

# A History-Based Macro By Example System

David Kurlander\*

Steven Feiner

Department of Computer Science  
Columbia University  
New York, NY 10027

E-Mail: {djk, feiner}@cs.columbia.edu

## ABSTRACT

Many tasks performed using computer interfaces are very repetitive. While programmers can write macros or procedures to automate these repetitive tasks, this requires special skills. Demonstrational systems make macro building accessible to all users, but most provide either no visual representation of the macro or only a textual representation. We have developed a history-based visual representation of commands in a graphical user interface. This representation supports the definition of macros by example in several novel ways. At any time, a user can open a history window, review the commands executed in a session, select operations to encapsulate into a macro, and choose objects and their attributes as arguments. The system has facilities to generalize the macro automatically, save it for future use, and edit it.

**KEYWORDS:** Macros, demonstrational techniques, histories, graphical representations, programming by example.

## INTRODUCTION

When applications are made *extensible*, the entire user community benefits. Individuals can customize their applications to the tasks that they often encounter, and experts can encapsulate their expertise in a form that less skilled users can exploit. By writing a macro or program, users can extend an application to perform tasks not included in the original interface; however this typically requires both programming skills and familiarity with the application's extension language. Systems with a *macro by example* or *programming by example* component generate code auto-

matically in response to tasks demonstrated by the user through the application's own interface [12]. These systems make the benefits of extensibility accessible to the entire user community.

Many applications, such as GNU Emacs [14], have a macro by example facility, but lack a visual representation for the macros. Without a visual representation, it is impossible to review the operations that compose the macro. When there is an error in such a macro, the macro must be demonstrated once again from scratch. If an error occurs in a macro without a visual representation, the system cannot provide a comprehensible error message explaining which step generated the error.

Though visual representations are clearly important for a macro by example facility, many systems omit this component since it is problematic how to statically display commands executed through an application's graphical user interface. We have developed a technique for visually representing such commands. Previously we used a representation, called *editable graphical histories*, to provide a visual record of commands executed in a session with a graphical editor [6]. We have extended this technique to represent macros by example, and support the definition and editing of these macros. Here we introduce a macro by example facility that uses editable graphical histories as its visual representation, and discuss the many ways that the macro facility takes advantage of these histories.

The macro by example system described in this paper is implemented as part of Chimera, an editor system with modes for editing 2D illustrations, user interfaces, and text [8]. Macros can currently be defined in both the illustration and user interface editing modes. All of the examples in this paper are generated from the PostScript output of Chimera and its macro by example facility.

---

\*Author's current address: Microsoft Research, One Microsoft Way, Redmond, WA 98052-6399.

In the next section we discuss how other example-based systems have dealt with the issue of representation. Then we briefly discuss editable graphical histories, and in the rest of the paper focus on how they support a macro by example facility.

### RELATED WORK

Since most programming by example research has dealt with problems other than representation, many systems ignore this issue. Peridot [11] and Metamouse [10] provide highlighting or feedback for individual program steps, however they depict a single step at a time with no visual representations for the complete procedures which they infer. A more comprehensive graphical representation would allow the user to quickly examine and edit any step.

Representing commands in text-based systems tends to be easier, since the textual commands themselves form a convenient representation. Tinker, a text-based programming by example facility, has a textual audit-trail of steps used in constructing procedures [9]. To edit the demonstrated procedure, the user can either textually edit these steps or the resulting LISP procedure. Tweedle, a graphical editor with both a WYSIWYG view and a textual code view, allows procedures to be generated in both views [1]. However, to edit a procedure, the user must be able to understand the code view. In the MIKE UIMS, graphical macros can also be defined by demonstration [13]. In this system macros can be defined and edited largely in demonstration mode, but the visual representation of graphical commands is textual.

A programming by example component of SmallStar, a miniature version of the Star user interface, adopts a mixed text and iconic representation for macros [4]. The system uses a predefined set of icons or pictographs to represent entities on the desktop. The domains for which our system is targeted are more graphical in nature, so prefabricated icons will not suffice. As will be discussed in more depth in the next section, our approach is to generate graphics automatically to represent the operations in the macro.

All the programming by example systems discussed thus far have special operations to start and stop recording events. In our system, operations are always being recorded by an undo/redo mechanism. When users realize that a set of operations that they had performed are generally useful, they can always open up a history window and encapsulate the interesting operations into a macro. A programming by example system named EAGER also generates macros from a history [3]. It constantly monitors the command stream for repeated operation sequences. When a repetitive task is detected, the system presents feedback that indicates the tasks it anticipates, and when users are confident in EAGER's predictions, they can have it automatically generate a generalized procedure. However, this procedure has no graphical representation.

### EDITABLE GRAPHICAL HISTORIES

Command histories in our graphical user interface are represented visually using a comic strip metaphor. Actions in the history of the interface are distributed over a sequence of panels. We refer to this representation as an *editable graphical history* [6]. Figure 1 shows a graphical history representation of the commands that added text and a drop shadow to the horizontal oval labeled GENERATOR in Figure 2.

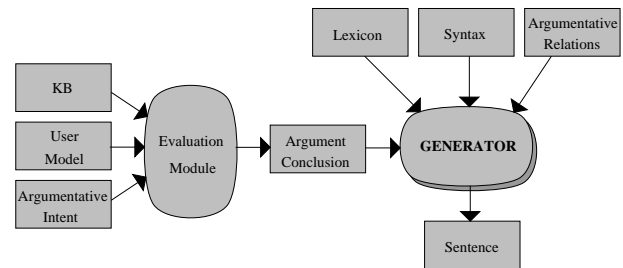


FIGURE 2. A technical illustration created with Chimera.

Editable graphical histories use several techniques to make a sequence of commands more comprehensible. The panels are graphical, and use the same visual language as the interface itself. Since the user of the system is already familiar

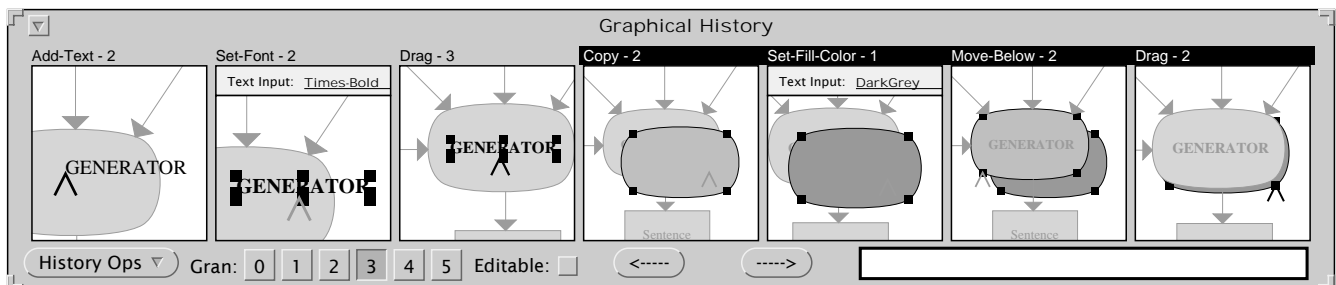


FIGURE 1. A graphical history representation of steps that add text to an oval and create a drop shadow. These steps were used in creating part of Figure 2. Panels whose labels are shown in reverse video have been selected by the user to create the macro shown in Figure 3.

with the interface's conventions, it is easy for them to interpret the histories.

Related commands are coalesced into single panels by pattern matching rules. This makes the histories more compact, but also makes them easier to interpret, since instead of showing physical commands the system shows logical commands. For example, the first panel of Figure 1, represents commands that have added a text label to the oval. The two commands move the caret (the software cursor) to the desired position for the text, and then insert the characters. The second panel changes the font of this text and includes both an object selection and Set-Font command. An interactive elaboration facility can be used to expand higher-level panels into their lower-level components.

If we were to shrink down the entire screen to fit in each panel, then it would be difficult to see the changes that the panels represent. Instead the system shows only those objects that participate in the operation, as well as a little scene context to indicate where on the screen the operation was performed. Each panel of Figure 1 contains only a subset of Figure 2 or of Chimera's control panel. For example the second panel includes the Text Input widget, in which the name of the new font was typed, as well as the selected text. The history mechanism adds other objects to the panels, only to provide context. Objects in the panels are rendered according to their roles in the explanation. Currently the rendering pass subdues contextual objects by lightening their colors. This usually makes it easy to distinguish these objects from those that participate in the operation.

### MACRO DEFINITION

Macro definition in Chimera consists of two primary passes. In the first pass the task is demonstrated using the regular user interface. The dialogue for this pass is indistinguishable from regular user-interaction—there are no special commands to execute, and no special operations to start and

stop macro recording. Since people often do not think of defining a macro until they have executed the steps at least once, the commands may already have been demonstrated, and no additional repetition is necessary.

In the second pass, the demonstrated sequence of commands is supplemented with additional information to convert this sequence into a macro. The commands executed in this pass are different than those forming the ordinary application dialogue. This pass includes selecting a set of previously executed commands to encapsulate into the macro, selecting arguments for the macro, generalizing the commands to work in other contexts, and debugging and saving the macro. Splitting macro definition into a demonstrational step and a generalization step was first done by Halbert in SmallStar [4]. It has the advantage that the demonstrational pass of the macro is purely demonstrational, and certain constructs, such as conditionals and loops, which are difficult to add by demonstration, can be introduced in a separate non-demonstrational pass.

However, unlike SmallStar which had special commands to start and stop recording a macro, commands in Chimera are always being recorded by an undo/redo facility. At any time, users can open up the history window, review the commands executed in a session, or undo and redo some of these commands. They can also select a set of commands to be incorporated into a macro. The history of Figure 1 shows a set of commands that add text to an oval and construct a drop shadow for the oval. Recognizing that drop shadows are necessary for other objects in the scene, we can now extract the relevant panels from the history and turn them into a macro.

First we select these panels using the mouse, and as feedback, the selected panel labels appear in reverse video, as shown in Figure 1. The *macroize* operation, which is executed next, takes a panel selection, and opens up a new Macro Builder window on these panels. This window initially contains only those panels that were selected in the graphical history.

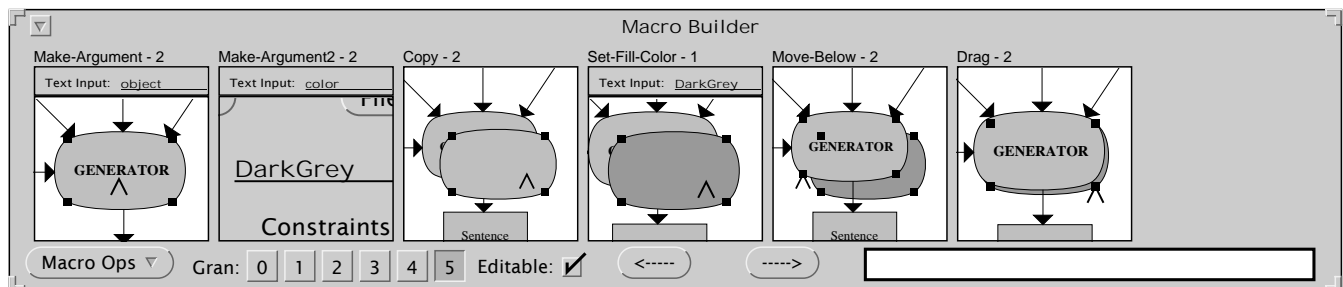


FIGURE 3. Macro Builder window containing operations to add a drop shadow to an object.

### Argument Declaration

As the next step, we declare the arguments to the macro. We do this by selecting the arguments where they appear in the panels, and providing them with names. To select a component of a panel, we first have to make the panels *editable*. This is done by checking the editable box at the bottom of the Macro Builder, which replaces the static graphical representations of the history panels with fully editable graphical canvases. Objects in these canvases can be selected and manipulated in the same manner as objects in a regular scene. The first argument to this macro will be the object for which the shadow is generated. We examine the panels in the macro builder window for an instance of the original oval. This oval appears in each of the panels, so we select any one of these instances, give it the name “object” by typing this name in the Text Input widget of the control panel, and execute the Make-Argument command.

A panel is added to the beginning of the history, depicting the argument selection. Argument declarations are placed at the beginning of our macros, just as they appear at the beginning of traditional procedures. The resulting panel is the first of the sequence of panels depicted in the Macro Builder of Figure 3. The argument declaration panels show the arguments as they appear before the operations in the macro were invoked, plus additional scene context. They are not just copies of the panels that were used for selecting the arguments. Scene objects that do not exist at the beginning of the macro, such as the oval produced by the copy operation in the third panel of Figure 3, are not plausible arguments, and Chimera will not allow them to be used for this purpose.

In addition to choosing graphical objects to be arguments to a macro, we can also choose graphical properties such as color or linestyle. To select a graphical property, we can select from the history a widget in which this property is displayed. Widgets can be selected just like any other graphical object. For this macro, we would like the color of the drop shadow to be a second argument. First we locate the panel in which we specified the color. This is the Text Input field of the fourth panel of Figure 3. We select this widget, and give the second argument the name “color”. A new argument declaration panel is created, which is the second panel of Figure 3.

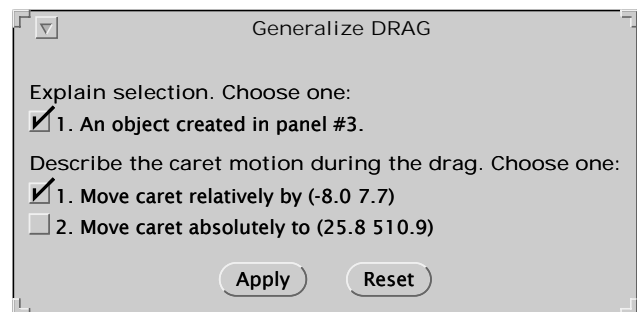
Recall that our system chooses rendering styles for panel objects according to the objects’ roles in the explanation. However in Figure 3, all objects are rendered in their usual fashion. We automatically revert to this standard rendering when panels are made editable, since these panels become fully editable scenes, and the user may want to query or manipulate the colors and other graphical properties of the objects in these panels. When the panels are restored to their

original uneditable state, the original rendering is also restored.

### Generalization

Next it is important to generalize the macro operations to work in new contexts. This generalization can be either specified by the user, or inferred by the system with the help of a built-in inference engine. For each editor command, Chimera has been supplied with a set of different interpretations, as well as heuristics for distinguishing when each interpretation is likely. When choosing a default generalization of a command, the system evaluates the heuristics in the context of the graphics state to produce an ordered list of possible intents. The user can view the system’s generalizations and override them if necessary. Once again the graphical history representation is useful as a means of selecting panels, this time for choosing panels to be generalized.

For example, after selecting the last panel of Figure 3, we execute the Generalize-Panel command. The window shown in Figure 4 appears, containing the various generalizations that the system considered plausible in the given context, with the most likely interpretation selected. The generalizations of all the operations contained in a panel can



**FIGURE 4.** A form showing the system’s generalizations for the last panel of Figure 3.

be viewed and modified at once. This panel contains the selection of the drop shadow, and the subsequent translation of the shadow to lie at the appropriate offset under the original object. Only one of the built-in interpretations for the selection is valid in the context of the last panel: that the object selected at this step is the object created in the third panel.

The Generalize-Panel command need not be executed explicitly for every panel in the macro. Another command can be used to set or reset all panels to their default generalization. When the macro is executed, all panels that have never been generalized are automatically given a default generalization.

### Generalizing a selection

The system has a number of possible interpretations of object selections. As an example of the types of generalizations Chimera is capable of performing, we list the various classes of selection generalizations here. An object may be selected because of the following classes of reasons:

- **Argument.** The object is an argument to the macro.
- **Constant.** The object is a constant in the macro.
- **Component.** The object is a particular component of another object, or a parent of another object. Example: first vertex of a polyline.
- **Temporal Reference.** The object was referenced in a particular macro step. Example: object created in panel #3.
- **Position.** The object shares a particular geometric relationship with another object. Example: leftmost segment of a box.

Selection criteria can also be combined in two ways:

- **Disjunction.** Multiple objects selected for different reasons. Example: an object is selected because it is either argument 1 or argument 2.
- **Composition.** The composition of multiple selection criteria. Example: first vertex of the second segment of argument 1.

This set of selection criteria is by no means complete. For example, a set of objects may have been selected because they share a particular graphical property in common (e. g., the same fill color), and Chimera cannot detect such an intent. Even within the categories above, there are many other selection criteria that we would like the system to consider. For example, it will not propose that an object was selected because it overlaps another interesting object.

### Generalizing a move

The second checklist of Figure 4 explains the system's generalization of the move or drag operation. There are two possible explanations that fit the bill: a relative translation and an absolute move. In this case, the system chooses the relative translation as most likely. If the dragged objects were moved so that the caret, the software cursor, snapped to an object or an intersection point (of either scene objects or alignment lines), then this would be considered the most likely interpretation. This allows us to define macros that perform geometric constructions, using the snap-dragging interaction technique developed by Bier [2]. For example, we can use this technique to define macros in Chimera that bisect angles, construct the midpoint of lines, and align shapes.

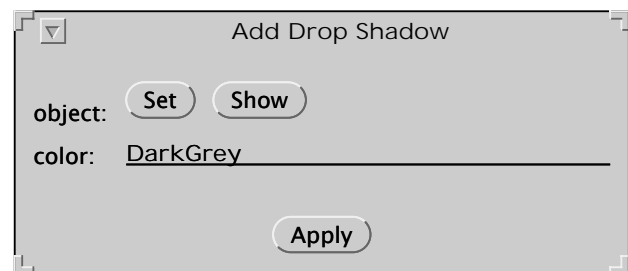
### Representing generalizations textually

Our macro facility represents generalizations as textual supplements to the graphical display of commands. Another approach might involve adding graphical symbols to the panels in order to make the system's interpretations of the commands clear. This approach has several problems. If the number of generalizations known by the system is large, then the graphical vocabulary must also be large. Unless the same graphical conventions are used by the system during normal editing, the user would need to learn a new visual language in order to define macros. By representing generalizations textually, in English, our generalizations are accessible to all users of the system. Our approach is similar to that of SmallStar, in which generalizations are displayed as textual data descriptions [4].

### Macro archiving and invocation

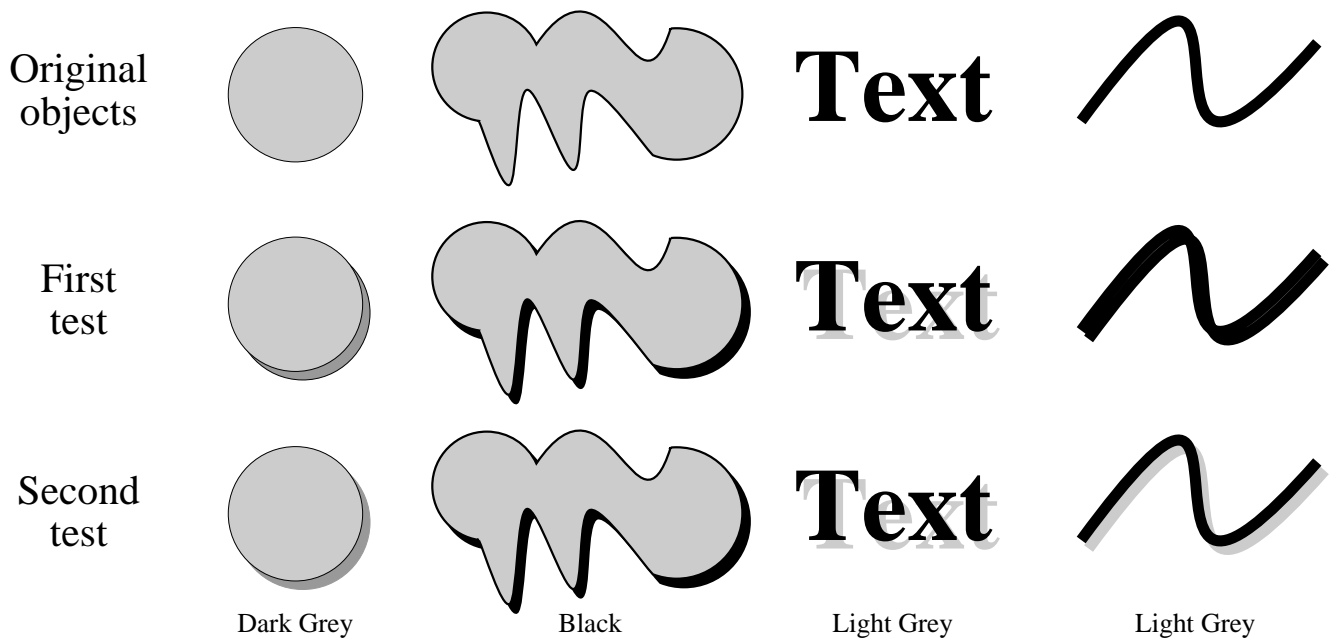
After a macro has been generalized, it can be named, saved, and invoked. Currently we save macros with all of the scene state that was present at definition time. This allows an editable graphical history representation of the macro to be recreated for subsequent editing that is identical to the panels originally displayed in the Macro Builder window during macro construction time.

To invoke a macro, the user executes a menu command and a macro invocation window pops up on the screen. For the drop shadow macro that we have just defined, this window is shown in Figure 5. The window contains an entry for each of the arguments declared previously. The first argument, "object" is assigned two buttons: one to set the argument and the other to show it. The second argument, "color", is a property argument, and Chimera uses a different technique to set and show property arguments. For each property argument, a copy of the widget used to specify the argument during the original macro demonstration is included in the



**FIGURE 5.** Window for setting arguments and invoking the drop shadow macro.

invocation window. Since the Text Input field of the control panel was originally used to specify the color of the drop shadow, this widget is copied and added to the invocation window. As a default, the widget contains the value specified for this parameter during macro demonstration time.



**FIGURE 6.** Testing the macro. The first row contains a set of test objects. The next row contains the results of invoking the original macro on these objects, using the colors named at the bottoms of the columns. The final row shows the results of invoking the debugged version of the macro.

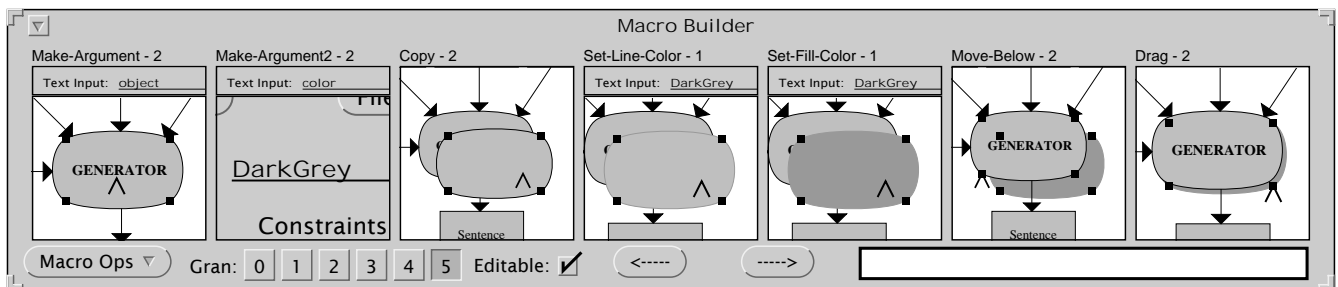
Effectively this parameter is treated as an optional parameter with a default. If the user does not change its value in the macro invocation window, the macro will use the value that was used during demonstration.

### Testing and debugging

An important part of programming, demonstrational and conventional alike, is the testing and debugging phase. In the top row of Figure 6 we have created four different shapes, in the second row we apply the drop shadow macro to each, using the colors listed at the bottoms of the columns. To create the drop shadow for the circle, we use the dark grey default color already in the macro invocation window. Next, we change the shadow color to black, and apply the macro to the shape composed of splines and arcs. Then, with the shadow color set to light grey, we apply the macro to the text and finally the Bezier curve.

At this point we notice a bug in the macro. Though we expect the macro to add a light grey drop shadow to the Bezier, the drop shadow is black. To debug the macro, we go back to the original Macro Builder window in Figure 3 and examine the commands that it contains. The bug quickly becomes apparent. Though we changed the fill color of the drop shadow, we never changed the line color. On inspecting the results of this initial test again, it is clear that the circle's drop shadow is incorrect as well since it too has a black line, yet we did not notice a problem at first because the shadow is dark.

The graphical history representation supports editing operations on either macros or the history directly, in place. When the panels are made editable, new commands can be executed directly on the panel objects. These additional commands can be propagated into the history at the point at which they are inserted, by executing the Propagate-Panel-



**FIGURE 7.** Final version of the Macro Builder window containing operations to add a drop shadow to an object.

Changes command. When this command is executed, the system transparently undoes all of the operations after the newly inserted operations, executes these new operations, and redoes all of the operations that had been undone. If commands are added to the history, rather than a macro, the editor scene corresponding to this history is updated according to the changes. In all cases, the subsequent panels of the history are regenerated to take into account the changes that had been inserted earlier.

To fix the bug in the macro shown in Figure 3, we need to add a Set-Line-Color operation. To do this, we type Dark-Grey in the Text Input widget of Chimera's control panel, and execute the Set-Line-Color command in the third panel of the Macro Builder window where the copy is already selected. Next we select this panel, and execute the command that propagates the newly inserted command into the history just after the copy command. The resulting macro is shown in Figure 7. An additional panel was created (panel 4) to represent the newly added operation, and subsequent panels all show the copy with its new line color.

After adding the new Set-Line-Color panel to the macro, we generalize this panel, and execute the macro on our test cases once again. The results, shown in the last row of Figure 6, are now as we expected.

Panels can also be deleted from histories or macros. The user can select a sequence of panels, and execute a command that removes these commands as well as any effect that they had. As with command insertion, Chimera must reformulate the panels appearing after the change, taking into account the modified scene.

## CONCLUSIONS

We have developed a graphical history representation that supports macro construction in a variety of different ways. The graphical history representation allows people defining new macros by example to review the commands that they have performed. Others who were not present during macro definition time can examine the contents of a macro. The commands are displayed graphically, in the same visual language as the interface itself, thus people who have used the system for ordinary editing can understand the macro representation.

The history representation provides a means of selecting operations. This is useful for two different steps of macro creation. At any time, the user can scroll through the history, and select out useful commands for a new macro. Accordingly, Chimera needs no additional commands to start and stop macro recording. Later, a user may want to view or change the generalizations associated with a set of panels.

Again, the graphical representation can be used to select the appropriate panels.

The macro representation makes it very easy to select arguments. After selecting the checkbox that makes the panels of a macro editable, we can select objects directly in the panels and turn them into arguments. Graphical properties can also be turned into arguments by selecting the widgets that set these properties from the macro panels.

Macros are not always defined correctly the first time, and the histories present an interface for editing commands. New commands can be inserted by invoking additional commands in the editable panels, and executing a command that propagates the changes. Unwanted commands can be removed by selecting panels and deleting them.

The macro system itself often refers to the representation when communicating important information to the user. During the generalization process, the interpretation of a command may refer back to steps made at an earlier point in time. For example, the system often needs to refer to an object that was created in a particular panel, or a measurement that was made at a certain step. Macros can generate run-time errors if they are invoked on objects of the incorrect type. Chimera also uses the macro representation to indicate which panel of the macro generated an error.

In summary, graphical histories facilitate macro definition in five different capacities:

1. Reviewing macro contents
2. Selecting operations
  - to encapsulate in a macro
  - to set and view generalizations
3. Selecting arguments
4. Editing macro contents
  - inserting operations
  - deleting operations
5. Referencing operations
  - during generalization
  - in error messages

## FUTURE WORK

The graphical histories representation can support macro definition and testing in a number of other ways. We could use this representation to show, step by step, the effects of applying a macro to new arguments. By placing together multiple macro viewers vertically, aligned so that similar panels are registered together, we could easily compare the effects of applying a macro to different argument sets, and quickly find the panel in which one of the macros generates unexpected results.

We could also use the panels to specify additional command generalizations when the existing ones fail. For example, one step in a macro might involve reducing a box's width by half. If the system is incapable of inferring the desired intent of this operation, the user might be able to add annotations to the panels that make the intent explicit. Since the panels are editable, the user might be able to use direct manipulation techniques to define a *temporal constraint* between the width of the box at two different points in time (or panels). The interface for doing this might be similar to Chimera's interface for defining constraints between separate objects.

There are a number of basic ways in which our macro by example facility can be enhanced. Currently when we save macros, we save all of the scene state, which allows us to restore the original graphical representation of macros for subsequent editing. This increases the storage requirements of the macro. We could also provide an option that automatically strips superfluous scene objects from the macro. Not only would this reduce storage, but it might also make the graphical macro representation clearer since it would contain no extraneous objects.

We would also like to expand the generalizations that Chimera is capable of making, so that a greater number of useful macros by example can be defined. Loops and conditionals would also increase the power of our macro facility. We have experimented with using graphical search [5] and constraint-based search [7] as iteration mechanisms for graphical macros, but others would be helpful as well.

Finally, it is important to provide a means of representing changes to the macros within the graphical representation, so that changes can be undone. Currently, Chimera can generate histories for macro panels that have been edited, but once the panel changes have been propagated into the macro, or a panel has been deleted, the information is lost.

## ACKNOWLEDGMENTS

Michael Elhadad made suggestions that led to an improved system. Members of the Programming by Example Workshop at Apple Computer provided insights helpful in revising this paper. This work was partially sponsored by a grant from IBM Watson Research Labs.

## REFERENCES

1. Asente, P. Editing Graphical Objects Using Procedural Representations. DEC WRL Research Report 87/6. November 1987. Revised version of Stanford Ph. D. thesis.
2. Bier, E. A., and Stone, M. C. Snap-Dragging. Proceedings of SIGGRAPH '86 (Dallas, Texas, August 18-22, 1986) In *Computer Graphics 20*, 4 (August 1986). 233-240.
3. Cypher, A. EAGER: Programming Repetitive Tasks By Example. In *CHI '91 Conference Proceedings* (New Orleans, LA, April 27-May 2, 1991). 33-39.
4. Halbert, D. C. *Programming by Example*. Xerox Office Systems Division Technical Report, OSD-T8402. December 1984.
5. Kurlander, D., and Bier, E. A. Graphical Search and Replace. Proceedings of SIGGRAPH '88 (Atlanta, Georgia, August 1-5, 1988). In *Computer Graphics 22*, 4 (August 1988). 113-120.
6. Kurlander, D., and Feiner S. A Visual Language for Browsing, Undoing, and Redoing Graphical Interface Commands. *Visual Languages and Visual Programming*. S.K. Chang (ed.). Plenum Press, New York, NY. pp. 257-275, 1990.
7. Kurlander, D., and Feiner, S. Interactive Constraint-Based Search and Replace. In *CHI '92 Conference Proceedings* (Monterey, CA, May 3-7, 1992). 609-618.
8. Kurlander, D. *Graphical Editing by Example*. Ph.D. Thesis. Columbia University. Computer Science. In preparation. 1992.
9. Lieberman, H. An Example Based Environment for Beginning Programmers. In *Instructional Science 14*, (1986). 277-292.
10. Maulsby, D. L., Witten, I. H., and Kittlitz, K. A. Meta-mouse: Specifying Graphical Procedures by Example. Proceedings of SIGGRAPH '89 (Boston, MA, July 31-August 4, 1989) In *Computer Graphics 23*, 4 (July 1989). 127-136.
11. Myers, B. A. *Creating User Interfaces by Demonstration*. Academic Press, Boston, 1988.
12. Myers, B. A. Demonstrational Interfaces: A Step Beyond Direct Manipulation. Technical Report CMU-CS-90-162. Carnegie Mellon University, School of Computer Science. August 1990.
13. Olsen, D. R. Jr., and Dance, J. R. Macros by Example in a Graphical UIMS. *Computer Graphics and Applications 8*, 1 (January 1988). 68-78.
14. Stallman, R. GNU Emacs Manual, Sixth Edition, Version 18. Free Software Foundation, Cambridge, MA. March 1987.