Visual Representations as Feedback in a Programmable Visual Shell

Francesmary Modugno Brad A. Myers

March 1993
CMU-CS-93-133

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This work supported by the National Science Foundation under grant number IRI-9020089. The first author is supported by a fellowship from the Fannie and John Hertz Foundation. 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.

Keywords: User Interfaces, Intelligent Interfaces, End-User Programming, Programming by Demonstration, Demonstrational Systems, Visual Languages

Abstract

Although visual shells (e.g., Apple Macintosh) are easy to use, they are difficult to program, especially by non-expert users. PURSUIT is a visual shell that contains a programming by demonstration (PBD) system to address this problem. In a PBD system, users execute actions on data and the underlying system attempts to construct a program. Such systems, however, have limitations: feedback is often difficult to understand, disruptive or non existent; and programs often have no representation for users to examine or edit.

PURSUIT addresses these problems by presenting the inferred program in an editable *graphical* language *while it is being constructed.* The language is a state-based visual language in which icons represent objects such as files and folders and changes in these icons represent operations. The representations are similar to the objects and effects that users see in the real interface. This provides users with an immediate view of the code and feedback in a way that is familiar, easy to understand, and non-disruptive. An intuitive representation that also serves as feedback makes it easier for users to write programs to customize their systems.

1 Introduction

A visual shell, like the Xerox Star or the Apple Macintosh, is a direct manipulation interface to an operating system. Such interfaces were developed to hide the complexity of the underlying operating system from the user. Ideally, these systems should be easy to use yet powerful enough to allow novice and non-expert users to tailor the system. Users should easily be able to write programs to handle repetitive tasks, such as "back up all *TEX* files in the current folder", and simple transformations, such as "change zip code 15213 to 15213-3890 in all my letters". While existing visual shells are easy to use, customizing them is difficult or impossible. How can we provide this tailoring capability *in way that is as easy to use as and consistent with the direct manipulation paradigm*?

Programming by Demonstration (PBD) systems can potentially solve this problem. In a PBD system, users execute actions on examples and the system constructs a general program. Such systems enable users to create general procedures without having special programming skills. They are easy to use because users operate 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 many provide no editing facilities. This makes it difficult for users to know if the system has inferred correctly and to correct any errors. It also makes it difficult to revise or change a program.

In order to address these problems, we are exploring the use of a novel *graphical* representation of the program *while it is being written*. Programs are represented in a state-based 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. By overcoming some of the problems with PBD systems, this approach makes PBD techniques more effective in helping users easily customize their systems.

2 Why Represent the Program at All

There are several reasons to have a static representation of the program. First, it allows users to review the program's code and verify that it does what they intended. It also gives users the opportunity to alter the code.

For example, they may need to correct errors (both on the part of the system and the user); they may desire to generalize the program further; they may wish to update the program sometime in the future; or, they may wish to use the code as a basis for solving a similar problem.

Finally, providing a representation of the program *during* the demonstration provides the user with a source of feedback. *Feedback* in a PBD system is the form of communication between the system and the user. It serves to inform the user of the system's inferences; to enable the user verify these inferences; and to obtain guidance from the user as to the salient features of the example over which to generalize. There are many forms that this feedback can take: dialog boxes (e.g. property sheets in SmallStar [5]); questions and answers (e.g. Peridot [12]) and Metamouse [10]); textual representation of the code (e.g. Tinker [8]); meta character in the program area (e.g. the cat icon in Eager [3]); animation (e.g. rehearsals in Rehearsal World [4]); and sound (e.g. Mondrian [9]).

Our approach has several benefits over these forms of feedback. Unlike dialog boxes and the question and answer style, it is not disruptive, since the user does not need to respond to it. Unlike programs represented in a textual language, it does not require the user to learn a new language that is very different from the interface. Unlike meta characters used in the programming area, users do not need to learn about special characters, their uses or how to respond to them. Having a good visual representation of the developing program, therefore, is easy to understand, not disruptive and gives the users full knowledge of the system's inferences.

3 Why Use a Visual Language?

One reason for using a graphical representation is that in both visual shells and visual languages the objects of interest are represented using pictures and symbols. Users can apply some knowledge of how an object's icon is manipulated in the interface to how its representation is manipulated in the program.

Another reason for using a visual language is that visual programs are easier to understand than textual ones, especially for short programs [2] Many of the problems with visual languages, such as scalability, won't arise in the visual shell domain because most programs users tend write are rel-

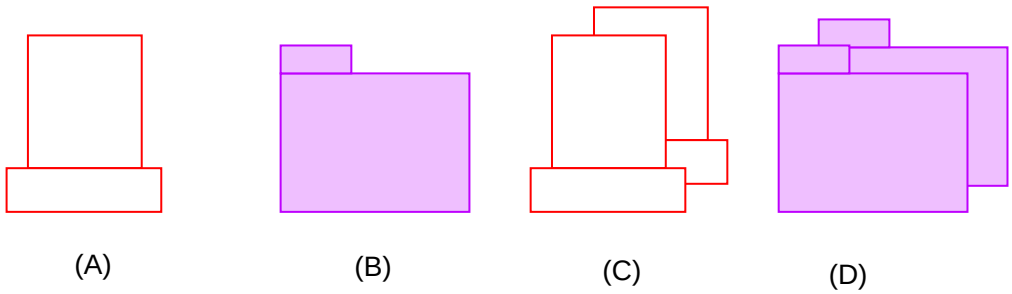

Figure 1: The basic data types in the visual language: (a) a file; (b) a folder; (c) a set of files; (d) a set of folders.

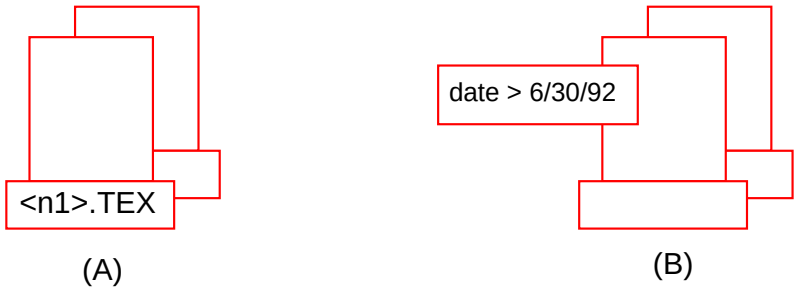

Figure 2: Attributes attached to set icons constrain the objects in the set to have particular properties. They are also used to depict system inferences. For example, the first set (a) contains all those files whose name ends in *.TEX*. The second set (b) contains all those files edited before June 30, 1992.

atively short and simple. They are usually no more than a page in length and often do not contain nested loops or nested conditionals [1].

The benefits of the language described here are further discussed below.

4 The Representation Language

The visual language is based on the comic strip metaphor [6]. Icons are used to represent data objects and changes in the data objects represent operations.

The language contains two basic data types: files and folders. These are represented using familiar icons. Sets of data objects are represented by overlaying two icons of the same type and offsetting them by some small distance (see figure 1). In order to allow for abstract sets of objects, graphical constructs called *attributes* are attached to set icons. Attributes constrain the objects in the set to have a particular value for a certain property, such as their date of modification (figure 2). Attributes serve to indicate the underlying PBD system's inferences.

Associated with each data object is a set of graphical properties, such as its location, icon appearance, name, etc., that represents properties of the real object in the interface. An operation is shown by the changes it causes to the graphical properties of data objects. Two panels are used to represent an operation: the *prologue*, which contains the data object before the operation has occurred; and the *epilogue*, which contains the data objects after the operation has occurred and shows any changes to the objects caused by the operation. An operation's representation, therefore, looks very much like the changes the user sees in the real user interface when executing the operation (e.g., figure 4).

A program is a series of operation panels concatenated together, along with representations of control constructs such as loops and conditionals. However, because two panel per operations may result in long, space inefficient scripts, the language generator contains heuristics for making programs more concise. For example, when the epilogue of an operation contains the prologue of the subsequent operation the two panels may be combined into one. Also, when the prologue of an operation contains no useful information it may be omitted. These heuristics and other features of the language are illustrated in the following example.

5 An Example

Assume that periodically a user backs up all the *.TEX* files in her "papers" folder. To backup the files, she copies them to the "backup" folder and then compresses the copies. In order to write a program to do this automatically, the user demonstrates the actions of the program on a particular set of files. During the demonstration, the underlying PBD system attempts to create a program. While the program is being constructed, a visual representation

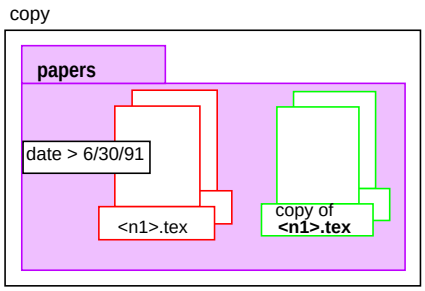

Figure 3: The copy operation. The attributes on the files set icon indicate the inferences the system made: all copied files were last modified after 6/30/91 and have a name of the form *<n1>.tex*. All files in the set must satisfy these two properties.

of it appears.

Figures 3-5 show a simulation of the developing visual script during the demonstration. The first panel (figure 3) appears after the user opens the "papers" folder, selects all those files to be copied and copies them. Notice that the copy operation is represented with only one panel. This is because the prologue for the copy operation provides no extra information than the epilogue already provides. Therefore, the first panel is redundant and is omitted. This is an example of a heuristic to make the visual representations concise.

After the user moves the copies by clicking on them and dragging them to the "backup" folder, the new panel in figure 4 appears depicting the move. Only one panel is added to illustrate the move, since 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.

6 Benefits of this Representation

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

Using icons to represent data minimizes the use of explicit variables and

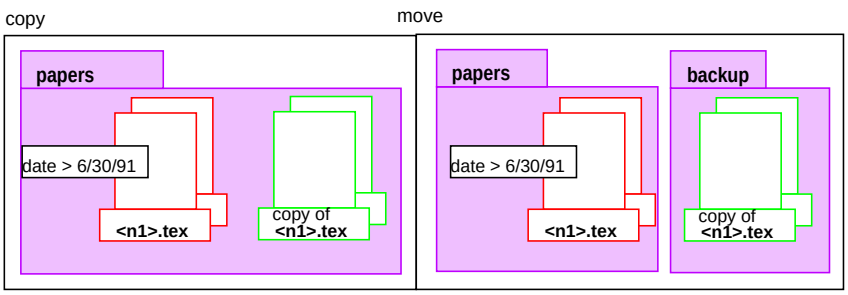

Figure 4: After the user drags (moves) the copies to the "backup" directory, the second panel appears. Notice that in the script the set of copies icon has moved from the "papers" folder to the "backup" folder, reflecting the changes the user sees in the actual interface.

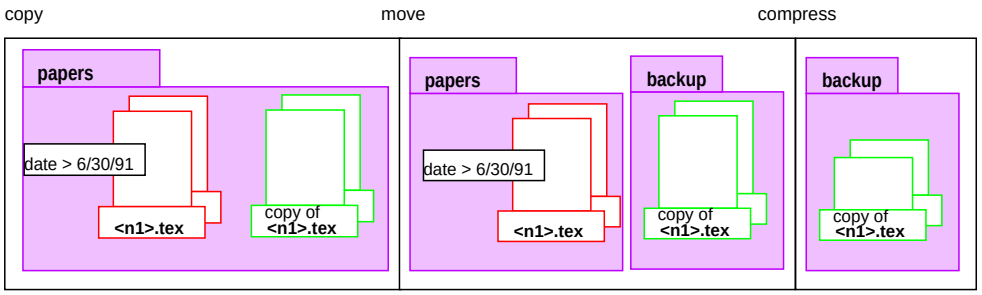

Figure 5: The completed script. The compress operation is represented by the difference in height of the icons for the copies in the second and third panels. This difference is similar to the change in appearance of the icons of the real files that the user would see in the actual interface.

assignment statements. This makes programs more concrete. In addition, data icons are very similar in appearance to the desktop objects they represent. Languages that represent data as closely as possible to the real objects are easier to understand [7].

Similarly, operation representations are easy to understand because the changes in objects in a program mirror the changes in the actual desktop objects when the operations are executed. Also, by representing operations in this familiar way, users do not need to learn special "code" or languages to understand a program. In essence, programs become symbolic representations of changes to data over time.

Finally, the use of attributes and set objects also helps to minimize the use of traditional control structures, such as loops and conditionals. These program constructs have proven difficult for novices to understand.

7 Conclusion

We have presented a novel design for a visual language to serve as both the form of feedback and the representation of program code for a PBD system in a visual shell domain. The language incorporates some of the same principles of cognition that have made spreadsheets so successful: the use of familiar, concrete representation; and the use of immediate feedback. Providing this representation and feedback makes the PBD system more effective, making the customization task easier. Initial reaction to the language has been positive.

Currently, the language is being implemented as part of the PURSUIT [11] interface. Using this prototype, we are exploring heuristics, such as omitting prologue panels and combining several operations into a single panel, to make the visual representation more concise. We are also extending the language to include representations of other objects, such as dialog boxes, windows and menus. Because we would like the visual language to include representations of utilities that users may add to the system, we have designed a simple declarative language for specifying the graphical representations of operations. This enables the visual language to be extended easily.

References

[1] Keys Botzum. An empirical study of shell programs. Technical Report in Progress, Bell Communications Research, 1992.

[2] Nancy Cunniff and Robert P. Taylor. Graphical vs. textual representation: An empirical study of novices' program comprehension. In Gary M. Olson, Sylvia Sheppard, and Elliot Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 114–131, Washington, D.C., December 1987.

[3] Allen Cypher. Eager: Programming repetitive tasks by example. In *Proceedings of CHI '91*, pages 33–40, New Orleans, LA, April 1991.

[4] William Finzer and Laura Gould. Programming by rehearsal. *Byte Magazine*, 9(6):187–210, June 1984.

[5] Daniel C. Halbert. *Programming by Example*. PhD thesis, Computer Science Division, University of California, Berkeley, CA, 1984.

[6] David Kurlander and Steven Feiner. Editable graphical histories. In *Workshop on Visual Languages*, pages 127–134, Pittsburgh, PA 15213, October 1988. IEEE.

[7] Clayton Lewis and Gary M. Olson. Can principles of cognition lower the barriers to programming? In *Empirical Studies of Programmers: Second Workshop*, pages 248–263. Ablex Publishing Corporation, 1987.

[8] Henry Lieberman. Seeing what your programs are doing. *Int. J. Man-Machine Studies*, 21(4):311–331, October 1984.

[9] Henry Lieberman. Mondrian: A teachable graphical editor. unpublished notes, 1992.

[10] David L. Maulsby, Ian H. Witten, and Kenneth A. Kittlitz. Metamouse: Specifying graphical procedures by example. In *Computer Graphics*, pages 127–135, Boston, Massachusetts, July 1989. ACM SIGGRAPH.

[11] Francesmary Modugno. *Pursuit: Adding Programming in the Interface*. PhD thesis, Carnegie Mellon University, In progress.

[12] Brad A. Myers. *Creating User Interfaces by Demonstration*. Academic Press, Boston, Massachusetts, 1988.