In the 1994 East-West International Conference on Human-Computer Interaction. St. Petersburg, Russia, August, 1994.

Exploring Graphical Feedback in a Demonstrational Visual Shell

Francesmary Modugno and Brad A. Myers

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 USA

Abstract. We present a visual language that serves as a novel form of feedback in a Programming by Demonstration (PBD) interface. The language explicitly represents data, such as files, with unique icons and implicitly represents operations by changes to data icons, so that operations reflect the changes seen in the interface when executed. In addition, the language is used to provide feedback to the user by indicating loops and inferred sets, specifying parameters, etc., and is integrated into other parts of the PBD system, such as the editor. This forms a close union between the interface, the PBD system and the program representation and helps bridge the gap between the user's mental model of the programming process and the actual programming task.

1 Introduction

In his classic 1983 article, Ben Shneiderman introduced the concept of a "direct manipulation" interface, in which objects on the screen can be pointed to and manipulated using the mouse and keyboard[25]. The Apple Macintosh, introduced in 1984, quickly popularized this interface style and today direct manipulation interfaces are widely used. Unfortunately, these interfaces have some well recognized limitations. For example, in textual interfaces such as the Unix shell, it is common for users to construct parameterized procedures (called "shell scripts") that automate repetitive tasks. This is often difficult to do in the Macintosh Finder or other "visual shells". Direct manipulation interfaces also do not provide convenient mechanisms for expressing abstraction or generalizations, such as "all `.tex` files edited on August 19, 1993." As a result, while direct manipulation interfaces are often easier to learn and use than their textual counterparts, users often find that complex, high-level, repetitive tasks are more difficult to perform in this domain.

Programming by Demonstration (PBD) systems[5] can potentially solve this problem. In a PBD system, users execute actions on concrete examples and the

system constructs a general program[23]. Such systems enable users to create general procedures without having special programming skills. They are easy to use because users operate by manipulating data the way they normally do in the interface. Unfortunately, they have limitations: they can infer incorrectly; most contain no static representation of the inferred program; feedback is often obscure or missing; and few provide editing facilities. This makes it difficult for users to know if the system has inferred correctly, to correct any errors, and to revise or change a program.

In order to address these problems, the Pursuit[19] demonstrational visual shell is exploring the use of a *graphical* representation of the program *while it is being written*. Programs are represented in a visual language in which data objects, such as files and folders, are represented as icons and operations are represented by the changes they cause to data icons. In addition to representing the program code, the graphical representations are used throughout the programming process in order to communicate with users. By addressing some of the limitations of PBD systems, this approach should make PBD techniques more effective in helping users manipulate and customize their systems.

2 Related Work

There have been several approaches to adding end-user programming to visual shells. Some visual shells contain a macro recorder (*e.g.*, SmallStar[9]; QuicKeys2, MacroMaker and Tempo II for the Macintosh; and HP NewWave) that makes a transcript of user actions that can be replayed later. Although effective in automating simple, repetitive tasks, macro recorders are limited because they record exactly what users do – only the object that is pointed to can be a parameter and the transcript consists of a straight-line sequence of commands. To generalize transcripts, some macro recorders produce a representation of the transcript in a textual programming language for users to edit. However, this requires users to understand a programming language that is significantly different from the desktop and does not take advantage of the visual aspects of the interface.

Some visual shells have invented a special graphical programming language (*e.g.*, [2, 8, 10, 12]) to enable users to write programs. Most of these languages are based on the data flow model, in which icons represent utilities and lines connecting them represent data paths. Unfortunately, most contain no way to depict abstractions or control structures. The types of programs users can write are quite limited. In addition, these languages require users to learn a special programming language whose syntax differs significantly from what they see in the interface. Finally, constructing programs by wiring together objects is quite different from the way users ordinarily interact with the system.

3 Motivation for the Pursuit Approach

Visual shells are easy to use because of the constantly visible, concrete, familiar representations of data objects and the illusion of concrete manipulation of these data objects[25]. Unfortunately, this "conceptual simplicity" is often lost when programming is introduced: users interact with the system visually, but usually program it off-line in a textual programming language. Users must develop two very different bodies of knowledge: one for interacting with the system and one for programming it. Pursuit attempts to bridge this gap. By allowing programs to be specified by demonstration and by representing programs in a visual programming language that reflects the desktop, users can apply knowledge of the

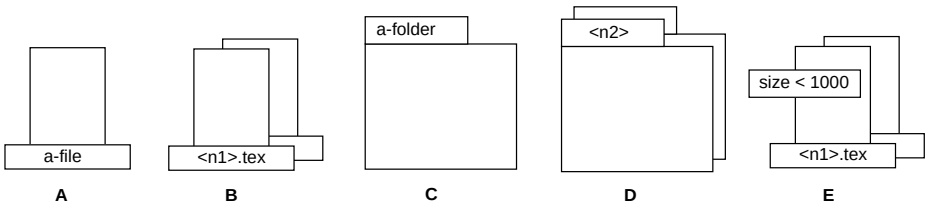

Fig. 1. The basic data types in the Pursuit visual language: (A) a file; (B) a set of files; (C) a folder; (D) a set of folders. Attributes attached to set icons constrain the objects in the set to have particular properties. They are also used to depict system inferences. For example, the first set of files (B) contains all those files whose name ends in `.tex`. The other set (E) contains all `.tex` files smaller than 1000 bytes.

interface and its objects to the visual language and its objects when constructing, viewing and editing a program.

3.1 Benefits of Representing the Program

There are several reasons to have a static representation of the program. First, it allows users to review the program's code and confirm that it does what they intended. It also gives them the opportunity to alter the code in order to correct errors, or to generalize the program further.

Furthermore, providing a representation of the program and its components *during* the demonstration gives users *feedback*. Feedback informs users of the system's inferences. Users can verify these inferences and suggest important features of the example to generalize.

Pursuit can detect loops over sets of data, branches on exit code conditions, and common substring patterns (see [19] for details). A good representation of the program *during* the demonstration informs users of the system's inferences as they are made. Furthermore, Pursuit's visual language not only represents the program code, but also indicates which operations to collapse into a loop, which data objects to execute the loop over, which data objects to use as parameters, etc.

There are other forms of feedback we could have chosen: dialog boxes[9]; questions and answers[17, 22]; textual representation of the code[14]; changing the appearance of actual interface objects (e.g. anticipation highlighting[4]); animation[7]; and sound[16]. Our approach has several benefits over these other forms. Unlike dialog boxes and the question-and-answer style, it is not always disruptive, since users can ignore the program representation until they are finished demonstrating it. Unlike programs represented in a textual language, it does not require that users learn a language that is very different from the interface. Finally, unlike anticipation highlighting, animation, and sound, there is a static representation for users to examine and edit.

4 The Representation Language

The Pursuit visual language[20] is based on the comic strip metaphor[13]. Familiar icons are used to represent data objects, such as files and folders. Sets of data objects are represented by overlaying two icons of the same type (Figure 1). For *abstract* sets of objects, graphical constructs called *attributes* are attached to set icons. Attributes constrain the properties of set objects and indicate the underlying PBD system's inferences. For example, the attribute in Figure 1 constrains the set to files smaller than 1000 bytes.

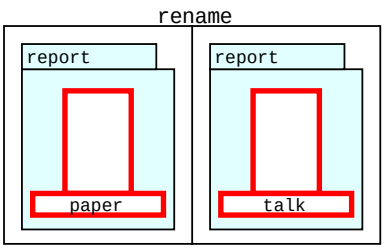

Fig. 2. The representation of the operation `rename paper talk`. The first panel shows the icon representing the file `paper` located in the `report` folder before it is renamed. The second panel shows the same file after the `rename` operation. The change in the icon's name represents the `rename` operation.

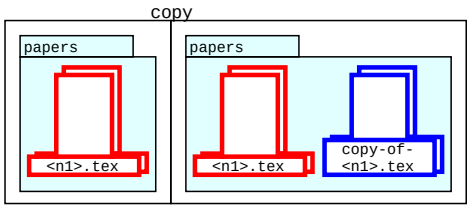

Fig. 3. The `copy` operation. The name on the file sets indicate the inferences the system made: all copied files have a name ending in `.tex`.

Each data icon contains a set of graphical properties, such as its location, icon appearance, name, etc. These graphical properties represent properties of the real data object in the interface. Operations are represented by changes in the graphical properties of data icons. Two panels are used to represent an operation. The *prologue* shows the data icons before the operation and the *epilogue* shows the icons after the operation (Figure 2). An operation's representation, therefore, resembles changes users see in the real interface when executing the operation[1].

A program is a series of operation panels concatenated together, along with representations of loops, conditionals, variables and parameters. However, using two panels per operations can produce space-inefficient scripts. To make programs more concise, Pursuit contains heuristics that combine knowledge of the domain with information about operations. The following examples illustrate these heuristics and other features of Pursuit.

4.1 A Simple Visual Program

This example illustrates how to write a program to backup all the `.tex` files in the `papers` folder. To backup the files, the user copies them to the `backups` folder and then compresses the copies. To automate this task, the user demonstrates its actions on a particular set of files. During the demonstration, the PBD system tries to create a program.

Figures 3-5 show the developing script during the demonstration. The first two panels (Figure 3) appear after the user opens the `papers` folder, selects

[1] Pursuit's visual representations are similar to those of Chimera[13] and Mondrian[15]. However, there are many differences. The most notable is that Pursuit contains explicit representations of variables, loops and conditionals and that Pursuit's layout is two-dimensional. See [19] for further comparisons.

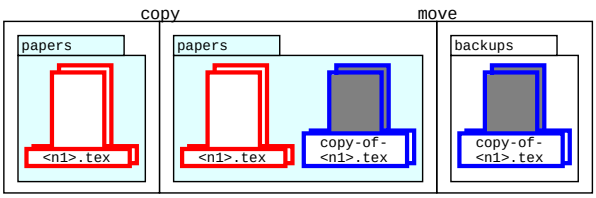

Fig. 4. After the user drags (moves) the copies to the `backups` folder, the third panel appears. Notice that in the script the set of copies icon has moved from the `papers` folder to the `backups` folder, reflecting the changes seen in the actual interface when the real copies were moved.

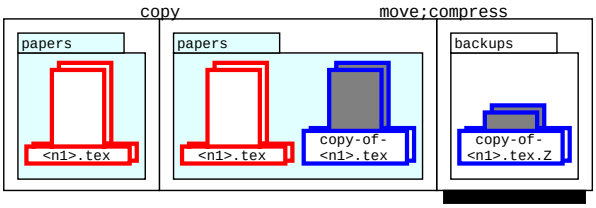

Fig. 5. The completed script. The `compress` operation is represented by the difference in the height and the name of the icon for the copies between the second and third panels. This difference is similar to the change in appearance of the icons for the real files on the desktop, where the `compress` operation replaces a file's icon with a shorter one and appends ".Z" to its name. To save space, Pursuit added the `compress` operation to the panel for the `move` operation, as indicated by the shadow.

the files to be copied and copies them. After the user moves the copies to the `backups` folder, the new panel in figure 4 appears depicting the move. Only one panel is added because Pursuit notices that *the epilogue of the* `copy` *contains the prologue of the* `move` *operation.*

Finally, the user selects all the copies and compresses them. Figure 5 shows the completed program. Another heuristic determines when several operations can be represented in a single panel. The shadow beneath the third panel indicates that it contains both the `move` and `compress` operations. By clicking on it, users can see the individual panels for the two operations.

4.2 More Complex Visual Programs

Next we illustrate Pursuit's representations of loops, conditionals and abstraction. Assume a user wishes to write a program to copy all the `.tex` files that were edited today. This could be done by demonstrating the script as in the previous example. However, suppose the `papers` folder contains a file named `copy-of-paper.tex`. If `copy` is executed on any set containing the `paper` file, it will fail because `copy` must produce an output file called `copy-of-paper.tex`. Therefore, the user must demonstrate this program on *two* examples – one in which the `copy` operation succeeds, and one in which it fails. In the latter case, the user must demonstrate what the program should do: delete the error causing file and re-execute the copy operation. Figure 6 shows the visual script after the second example is demonstrated.

After Pursuit confirms and executes the loop, it updates the visual script. The updated script (Figure 7) is an example of an explicit loop containing an explicit conditional. The panel in the upper left corner is a *declaration* and states

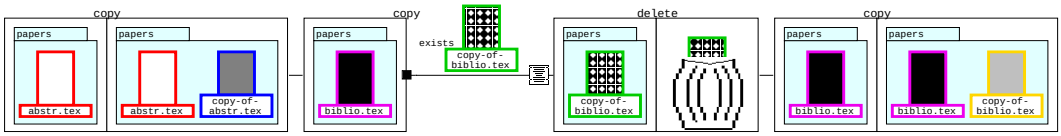

Fig. 6. The two examples used to demonstrate the copy program. The first two panels show a successful execution of the `copy` operation. The remainder of the script shows the unsuccessful execution and corrective actions. The black box on the prolog of the second `copy` operation (panel 3) indicates that the operation failed. The predicate following explains why – the existence of a file with the name `copy-of-biblio.tex`. The dialog box icon indicates that the operation popped up a dialog box. Clicking on the icon pops up the dialog box displayed when the operation failed. To correct the operation, the user deletes the error causing file (panels 4 and 5) and re-executes the copy operation (panels 6 and 7).

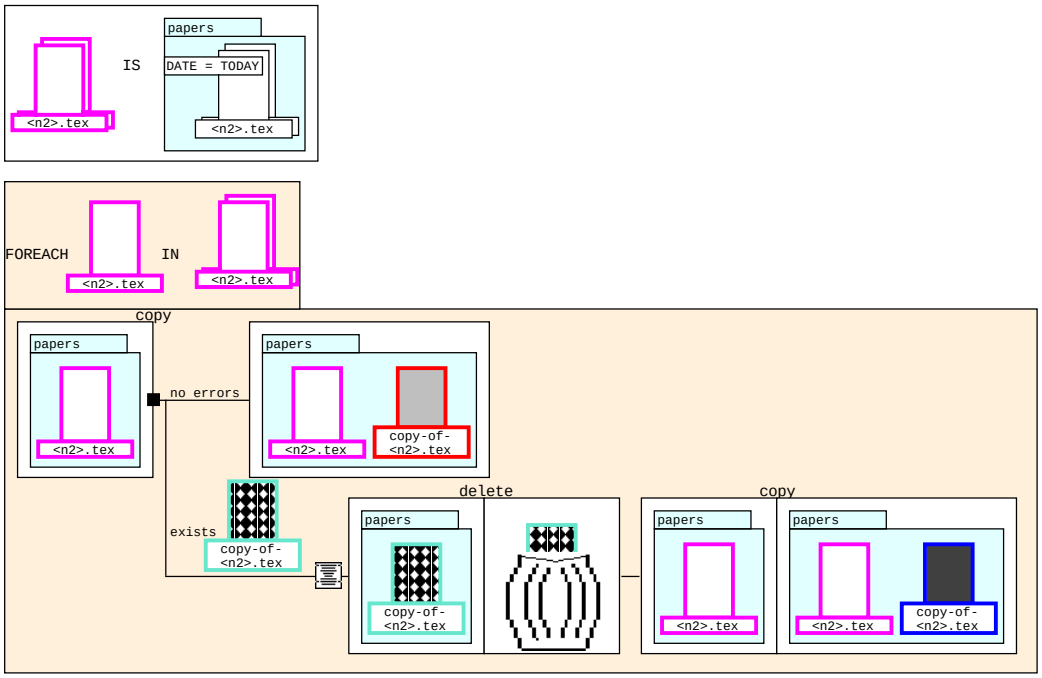

Fig. 7. The Pursuit script that copies each `*.tex` file in the `papers` folder that was edited today. If the `copy` operation fails because of the existence of a file with the output filename, the program deletes that old output file, and re-executes the `copy` operation. The upper panel is a declaration. It defines the loop set and depicts the inferences made by the PBD system.

that the loop iterates over the set of all `.tex` files in the `papers` folder that were edited today. The loop body is surrounded by the large outer rectangle. The black square on the right edge of the prologue of the first `copy` operation indicates that the program branches at this point. The first branch (containing the string "no errors") is taken when `copy` executes successfully. The lower branch is taken when `copy` fails because a file with the output filename already exists.

5 Feedback Beyond Program Representation

The main purpose of Pursuit's visual language is to represent demonstrated programs. In addition, Pursuit integrates the visual language into other parts of the PBD system. This helps improve feedback and forms a closer union between the PBD system and the program representation.

5.1 Inferring Iterations

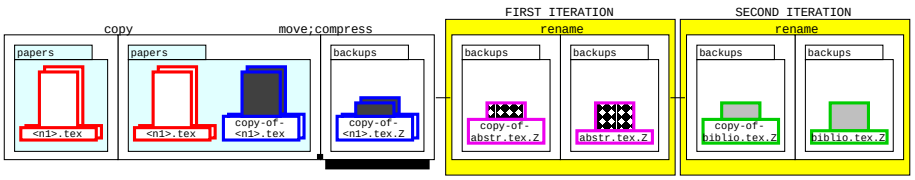

Fig. 8. When Pursuit detects a loop, it highlights the operations in the visual program so that users can confirm the inference.

Pursuit uses the visual script to indicate that it has inferred a loop. For example, suppose the user wishes to extend the program of Figure 5 so that the "copy-of-"prefix is removed from the compressed copies' names. To do so, the user removes the prefix from one of the copies and then another. Once Pursuit detects a loop, it highlights the panels containing the loop's operations (Figure 8). In this way, users can determine if Pursuit has correctly inferred the loop by looking at the highlighted operations.

Predictive methods (*e.g.,* Eager[4] and Metamouse[18]) force users to step through each operation of the iteration to verify the PBD system's inference. For a loop containing many operations, this could be tedious. To avoid this tedium, users can blindly trust the PBD system and have it complete the loop. They then can examine the interface to see if the PBD system was right. For loops that make many changes throughout the interface, this could be difficult and is very prone to error. By allowing users to confirm the loop's operations *before* it is executed, Pursuit reduces users' work and worry.

5.2 Inferring Sets and Subsets

When Pursuit identifies a set to loop over, it displays a dialog box containing the set's graphical representation (Figure 9). Users can confirm whether or not Pursuit has chosen the correct set by identifying the set's icon. This could be quicker and less error prone than displaying a dialog box listing all the set members.

Set icons are also used in a unique way in the visual script – to indicate the subset relationship. When Pursuit identifies one set as a subset of another, it defines the subset using the icon for the original set (Figure 10). In this way, the subset relationship is graphically depicted.

5.3 Editing Programs

Any PBD system will sometimes infer incorrectly or incompletely. Sometimes users might not think of all possible conditions that their program must work for.

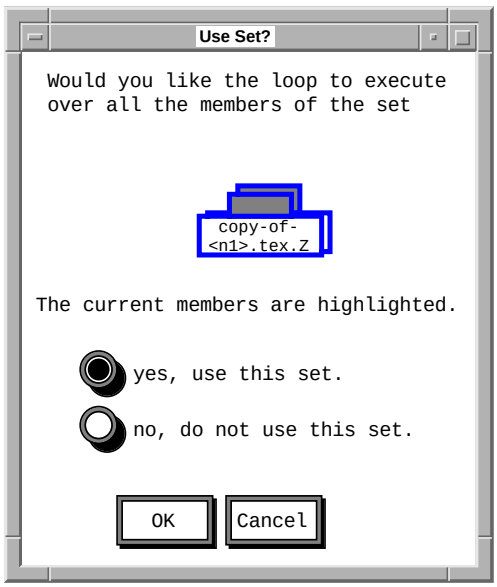

Fig. 9. A Pursuit dialog box asking the user to confirm the set to loop over. The icon for the set of files is the same icon found in the third panel of Figure 8. The dialog box appears when the user executes the `rename` operation on two members of the set representing the copied files.

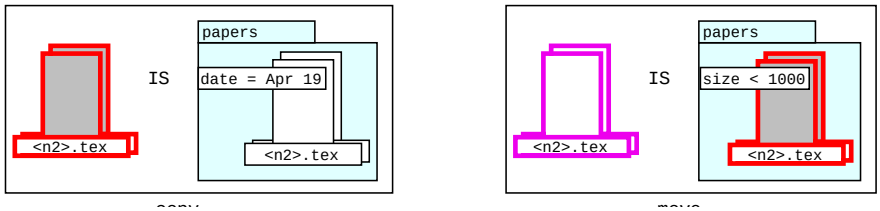

Fig. 10. Two Pursuit declarations. The first declaration defines the red set of files icon to represent all the `.tex` files in the `papers` folder that were edited on April 19. The second declaration defines the purple set of files icon to represent those files with size less than 1000 bytes in the red file set. Using the red file set icon in the second declaration concisely depicts the subset relationship.

Hence the PBD system must provide a way for users to edit a script. Pursuit uses a program's visual representation to assist in the editing process. For example, when the user has forgotten to demonstrate a program path and the program is executed in a state that causes it to go down that path, Pursuit pops up the visual representation of the program and highlights the path that the program has taken, indicating where the program is incomplete. The user can then continue demonstrating what the program should do and Pursuit automatically updates the script.

Data icons are also used by the editor. Sometimes users wishes to select a data object based on some criteria that cannot be expressed using attributes. For example, users might wish to check for the existence of an object or compare a property of one data object with a property of another data object. In this case users must construct an explicit branch with user defined predicates. To do so, the editor provides a menu of predicate templates describing all possible

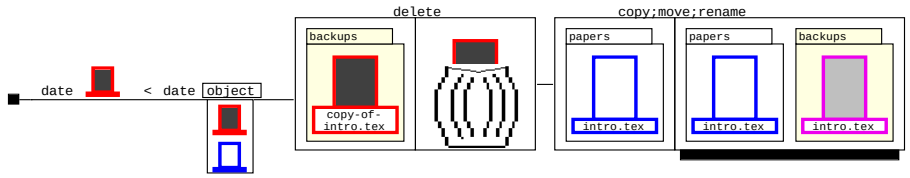

Fig. 11. The user is adding a branch before the demonstrated operations. To construct the predicate, the user selected a predicate template and directly edits it using the pull down menus in the template. The menus for objects in the template contain miniature representations of the objects' data icons.

forms that a predicate can take. Each template contains a list of menus for users to choose from in order to construct the predicate. To describe the data objects that users can select, each menu presents a list of the data objects using their visual representations (Figure 11). Users can select the desired data object by selecting its icon from the menu.

Other ways that data icons are used by the editor include indicating program parameters when saving a script, binding actual parameters to formal parameters when executing a script, and indicating the loop iteration set when wrapping a sequence of operations in an explicit loop.

6 Additional Benefits of Pursuit's Representations

The visual language representation provides a familiar and concrete representation of the program as well as immediate and easy to recognize feedback. It has several advantages.

6.1 Reducing the Use of Variables

Figure 5 illustrates an advantage of having icons represent data: the icons reduce the use of explicit variables, and remove a level of indirection that variables introduce. To identify an icon in a script, Pursuit assigns it a unique color. Although an icon's appearance may change throughout the script, its color remains the same. For example, in the second panel the set icon representing the output of the `copy` operation has the name "copy-of-<n1>.tex" and is in the `papers` folder. In the third panel, the same icon has the name "copy-of-<n1>.tex.Z" and is in the `backups` folder. Users can tell that two icons represent the same set because they have the same color. Color serves the same purpose as a variable name in textual programming languages.

6.2 Reducing the Use of Control Constructs

By concretely representing abstractions, attributes and sets, Pursuit reduces the need to explicitly represent loops and conditionals, constructs that are difficult for users to understand[6]. For example, in order to define the loop set in figure 7 in a traditional programming language, one would have to write code to loop through all the files in the `papers` folder and test to see which ones had names ending in `.tex` and were modified today. Attributes and sets make this looping and testing implicit.

6.3 Unifying the Interface, the Program and the Process

In general, programming forces users to navigate in two distinct work spaces[1]: the *physical* reality of the data, computer and programming language and the *psychological* reality of their mental model. The larger the gap between these work spaces, the more difficult the programming process[11], because users are forced to make paradigm shifts between the two spaces[3]. By integrating familiar visual representations within three different parts of the PBD system – the program representation, the communication of inferences, and the editor – Pursuit's visual language attempts to reduce these paradigm shifts. This saves time, decreases working memory load and reduces the chance for errors.

7 Status and Future Work

A prototype of Pursuit[21] has been implemented using Garnet[24]. We have been doing informal user studies to improve the system. For example, Pursuit initially contained a heuristic that often eliminated the prologue of the first operation of a program. Since several people had difficulty understanding scripts in which this heuristic was applied, it was eliminated. Further studies should help refine Pursuit's heuristics both in generating panels and generating attributes on sets.

In addition, we recognize that Pursuit programs can be space-inefficient and are exploring ways to remedy this, such as enabling users to select a sequence of panels to collapse into a single panel or automatically collapsing explicit loops into a single small panel.

Currently, Pursuit contains an editor to enable users to change set attributes; copy, cut and paste operations; add loops and user-defined branches; and demonstrate unfinished paths. We are expanding the editor to help insure that users do not create meaningless programs. For example, if the user deletes a `copy` operation then all subsequent operations that use the output copies could be deleted.

We are also exploring other ways of using visual representations as feedback, such as with animation. For example, consider the dialog box in Figure 9. Instead of suddenly appearing in the middle of the screen, a representation of the file set could emerge from the script, move to the center of the screen, and expand into the dialog box. Similarly, rather than disappearing instantly, the dialog box could shrink back into the set icon. Adding this animation would more closely link dialog boxes to the relevant section of the visual script.

Finally, formal user studies are planned to evaluate the system and determine how easy it is to use, how easy the visual language is to recall, and how well the system supports automating tasks.

8 Conclusion

We have presented a visual language that serves as both the main form of feedback and program representation for a demonstrational visual shell. By integrating visual feedback *throughout* the PBD system, Pursuit helps bridge the gap between users' mental model and the difficult task of programming. This should make the PBD system more effective.

9 Acknowledgments

The authors thank the participants at the Apple Computer 1992 Programming by Demonstration Workshop, especially Allan Cypher, David Kosbie, David Kurlander and David Maulsby, for helpful and continued feedback on this work. We

also thank James Landay for his comments on this paper. This research was supported by NSF grant number IRI-9020089 and by grants from the Hertz Foundation and AAUW.

References

1. D. Ackermann and J. Stelovsky. The Role of Mental Models in Programming: From Experiments to Requirements for an Interactive System. In P. Gorny and M.J. Tauber, editors, *Visualization in Programming*, pages 53–69. Springer-Verlag, 1986.
2. Kjell Borg. IShell: A Visual UNIX Shell. In *Proceedings of CHI '90*, pages 201–207, Seattle, Washington, April 1990.
3. Wayne Citrin. Visualization-Based Visual Programming. Technical Report CU-CS-535-91, Unversity of Colorado, July 1991.
4. Allen Cypher. EAGER: Programming Repetitive Tasks by Example. In *Proceedings of CHI '91*, pages 33–40, New Orleans, LA, April 1991.
5. Allen Cypher. *Watch What I Do: Programming by Demonstration.* The MIT Press, Cambridge, MA, 1993.
6. Stephanie M. Doane, James W. Pellegrino, and Roberta L. Klatzky. Expertise in a Computer Operation System: Conceptualization and Performance. *Human-Computer Interaction*, 5:267–304, 1990.
7. William Finzer and Laura Gould. Programming by Rehearsal. *Byte Magazine*, 9(6):187–210, June 1984.
8. P.E. Haeberli. ConMan: A Visual Programming Language for Interactive Graphics. In *ACM SIGGRAPH*, pages 103–111, Atlanta, Ga, 1988.
9. Daniel C. Halbert. *Programming by Example.* PhD thesis, Computer Science Division, University of California, Berkeley, CA, 1984.
10. Tyson R. Henry and Scott E. Hudson. Squish: A Graphical Shell for UNIX. In *Graphics Interface*, pages 43–49, 1988.
11. Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct Manipulation Interfaces. In Donald A. Norman and Stephen W. Draper, editors, *User Centered System Design*, chapter 5, pages 87–124. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986.
12. Branka Jovanovic and James D. Foley. A Simple Graphics Interface to UNIX. Technical Report GWU-IIST-86-23, The George Washington University, Institute for Information Science and Technology, Washington, DC 20052, 1986.
13. David Kurlander and Steven Feiner. Editable Graphical Histories. In *IEEE Workshop on Visual Languages*, pages 127–134, Pittsburgh, PA 15213, October 1988.
14. Henry Lieberman. Constructing Graphical User Interfaces By Example. In *Graphics Interface '82*, pages 295–302, Toronto, Ontario, Canada, May 1982.
15. Henry Lieberman. Capturing Design Expertise Interactively by Example. In *East-West Conference on Human-Computer Interaction*, St. Petersburg, Russia, August 1992.
16. Henry Lieberman. Mondrian: A Teachable Graphical Editor. In *Proceedings of InterCHI '93*, page 144, April 1993.
17. David L. Maulsby and Ian H. Witten. Inducing Programs in a Direct-Manipulation Environment. In *Proceedings of CHI '89*, pages 57–62, Austin, Tx, April 1989.
18. David L. Maulsby, Ian H. Witten, Kenneth A. Kittlitz, and Valerio G. Franceschin. Inferring Graphical Procedures: The Compleat Metamouse. Technical Report 91/388/12, Department of Computer Science, The University of Calgary, Calgary, Alberta, Canada, May 1990.
19. Francesmary Modugno. *Pursuit: Adding Programming to the Interface.* PhD thesis, Carnegie Mellon University, In progress.
20. Francesmary Modugno and Brad A. Myers. Graphical Representation and Feedback in a PBD System. In Allen Cypher, editor, *Watch What I Do: Programming By Demonstration*, chapter 20. MIT Press, 1993.

21. Francesmary Modugno and Brad A. Myers. Pursuit: Graphically Representing Programs in a Demonstrational Visual Shell. In *Proceedings of CHI '94*, April 1994. Video Program Presentation.
22. Brad A. Myers. *Creating User Interfaces by Demonstration.* Academic Press, Boston, Massachusetts, 1988.
23. Brad A. Myers. Demonstrational Interfaces: A Step Beyond Direct Manipulation. *IEEE Computer*, 25(8):61–73, August 1992.
24. Brad A. Myers et al. Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. *IEEE Computer*, 23(11):71–85, November 1990.
25. Ben Shneiderman. Direct Manipulation: A Step Beyond Programming Languages. *Computer*, 16(8):57–69, August 1983.

This article was processed using the LaTeX macro package with LLNCS style