Scripting Graphical Applications by Demonstration

Brad A. Myers
Human Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213
bam@cs.cmu.edu http://www.cs.cmu.edu/~bam

ABSTRACT

Writing scripts (often called "macros") can be helpful for automating repetitive tasks. Scripting facilities for text editors like Emacs and Microsoft Word have been widely used and available. However, for graphical applications, scripting has been tried many times but has never been successful. This is mainly due to the data description problem of determining how to generalize the particular objects selected at demonstration time. Previous systems have mostly tried to solve this using inferencing, but this has a number of problems, including guessing wrong and providing appropriate feedback and control to users. Therefore, the Topaz framework does not use inferencing and instead allows the user to specify how the appropriate objects should be found. This is achieved by recording changes to which objects are selected and searches for objects, so that scripts can be written with respect to the selected object, in the same way as Emacs keyboard macros. Furthermore, all values can be explicitly generalized in a number of ways, and scripts can be invoked as a result of other commands. By leveraging off of Amulet's command object architecture, programmers get these capabilities for free in their applications. The result is that much more sophisticated scripting capabilities available in applications with no extra work for programmers.

Keywords: Scripting, Macros, Programming by Demonstration (PBD), Command Objects, Toolkits, User Interface Development Environments, Amulet.

INTRODUCTION

Creating scripts (also called "macros") for textual applications like text editors and spreadsheets has a long and very successful history. These scripts are important for automating repetitive tasks that are so common in direct manipulation interfaces. They can also be useful for creating new commands and for customizing generic applications to be more effective for specialized tasks (such as making a drawing program like MacDraw more efficient for creating charts).

To appear in Proceedings CHI'98:
Human Factors in Computing Systems.
Los Angeles, CA. April 18-23, 1998

To create a script, the user typically goes into record mode, then performs some commands, which operate normally in addition to being recorded. The recorded script can then be re-executed later in different contexts. For example, the keyboard macro facility of the Emacs text editor [13] is easy to use and extremely useful for manipulating the text. Many Emacs users claim that such facilities are essential to use the editor efficiently, especially when making repetitive changes. Scripting facility also exist in Microsoft Word using Visual Basic, and spreadsheets have a long history of creating scripts by example.

However, for graphical applications, such scripting facilities have mostly been unsuccessful. Graphical applications here refers to programs such as drawing editors, CAD programs, graphical programming languages, and iconic manipulation programs such as "visual shells" which are graphical interfaces to the file system (like the Macintosh Finder or the Windows Desktop). We have analyzed the fundamental features that allow text editors to be successfully scripted by demonstration and then incorporated these features into a graphical editor framework called "Topaz." Topaz stands for transcripts of programs activated with zeal. The result is that powerful and sophisticated scripts can be created by example in any graphical program. This is in contrast to special-purpose scripting languages like Chimera [3] that only work for a drawing editor.

With Topaz, the user is able to:

- change which object is selected by moving forwards and backwards through the objects in a variety of ways, recording the change of selection in a script,
- search for objects by matching on various properties or by location, and cause the found object to become selected (this is a generalization of *graphical search and replace* [4]),
- execute subsequent commands with respect to the currently selected objects, so that the next time the script runs, it will operate on newly selected objects,
- generalize the parameters of the operations (colors, locations, numbers, strings, etc.) in a variety of ways, so the values can be computed at run-time,
- execute the script a specified number of times or continuously until an error occurs (such as a search failing),
- specify that the script should be automatically invoked before or after other commands are executed.

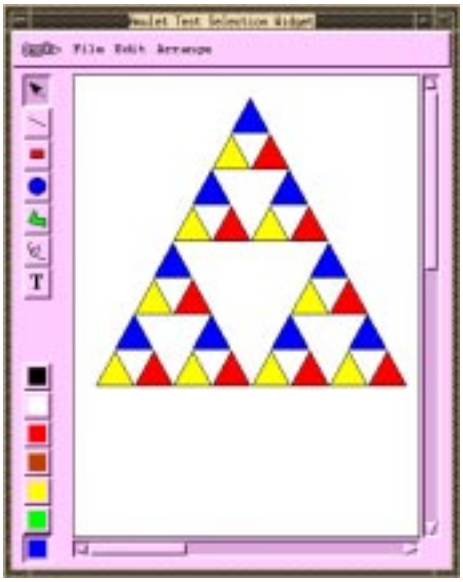

Figure 1. A drawing program created using Topaz showing the result of a script which subdivides a triangle into 3 smaller triangles, applied 13 times. This is called a "Sierpinski Gasket." Figure 6 shows the code of the script.

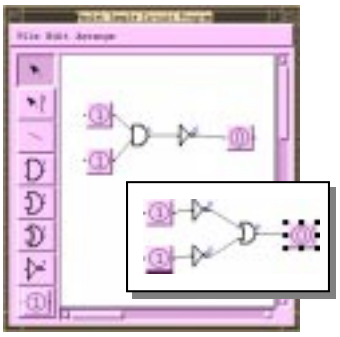

Figure 2. A sample circuit design program created using Topaz showing a circuit. The inset picture is the result of a script that converts an And gate and a Not gate into two Not gates and an Or gate and reconnects the wires.

The result is that users can create scripts *by demonstration* that perform such actions as replacing objects with patterns, performing repetitive edits, and creating graphical abbreviations. Specific examples of scripts created with Topaz are to:

- Build interesting patterns like the "Sierpinski Gasket" of Figure 1.
- Replace an And gate and a Not gate with the equivalent circuit according to DeMorgan's law, connecting all the wires appropriately, as shown in Figure 2.
- Put a drop-shadow beneath any type of selected object.
- Put an arch inside each rectangle as in Mondrian [6]. The user has full control over the whether the width of the side pieces and the height of the top is proportional to the size of the rectangle or constant.
- Insert a new node in a row of nodes and move all of the nodes that are on the right of the insert point further to the right to make room.
- Create a bar chart by making a row of rectangles whose heights depend on a given a list of numbers.
- Whenever a string is created, create a rectangle that is 10 pixels bigger than the string and center the rectangle behind the string.
- Perform the same edits to a set of graphics files, for example to replace all uses of an old logo picture with a new picture, and move all of the other objects a few pixels to the right and down to make room.
- And many more….

Of course, a particular application might have some of these commands built-in, but the goal here is to allow end users, who will not necessarily know how to program, to construct these kinds of scripts by demonstration when the application does *not* have the built-in command that they need.

We have implemented the Topaz framework using the Amulet toolkit [12], leveraging off of Amulet's command object architecture [11]. The result is that graphical applications get these sophisticated scripting facilities without any extra code in their applications. At this point, these facilities have been tested with a drawing program and a circuit editor, and we are working on a visual shell.

WHY ARE TEXT EDITORS EASIER TO SCRIPT THAN GRAPHICAL APPLICATIONS?

In a text editor like Emacs and Microsoft Word, most operations are performed with respect to the cursor, and there are a wide variety of ways to move the cursor using the keyboard and the mouse. These include moving the cursor forward and backward by characters, by words, by lines, or by sentences. The movement operations work in text documents because the content is an ordered sequence of characters, so forward and backwards are meaningful. Furthermore, for text with any kind of structure (including program code, content lists, etc.) moving forward by a line or by a certain number of words will correspond to moving by a *semantically* meaningful unit.

Scripts for repetitive actions take advantage of this ordering. A very common idiom used in Emacs keyboard macros is to move or search to the next occurrence of a pattern, and then perform operations on it. This script can then be repeated until all occurrences are processed. It is interesting to note that many powerful scripts can be written in this way without using conditionals or iterations, other than repeating the entire script until a search or move fails.

There are no equivalent operations in graphical programs. Just recording the low-level input events, which works well for keyboard events, does not work for mouse events since the specific location of the mouse is recorded when the buttons are clicked, and when the script is replayed, often

the wrong object is at that location. Indeed Microsoft Word turns off the use of the mouse to move the cursor while recording a script. Many attempts to provide scripting in graphical programs therefore try to infer the *meaning* of the mouse locations (that is, what is at this mouse location, and why was that object clicked on?). Other programs try to match the objects that were used in different executions of the script to try to create generalizations. These programs must use heuristics which means that the system can guess incorrectly, and often will not infer the correct program.

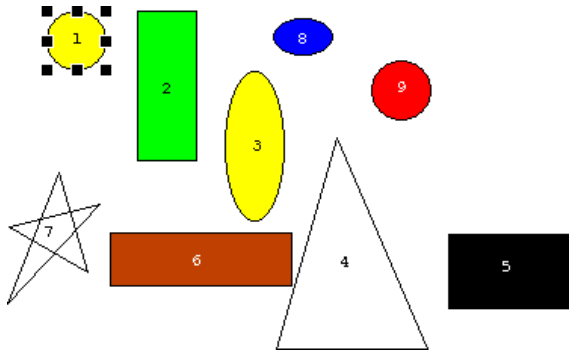

Figure 3. Circle 1 is selected. Moving the selection to the right will select rectangle 2, but then should oval 3 or 8 be selected? Currently, when moving to the right, Topaz selects objects in the order 1, 2, 3, 4, 5. But then moving the selection left from 5 selects 5, 4, 6, 7.

The approach taken in Topaz is to allow the user to specify how to find the correct objects using capabilities similar to those found in text editors, rather than trying to infer the generalizations. In graphics programs, the *selection* (often shown by black squares around objects as in Figure 3) corresponds to the cursor in text editors, and most commands operate on the selected set of objects. The innovation in Topaz is that users can change which objects are selected in graphical applications in a variety of ways, and have these recorded in a script.

Other innovations in Topaz are that there are various ways that the user can explicitly generalize a script so it will work in new contexts, and the ability to invoke a script automatically before or after other commands execute, as proposed in [2].

MOVING THE GRAPHICAL SELECTION

Text has a natural order, so commands like "forward" and "backward" are meaningful. However, in a graphical application, it is not so obvious what the order for all objects would be. The requirements for the ordering is that it have a well-defined first and last object, moving forward from the first object should go through every object exactly once, and moving backwards from the last object should select all the objects in the reverse order as forwards. It would also be good if the order made sense to users. However, in many cases, the particular order does not matter—it is often only important that each object be visited exactly once.

Our first idea was to go top-to-bottom, left-to-right, but this is not well-defined for some layouts of objects, and the backwards order is often not the reverse of the forwards order (see Figure 3).

Therefore, we decided to make the primary order for traversing objects be the display "Z" order from back to front. This has a number of advantages: it is well defined, reversible, and usually corresponds to the chronological order in which objects were created (older objects are further back unless the user has explicitly changed the order with a `To-Top` or `To-Bottom` command). A sophisticated user can also take advantage of this order in scripts. For example, to make sure that a script that creates arch inside of rectangles does not create arches inside the arches themselves, the script might start at the *end* of the list and move backwards. Since new rectangles are always created after the current end of the list, the selection will never get to the rectangles of the arch.

The current user interface for moving the selection is that the HOME key on the keyboard selects the first object, the END key selects the last object, and the left and right arrow keys select the next and previous objects in the Z order. If the shift key is held down, then the new object is added to the selection, in the same way as the shift key works with the arrow keys for text selection in Microsoft Word. All of the selection moving operations beep when a movement is not possible because there are no more objects on which to operate, and this stops the execution of the script.

There are some cases where it is important to find the next object in a graphical direction, so there are commands to select the next object to the left, right, up, down, inside or outside of the current object. These operations find the nearest object in that direction to the selected object that overlaps in the other dimension, so the reverse direction does not necessarily choose the same objects, as shown in Figure 3. These operations are available as buttons at the bottom of the search window (Figure 4).

SEARCHING FOR OBJECTS

Often it is important for scripts to only execute on specific objects. Previous demonstrational graphical systems have often tried to infer the properties from one or more examples, which is error-prone. Instead, Topaz allows the user to *search* for the appropriate object. This graphical search was introduced in 1988 [4] but still is not available even in sophisticated graphics editors like Adobe Illustrator, although it is provided by some CAD programs.

Topaz extends this searching to make it work in any graphical application, not just a drawing program. All objects in Amulet have a well-defined protocol for querying their properties. Topaz uses this to find what types of objects can be created and what parameters are available for each type. This does not require any new code in the application. Topaz automatically constructs a dialog box that allows the user to choose which properties should be used

for a search, as shown in Figure 4. Of course, this dialog box could be made to look much nicer if it was designed by hand, but we wanted to minimize the amount of custom code needed in each application, so we used a straightforward automatic mechanism to generate the search dialog. This is why it uses internal names (like `FILL_STYLE`) and a naïve layout.

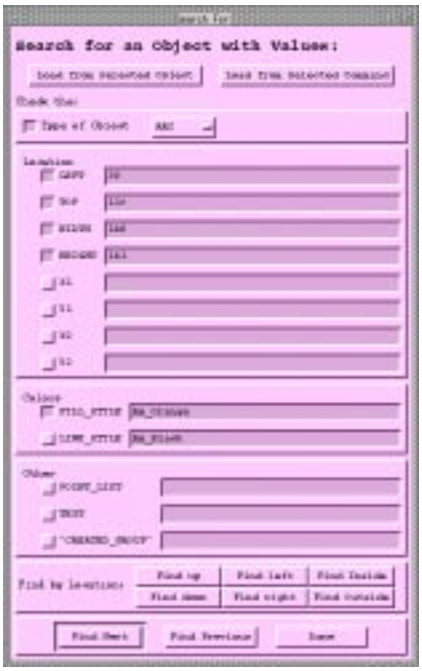

Figure 4. The search (find) dialog box automatically created by Topaz for the graphical editor of Figure 1. The values from an arc (oval) have been loaded.

To perform a search, the user can type in a value, or load the values from an existing object. There are eight ways to search for the next object, using the buttons at the bottom of the window. `Find Next` and `Previous` search in the "Z" order, and the others find in graphical layout order. All searches start from the current selection, or if nothing is selected, then from the first item in that order. For example, when searching to the `Right`, the first item is the left-most. "`Find Inside`" finds the back-most (in Z order) object on top of the selected object that is entirely inside it. This is useful for finding enclosed objects, like the label inside a box. "`Find Outside`" finds the front-most object behind the selection and completely surrounding it, and is useful for going from a label to its box. If nothing is selected, `Find Inside` starts from the back-most object, and `Find Outside` starts from the front-most.

When searching, Topaz only matches on the properties that have their checkboxes selected, so if nothing is checked, the search is through all objects. Selecting various properties supports more complex searches, such as "find the wire which is down from the selected object" which was needed for the script for Figure 2.

Figure 5. The undo history dialog box, in which previous commands can be selected for undo, repeat or scripting. Recording of selections and scrolling can also be turned on and off. Here, the commands for the script to create the triangles of Figure 1 are selected.

Figure 6. The commands of Figure 5 displayed in the scripting window, with some of the parameters already generalized into placeholders.

USER INTERFACE FOR SCRIPTING

Unlike most other demonstrational scripting facilities where the user has to think ahead that the next operations should be in a script, Topaz allows scripts to be created by selecting the desired commands from the list of commands displayed in the undo dialog box (see Figure 5). This list shows all the previous commands that have been executed. As reported earlier [11], the user can select any command in this list for selective undo or selective repeat.

The new feature added by Topaz is the ability to select a sequence of commands to be included in a script. (Selecting previous commands also was available for the graphical histories in Chimera [3].) Topaz allows non-contiguous sequences of commands to be selected (using the standard shift- and control- clicking), so that unlike other systems, the sequence of commands for the script do not have to be executed flawlessly without errors.

Once selected, the commands are expanded to show all their parameters, and are displayed in the scripting window (Figure 6). Now, the user can edit the script in various ways (discussed below). How the script will be invoked can be specified (the script of Figure 6 will be executed when the user hits the "F8" keyboard key). When the script is ready, it can be executed, saved, or removed (deleted).

EDITING AND DEBUGGING SCRIPTS

Many previous programming-by-demonstration systems seem to have assumed that all scripts would work the first time and never need to be changed, since there was no way to edit the recorded script (a notable exception was Chimera [3], which had nice editing facilities). In fact, some systems do not even have a reasonable representation of the recorded script that can be viewed. Topaz provides full editing of the script, including selecting the commands and deleting, cutting, copying, and pasting them. Also, commands in the undo dialog box (Figure 5) can be selected and inserted before or after any statement in the script to add new commands.

In order to debug the script, the user can select specific commands and execute just those commands, or single step through the script one command at a time. The results of the commands will be visible in the main application window, so the user can check if they are operating correctly. Because all commands can be undone, including the execution of the script itself, it is easy to back out of any operations that are incorrect.

An important capability not available in other systems that is this editing makes possible, is the ability to *demonstrate* new commands to be added to the middle of a script. The user can single step the script to the appropriate point, then execute the desired new commands, which will appear in the undo dialog box. These commands can be selected and inserted into the script at the current point. Then the user can continue single stepping the script to make sure that the subsequent script commands still work correctly.

GENERALIZATION OF PARAMETERS

An important feature of Topaz is the ability to *generalize* the parameters of operations. This is important to have a different object, position, or value used when the script is run, rather than the specific constant object used when the script was demonstrated. The user can select any value displayed in the script window and double-click (or use the "generalize" menu item in the Edit menu) to bring up a dialog box. After generalizing, the value displayed in the script is replaced with a descriptive placeholder. Bringing up the dialog box for a placeholder will allow it to be edited, or changed back into a constant. This is related to the "data description" property sheets in the SmallStar visual shell [1], but here they are domain independent. If an application has a special type of value, the programmer can add a new dialog box to handle it, but the built-in dialog boxes seem sufficient for many applications. Also, the dialog boxes have a "custom" option that will in the future link to a programming subsystem where any expression for computing the values can be entered in a language like JavaScript or Visual Basic. However, we have found the built-in options to be sufficient for most scripts.

Figure 7. The dialog box for generalizing objects, with the script window in the background.

There are three built-in dialog boxes: one for generalizing objects, one for positions, and one for all other values. The dialog box for generalizing objects (see Figure 7) lets the user pick how to get the object when the script runs:

- Use a constant object (which defaults to the original object, but the user can type a new value).
- Use whatever objects are selected at that point in the script. Often the user will arrange for the appropriate object to be selected before executing an operation.
- Pause the script and ask the user to select some objects.
- Use the object or objects that are returned or operated on by a previous operation in the script. For example, a resize operation might be generalized to operate on the object returned by a previous create command. The user specifies which previous command to use by selecting the command in Figure 6 and clicking the "`Load`" button in Figure 7. There are two options for this, because many commands return a variable number of objects (e.g., paste, duplicate or select-all) and there must be a way to specify that subsequent commands operate on *all* of the objects, no matter how many, or on a particular object (e.g., the first one) from the set.

Figure 7 shows that the clicked on object is used in three places in this script. The one the user actually clicked on is shown in green, and all other uses are shown in yellow.

The top of the generalizing dialog box allows all of these uses to be replaced at the same time, or just the specific one that the user clicked on.

Topaz performs one generalization automatically. When a set of commands is brought into the script window, Topaz first checks to see if any of the commands are the kind that create a new set of objects. This includes create commands (including Paste and Duplicate) as well as selection commands that define a set of objects. Topaz searches for any subsequent uses of these objects, and if found, replaces the occurrences with placeholders that refer to the results of the create commands. This replacement is made automatically because in almost every script, if an object is created and then manipulated, when the script is run, the manipulation should operate on the newly created object and not on the original constant object. For example, all of the object generalizations for the script of Figure 6 were automatically performed by this mechanism. If the user really wants to use the original, constant object, the automatically generated placeholder can be edited using the standard dialog box (Figure 7) back to the constant value, but this has never been necessary so far.

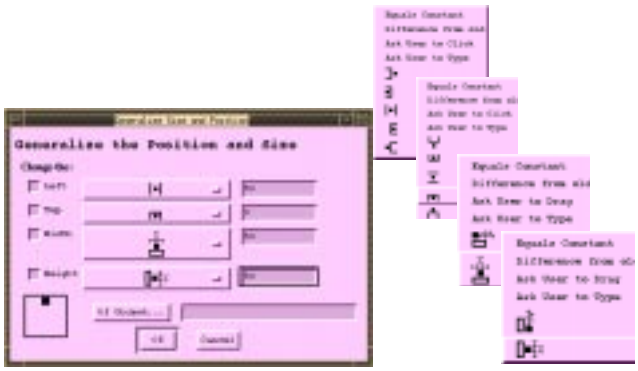

Figure 8. The dialog box for generalizing locations, along with the pop-up menus for setting the left, top, width and height. The picture in the lower left of the dialog box shows the result of the Left and Top selections.

Figure 8 shows the dialog box for generalizing locations. The user can click on the check boxes at the left of the dialog box to choose which parameters of the location to generalize. The options for the left coordinate are to be constant, a difference from the old value, a value specified by the user at run time either by clicking in the picture or typing, or a value computed with respect to a different object either by being to the right outside, right inside, center, left inside or left outside. If the center is selected, then the text input field is for the percent of the way across (the default is 50% which is in the center, but the user could type another value such as 33% to make the object be 1/3 of the way across). For all other relationships, the text box is the offset from the other object. The top properties are similar. For the width and height, the bottom two icons are to make it depend on another object by offset (e.g., my width is the other object's width + 10) or percent (my width is 50% of the other's width). When an option is chosen that depends on another object, the user can select which object to use. The normal choice is whatever is selected when this command runs.

All other types of values use a dialog box which allows the value to be constant, the current value from the palette if there is a palette registered for this type of value (like the color palette), a value chosen from a list, a value that the user types, or the value computed by a previous command.

One use for the list of values is to create a script that will process a number of files. The file `Open` and `Save-As` commands have the filename as their parameter, so the user can record a script with open and save in them, and then generalize the filename parameter to either be a list of filenames or to ask the user for the filenames.

Generalizing the values integrates well with the Search dialog box, since the search command uses as its parameters the values that were searched for. The user can therefore generalize parameters for a search in the same way as any other command, for example to search for the next object whose color is the same as the selected object, rather than some constant color.

INVOCATION OF SCRIPTS

Topaz supports a variety of ways to invoke a script. First, the `Execute` button on the script window (bottom of Figure 6) can be used. Second, Topaz supports the conventional ways to invoke scripts, using a keyboard accelerator or by putting the script into a menu of the application. In the future, scripts will be able to be executed at a certain time (for example to cleanup a disk every night).

Figure 9. The dialog box for specifying how to invoke scripts. The `Text_Create` command description was automatically entered when the user clicked on the `Load` button.

Topaz also supports a novel way to invoke scripts: before or after any other command is executed. This idea was proposed earlier [2], but it has never previously been implemented in any system. The idea is that the user not only can demonstrate the script to be executed, but also which commands the script should run after or before. For example, after demonstrating a script to surround an object with

a rectangle, the user brings up the invocation dialog box of Figure 9. Next, the user selects the "After" option, then demonstrates a `Text_Create` operation and selects it in the main undo dialog box (Figure 5) (or else the user can just select any previously executed `Text_Create` operation). Finally, the user hits the "`Load`" button in Figure 9 which loads the command description into the field. By default, all the parameters to the command are generalized so the script will be invoked whenever any `Text_Create` command is executed. This will put a rectangle behind any strings created. Alternatively, the user can select any parameter and use the original value of the example or any of the other generalizations discussed in the previous sections.

This provides the ability to create many intriguing scripts. For example, graphical abbreviations can be defined such as "whenever the text 'logo' is created, delete it and put in the logo picture read from a file." Since scrolling operations can be recorded, a script to scroll down after a search could be created.

To further the analogy with Emacs, scripts can be executed a specified number of times (using the count field at the bottom of Figure 6). The user can type a value or, like in Emacs, ^U will multiply the current count value by 4. For many scripts, the user will pick some really large number of times to execute, because, also like Emacs, all scripts are stopped if anything causes a beep (the low-level beep function was modified to set a global flag if a beep happens, and Topaz clears the flag before executing any script and checks the flag before each operation). Error dialogs always beep, and searches and movements of the selection causes a beep if they fail, and nothing beeps when everything is OK, so this is a good heuristic for stopping script execution.

Scripts are represented as a command like any other command, and are listed in the undo dialog box, so the execution of a script can be undone, repeated, or even included in other scripts. The count of the number of times to execute is the parameter for a script (which can be generalized), so a recursive script could even be written that would stop when the count got to zero or when there was a beep.

IMPLEMENTATION

An important feature of Topaz is that it requires almost no work from the application developer to use scripting, if the application is created using the Amulet framework. Amulet requires that all operations be encoded into the methods and data values of a *command object* [11]. Due to the way that these command objects are designed, they already provide Topaz with most of the information it needs. Command objects have a "Do" method that performs the operation, and an "Undo" method for undoing. They also contain methods to support *selective-repeat* and *selective-undo* which are when the user selects the command in the list of Figure 5 and asks for it to be repeated or undone in the current context. The scripting mechanism uses this selective-repeat to execute each command of the list. Selective-undo is used when the script itself is undone—it just undoes each of the component commands in the script. The selective-repeat has a companion method that tests whether the command can be executed in the current context, and this is used by the script before each command is executed to verify that it can be executed, and if not, an appropriate error message is provided. This makes it safe for Topaz to allow the user to insert or delete arbitrary commands in the script, since at run-time there will be a check to make sure that each command can execute. Command objects also adhere to a standard protocol for describing their parameters and generated values, so Topaz can inspect, display and generalize the values of any command.

Using the selective repeat mechanism, instead of just invoking the original Do method again, has a number of advantages. The Do method does not take any parameters, since the values to be used come from the selected object, the palette, and pop-up dialog boxes that ask the user. This is a very annoying feature of the scripts in Microsoft Word—they keep popping up the dialog boxes when the script is run. Pursuit [9] used "meta-dialog boxes" to allow the user to specify whether the dialog box should appear at run time, and if so, what parts should be filled in by the user, but this required that Pursuit parse and understand the dialog boxes. By using the selective repeat method, the parameters can be passed to the operation directly since usually Topaz computes the values. Of course, it would also be useful to let the user pop up the original dialog boxes when desirable, by using the original Do method of the command rather than the selective-repeat.

The command object architecture also helps Topaz record the scripts at the appropriate level. Scripts recorded at the mouse-movement-level fail because objects are not at the same place the next time. Applications written in Amulet must already encode the semantics of their operations into command objects, so recording at this level allows Topaz to create robust, repeatable and generalizable scripts without requiring Topaz to try to infer the "meaning" of the operations or of mouse events. Also, users can invoke commands in any way that is convenient (from the mouse, menus or keyboard accelerators) and these are recorded the same way in scripts, since they all use the same command object.

Amulet's built-in save and load mechanism allows applications to save and load their files with a minimum of code. All that is necessary is to register each of the main types that the user can create, and the important parameters of those types. Topaz takes advantage of this information to automatically construct the Search window, so again no extra work is required of the programmer.

Most graphical applications use Amulet's selection handles widget, and so the selection moving operations and searching come for free when this widget is used. Note that a Replace operation as in [4] could *not* be added without new code in the application, because Topaz would not know how to create new objects. In the current design, Topaz can

inspect all of the graphical objects, but it can only *change* the objects by re-executing commands that have already been demonstrated by the user.

All extra code in the application is optional. The programmer should register the palettes in the application, so that the value generalization can tell if the selected value can be computed from a palette. Also, the main menubars should be registered with Topaz so it will know how to add scripts to the menus if requested by the user (Figure 9).

STATUS AND FUTURE WORK

Topaz is mostly working, and has been integrated with a drawing program and a small circuit design program. The main hole is the lack of a general-purpose programming language for the "custom" generalization options. We would also like to investigate writing scripts that work across multiple applications. As we get more experience with users and test Topaz with a wider variety of applications, we will continuously refine the interface.

RELATED WORK

The main influence on Topaz is Emacs [13], and we have tried to provide the key features of Emacs to graphical editors, which has not been done previously. Vmacs [5] was an early attempt to apply Emacs ideas to graphics, but it did not support any scripting by example, and instead concentrated on parsing of freehand drawings. The idea of dialog boxes to generalize parameters in scripts was used in the SmallStar [1] visual shell, and Topaz provides this capability for any graphical application. Graphical search was introduced in [4] for a graphics editor, and Topaz allows it to be used in scripts and for multiple types of applications. Topaz is also the first to allow the generalization of the parameters of the search. Chimera [3] supported graphical histories of operations in a graphics editor, and allowed commands to be selected for scripts, but supported only rudimentary generalizations.

There are many previous examples of scripting by demonstration in graphical applications, including Peridot [10], MetaMouse [7], Mondrian [6], Pursuit [9], Gamut [8], etc., but most of these concentrated on inferencing algorithms rather than providing sophisticated controls to users.

CONCLUSIONS

Topaz allows the user to construct sophisticated scripts by demonstrating the desired commands and then explicitly generalizing the parameters, usually without the need for conditionals and embedded iterations. This is achieved by generalizing the cursor movement capabilities from Emacs to a graphical domain. Topaz also allows scripts to be executed before or after other commands. These capabilities are provided to users with almost no effort on the part of the application developer. We hope that these capabilities will be generally useful, and that they will appear in many more kinds of applications.

ACKNOWLEDGMENTS

For help with this paper, I would like to thank Bernita Myers, Rob Miller, Rich McDaniel, and Bruce Kyle. This research was partially sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326, and partially by NSF under grant number IRI-9319969. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

1. Halbert, D.C. "SmallStar: Programming by Demonstration in the Desktop Metaphor," in *Watch What I Do: Programming by Demonstration.* 1993. Cambridge, MA: MIT Press. pp. 102-123.

2. Kosbie, D.S. and Myers, B.A., "PBD Invocation Techniques: A Review and Proposal," in *Watch What I Do: Programming by Demonstration*, A. Cypher, Editor 1993, MIT Press. Cambridge, MA. pp. 423-431.

3. Kurlander, D. "Chimera: Example-Based Graphical Editing," in *Watch What I Do: Programming by Demonstration.* 1993. Cambridge, MA: MIT Press. pp. 271-290.

4. Kurlander, D. and Bier, E.A. "Graphical Search and Replace," in *Proceedings SIGGRAPH'88: Computer Graphics.* 1988. Atlanta, GA: **22**. pp. 113-120.

5. Lakin, F., *et al.*, "The Electronic Design Notebook: Performing Medium And Processing Medium." *Visual Computer: International Journal of Computer Graphics*, 1989. **5**(4): pp. 214-226.

6. Lieberman, H. "Dominos and Storyboards: Beyond Icons on Strings," in *IEEE Computer Society: 1992 IEEE Workshop on Visual Languages.* 1992. Seattle, WA: pp. 65-71.

7. Maulsby, D.L. and Witten, I.H. "Inducing Procedures in a Direct-Manipulation Environment," in *Proceedings SIGCHI'89: Human Factors in Computing Systems.* 1989. Austin, TX: pp. 57-62.

8. McDaniel, R.G. and Myers, B.A. "Building Applications Using Only Demonstration," in *1998 International Conference On Intelligent User Interfaces.* 1998. San Francisco, CA: To appear.

9. Modugno, F., Corbett, A.T., and Myers, B.A., "Graphical Representation of Programs in a Demonstrational Visual Shell -- An Empirical Evaluation." *ACM Transactions on Computer-Human Interaction*, 1997. **4**(3): pp. 276-308.

10. Myers, B.A., "Creating User Interfaces Using Programming-by-Example, Visual Programming, and Constraints." *ACM Transactions on Programming Languages and Systems*, 1990. **12**(2): pp. 143-177.

11. Myers, B.A. and Kosbie, D. "Reusable Hierarchical Command Objects," in *Proceedings CHI'96: Human Factors in Computing Systems.* 1996. Vancouver, BC, Canada: pp. 260-267.

12. Myers, B.A., *et al.*, "The Amulet Environment: New Models for Effective User Interface Software Development." *IEEE Transactions on Software Engineering*, 1997. **23**(6): pp. 347-365.

13. Stallman, R.M., *Emacs: The Extensible, Customizable, Self-Documenting Display Editor*. MIT Artificial Intelligence Lab, 1979.