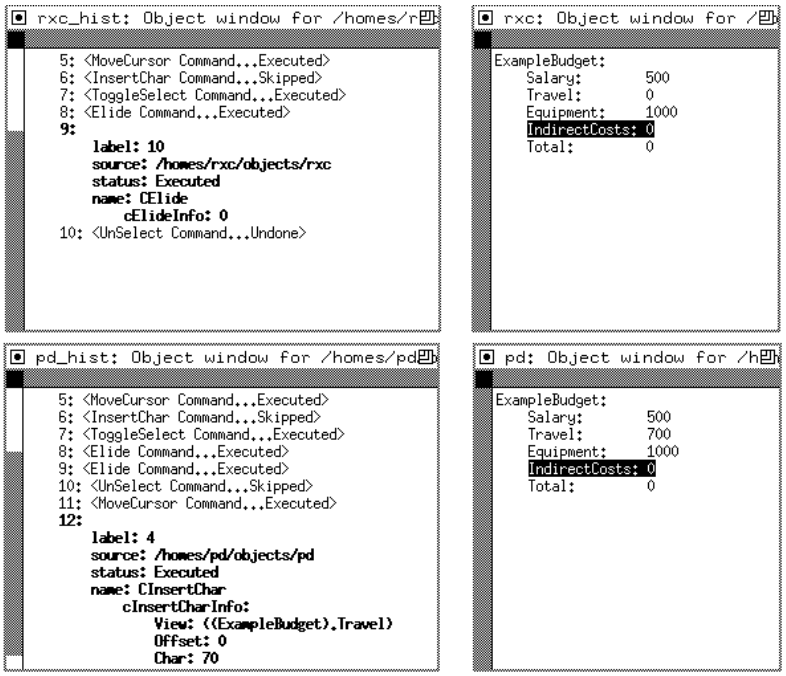

Figure 10: Undo with Selective Command Sharing

Our model of undo/redo provides support for undo/redo of not only interface commands but also collaboration and computation commands. We illustrate how the model handles these commands with examples.

The only effect of a transmission command is to share results with other users. What results are shared and with whom is determined by the coupling scheme. Thus the actual effect of transmission commands depends upon the coupling scheme. Since undoing of a command requires us to undo all the effects of a command, undoing a transmission command requires that the effects of sharing be removed. As an example, Figures 11 and 12 demonstrate the effects of executing and undoing the `Transmit` command in Suite. Specifically, Figure 11 shows that the effect of the `Transmit` command is to transmit the result of the previous `InsertChar` command to a collaborating user. Thus undoing of the `Transmit` command removes the effect of sharing the command.

A computation command computes values and updates the interface state. What computations are carried out and what values are changed in the interface as a result of a computation depends upon the actual computation being invoked. [5]

Continuing with our example, consider the interaction of Figure 13 in which one of the users edits the displayed data and executes the `Accept` command, which is a computation command in Suite. As a result of this command, the entity being edited computes and updates the values in the fields `IndirectCosts` and `Total`. Since we associate these changes in the display with the `Accept` command, undoing of the `Accept`

[5] In our system, it is possible to remove all effects of a collaboration command, but it is not always possible to remove all effects of a computation command. In the next section we discuss how effects of general computation can be reversed.

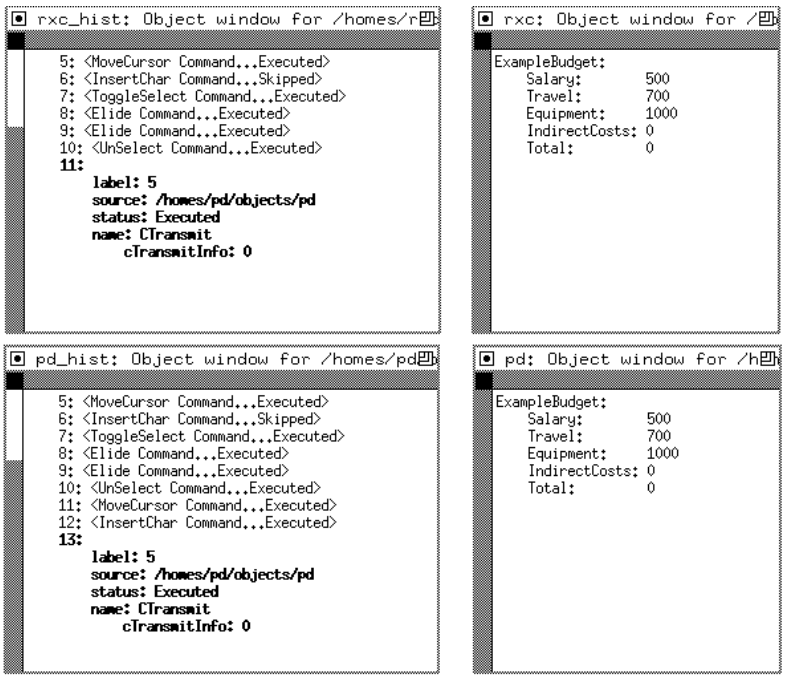

Figure 11: Effects of a Transmit Command

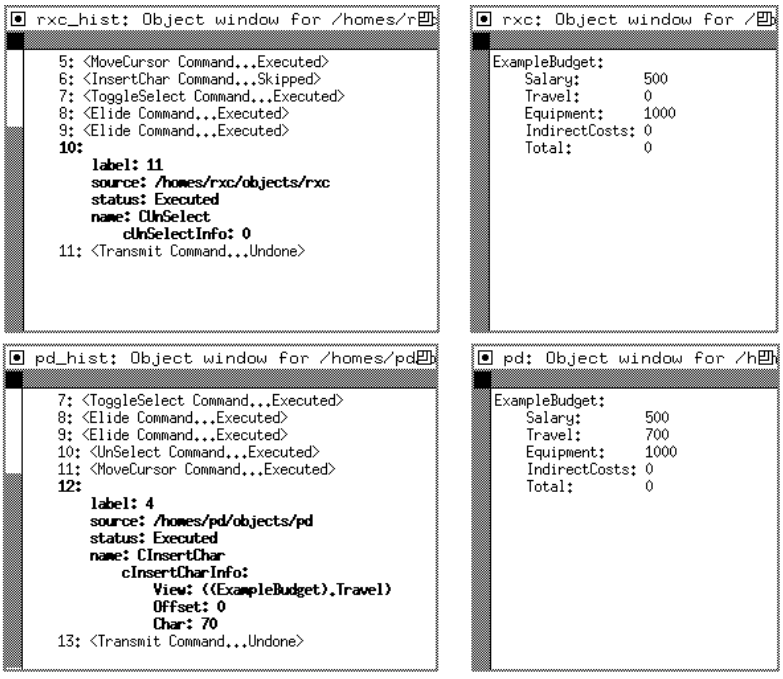

Figure 12: Undo of a Transmit Command

command also undoes the updating of the displays that occured(Figure 14).

Note that upon undoing, although the effects of the coupling 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.

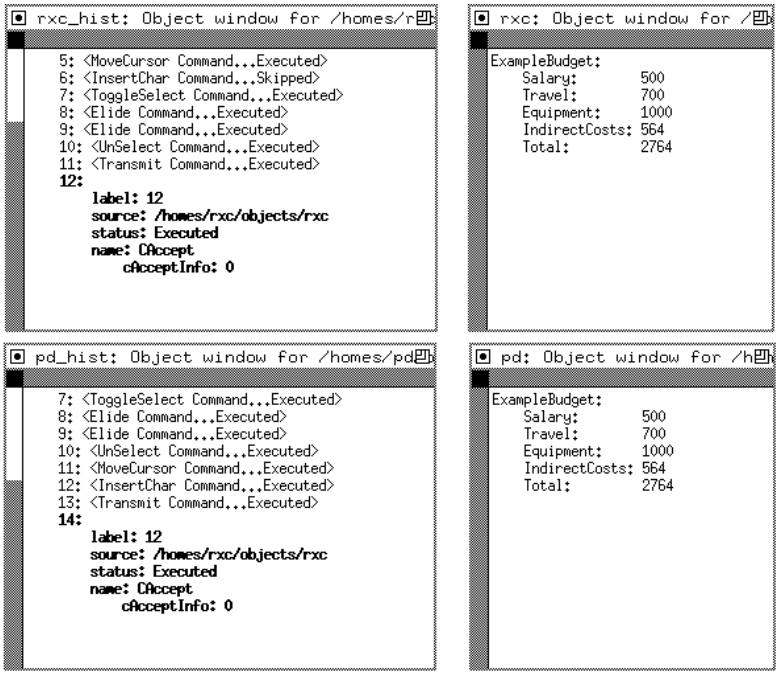

Figure 13: Effects of an Accept command

4 Undo/Redo Implementation Model

One approach to implementing the undo/redo semantics described above is to require each program to implement all aspects of the model. However, this makes the overhead of supporting undo/redo for a program very high. Therefore, we have devised a high-level implementation model which divides the responsibility of undoing/redoing commands between application programs and the system. This model is based on the Suite model for multi-user programs, which divides the responsibility of "doing" commands between the application program and the system.

Suite divides a multi-user program into an application program and multiple generic dialogue managers, each of which interacts with a particular user [2, 3]. To illustrate the Suite multi-user program model, consider how the multi-user program described in the previous section is implemented in Suite. The application program presents a set of *active values* for editing to the the system supplied dialogue manager. The application program also associates *update handlers* to react to changed values of active values.

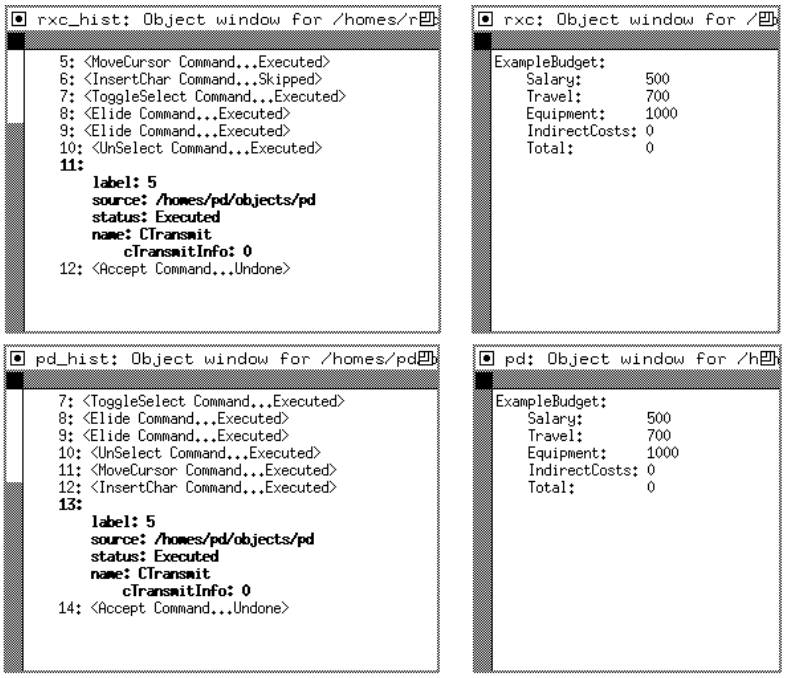

Figure 14: Undo of an Accept command

When a user edits an active value and commits it, the dialogue manager invokes the appropriate update handler to inform the application program of the changed value. All functional aspects of the multi-user program are carried out by computations in the update handler. Below we show the update handler of the program described in the previous section.

```
typedef struct {
  int Salary, Travel, Equipment;
  int IndirectCosts, Total;
} Budget;

void UpdateBudget(name, val)
  char *name; Budget *val;
{
  val->IndirectCosts = (val->Salary + val->Travel) * OVERHEAD;
  val->Total = val->Salary + val->Travel + val->Equipment + val->IndirectCosts;
  Dm_Update ("ExampleBudget", "Budget", val);
}
```

We support our model of undo/redo by undoing/redoing the actions taken by dialogue managers as well as the application programs in response to a user command. For a multi-user program to implement the multi-user undo/redo model, the program need only provide support for the undo/redo of actions it takes when its update handlers are called. If these actions cannot be undone/redone, the usefulness of the undo/redo

model is restricted only to multi-user text editing.

To ease the burden of application programs from having to keep track of all actions they invoke, Suite has augmented the dialogue manager-application program interface with an *update undo handler*. The application programs must implement update undo handlers as inverses of the corresponding update handlers. The update undo handler is invoked by a dialogue manager when a command is undone whose execution resulted in the invocation of the update handler. Thus the responsibility of an application program in supporting the undo/redo model reduces to providing update undo handlers. All other aspects of the undo/redo model are handled by system supplied components.

We can classify the application programs according to the kind of operations they invoke in their update handlers. This also determines how much effort the programmer of an application program has to expand.

4.1 Undo Unaware Programs

If an application program carries out computation after receiving changed values from a dialogue manager but does not maintain any state information itself, the task of undoing is made very easy. In such an application , state changes are made only in the state of dialogue managers interacting with the application program. As illustrated in the previous section, the undo support necessary for such updating of dialogue manager displays is automatically provided by the dialogue manager. Thus application programs in this class do not need to participate in any undo/redo handling and are termed *undo unaware* application programs. The application whose update handler is presented above is one such program.

4.2 State Caching Programs

Often a program needs to cache the value of the active values in its own data. For instance, in our example the program needs to keep a copy of the values being edited to support a program, `GetLatestBudget`, that can fetch the latest active values from the application program. To support such a function, the update handler needs to copy the changed values into a variable local to the application program. This is shown below in the modified version of the update handler, where a copy of the updated value is retained in the program variable `ExampleBudget`.

```
Budget ExampleBudget;
void UpdateBudget(name, val)
  char *name; Budget *val;
{
  val->IndirectCosts = (val->Salary + val->Travel) * OVERHEAD;
  val->Total = val->Salary + val->Travel + val->Equipment + val->IndirectCosts;
  ExampleBudget = *val;
  Dm_Update ("ExampleBudget", "Budget", val);
}
```

With such an update handler, it is not sufficient to undo the updating of dialogue manager displays. Undoing of the `Accept` command must restore older value of the active value to the program variable.

This undoing is easily achieved by cooperation between the dialogue manager and the application program. We note that the dialogue manager needs to maintain older copies of the active values for undoing of its own actions. The application program takes advantage of this by setting the update undo handler to be the same as the update handler. Thus when an dialogue manager is processing an undo request for a command that resulted in an update call, it calls the update undo handler. This results in a call to the update handler which can restore the old value of the program variable.

In this class of programs, the programmer does not need to write any new code to participate in the undo/redo model. However unlike the undo unaware programs, these programs need to be aware of update undo handlers, thus requiring a higher level of undo awareness.

4.3 Undo Aware Programs

Since collaborative applications can be arbitrary programs, sometimes the update handlers needs to carry out other tasks in addition to caching the latest value and updating displays. Consider the update handler shown below, which makes a record of the active value in a text file whenever the value is updated.

```
Budget ExampleBudget;

void UpdateBudget(name, val)
  char *name; Budget *val;
{
  FILE *fp;
  val->IndirectCosts = (val->Salary + val->Travel) * OVERHEAD;
  val->Total = val->Salary + val->Travel + val->Equipment + val->IndirectCosts;
  fp = fopen("record", "a");
  fprintf(fp, "Salary %4d Travel %4d Equipment %4d IndirectCosts %4d Total %4d\n",
  val->Salary,val->Travel,val->Equipment,val->IndirectCosts, val->Total);
  fclose(fp);
  ExampleBudget = *val;
  Dm_Update ("ExampleBudget", "Budget", val);
}
```

In general the tasks carried out by an update handler could be arbitrarily complex.

When an dialogue manager attempts to undo a command that resulted in an invocation of an update handler(an accept command), it checks to see if an update undo handler is defined for the active value. If one is defined, it is invoked with the old and the new copies of the active value for this update handler. In this class of programs, the update undo handler must arrange to reverse the steps carried out during the update handler.

Below we show the update undo handler for the update handler shown above. Since the display updating requests invoked in the update handler are undone by the dialogue managers themselves, the update undo handler needs to only restore the record file and the cached value.

```
UndoBudget(path, old_val, new_val)
  char *path; Budget *old_val; Budget *new_val;
{
  int fd; long len;
  fd = open("record", O_RDWR);
  len = lseek(fd,0L,SEEK_END);
  ftruncate(fd, len-69);
  close(fd);
  ExampleBudget = *old_value;
}
```

Such a simple interface for undoing the accept command is possible because of the controlled dialogue manager-program interface provided in Suite where the program state can only be updated through the update handlers. These facilities work for most update handlers but there are applications for which the program must implement it's own undo. For example, any update handler that relies on state of the program other than the updated value must itself arrange to keep logs of such state change because the dialogue manager can provide no help in such reversal.

So far we have only discussed undoing an update handler invocation. Since the semantics associated with redo are same as that of reexecuting an update, when an accept command is redone, the dialogue manager simply reinvokes the update handler. Thus no other support is required for the redoing of update handler invocation.

5 Conclusion

Undo/redo is indispensable in multi-user interfaces. We have developed a model of multi-user undo/redo which is useful across a range of coupling schemes, floor control policies and (atomic and non-atomic) broadcast schemes. We have shown that it is possible and desirable to partition the support required for multi-user undo/redo between the interface component and the application program. The result is a general model in which many application programs have to do little or no work to support the undo/redo facilities.

Our undo/redo 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. As the example interactions in the paper illustrate, our model has several pleasant properties:

Compatibility Multi-user undo/redo behaves like single-user interactive undo/redo when only one user is interacting with the system.

Undo/Redo Independence Multi-user undo/redo does not require attention or intervention of all users in a collaborative session. In particular, passive users remain synchronized with the state of a collaborative session without executing any commands themselves. Thus the model keeps the number of metacommands that must be executed by users to a minimum.

Collaboration in Undo/Redo It is possible to undo commands issued by other users. This is a direct analogy of the ability of collaborative users to collaborate by executing commands on behalf of other users.

Automation of Undo Support The model supports automation since it requires little or no code to support undo/redo for many programs.

Multi-user undo/redo support described in this paper is only a first step towards a multi-user undo/redo model. We plan to explore how availability of a multi-user undo/redo facility may affect design of the access control and concurrency control facilities of collaborative systems. We also plan to explore how command execution, coupling schemes and multi-user undo/redo can be combined to provide comprehensive session management facilities in collaborative applications.

The implementation model described in this paper must rely upon the analysis by the programmer to determine to which class of application programs a particular program belongs. We would like to explore whether, with appropriate support from a compiler, we can automate the task of classifying application programs into one of the three classes. It would be useful to explore the possibility of further automation of undo/redo support required from application programs.

References

[1] 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.

[2] 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.

[3] Prasun Dewan and Rajiv Choudhary. Primitives for programming multi-user interfaces. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 69–78, November 1991.

[4] 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.

[5] C. A. Ellis, S. J. Gibbs, and G. Rein. Design and use of a group editor. Technical Report STP-414-88, MCC Software technology Program, 1988.

[6] J. Robert Ensor, S. R. Ahuja, David N. Horn, and S. E. Lucco. The rapport multimedia conferencing system – a software overview. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 52–58, March 1988.

[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] J. R. Horgan and D. J. Moore. Techniques for improving language-based editors. *ACM Software Engineering Notes*, 9(3):7–13, May 1984.

[9] J.C. Lauwers and K.A. Lantz. Collaboration awareness in support of collaboration transparency:requirements for the next generation of shared window systems. In *Proceedings of ACM CHI'90*, pages 303–311, April 1990.

[10] Cai Linxi and A. Nico Habermann. A history mechanism and undo/redo/reuse support in aloe. Technical Report CMU-CS-86-148, Department of Computer Science, Carnegie-Mellon University, 1986.

[11] Judith S. Olson, Gary M. Olson, Lisbeth A. Mack, and Pierre Wellner. Concurrent editing: The group's interface. In *Human Computer Interaction – INTERACT '90*, pages 835–840, 1990.

[12] Mark Stefik, Gregg Foster, Daniel G. Bobrow, Kenneth Kahn, Stan Lanning, and Lucy Suchman. Beyond the chlakboard: Computer support for collaboration and problem solving in meetings. *Communications of the ACM*, 30(1):32–47, January 1987.

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

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

[15] 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, November 1991.

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

[17] Yiya Yang. Experimental rapid prototype of undo support. *Information and Software Technology*, 32(9):625–635, November 1990.