Pursuit: Visual Programming in a Visual Domain

Francesmary Modugno Brad A. Myers

January 1994
CMU-CS-94-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We present a new visual programming language and environment that serves as a form of feedback and representation in a Programming by Demonstration system. The language differs from existing visual languages because it explicitly represents data objects and implicitly represents operations by changes in data objects. The system was designed to provide non-programmers with programming support for common, repetitive tasks and incorporates some principles of cognition to assist these users in learning to use it. With this in mind, we analyze the language and its editor along cognitive dimensions. The assessment provides insight into both strengths and weaknesses of the system, suggesting a number of design changes.

This work supported by the National Science Foundation under grant number IRI-9020089. Additional support provided by the Hertz Foundation and the AAUW. 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: Cognitive Dimensions, End-User Programming, Programming by Demonstration, Visual Language, Visual Shell, Pursuit

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 (Shneiderman, 1983). 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 ("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. Pursuit (Modugno, in progress) is a visual shell aimed at providing end-user programming capabilities *in a way that is consistent with the direct manipulation paradigm.*

To enable users to construct programs, Pursuit contains a Programming by Demonstration (PBD) system (Cypher, 1993). In a PBD system, users execute actions on real data and the underlying system attempts to construct a program (Myers, 1991). 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.

Pursuit addresses these problems by presenting the evolving program in a *visual* language *while it is being constructed.* Unlike other visual languages, which explicitly represent operations and leave users to imagine data in their heads, Pursuit's visual language explicitly represents data objects using icons and implicitly represents operations by the visible changes to data icons.

In general, programming forces users to navigate in two distinct work spaces (Ackermann and Stelovsky, 1986): 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 (Hutchins, Hollan and Norman, 1986), because users are forced make paradigm shifts between them (Citrin, 1991). There are four properties of Pursuit that attempt to reduce the gap between the two spaces. First, programs are specified by executing real actions on real data. The Pursuit PBD system, therefore, extends the direct manipulation paradigm to enable users to *specify* programs in much the same way that they invoke operations – through direct manipulation. Second, programs are represented in a visual language in which the data and operations of a program look very much like the actual objects and changes users see on the desktop when constructing the program. Hopefully, this will make Pursuit programs look more familiar than programs written in a textual language or in a visual language that does not reflect the interface. Third, during a demonstration, the program appears incrementally as the user executes each operation. In this way, users learn *interactively* how data and operations (*i.e.* program syntax) are represented. Finally, Pursuit's visual representations are integrated within three different parts of the PBD system – the program representation, the communication of inferences, and 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. Hopefully, the combination of these four properties will make it possible for novices and non-programmers to learn to construct and recognize Pursuit programs.

This paper describes the Pursuit visual language and its editor. It then reports our experience using cognitive dimensions (Green, 1989 and 1991) to determine how well Pursuit meets its design criteria. Finally, we suggest a number of design changes to improve the language and editor based on this analysis.

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 (Halbert, 1984); 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.*, Jovanovic and Foley, 1986; Haeberli, 1988; Henry and Hudson, 1988; Borg, 1990) 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 PURSUIT'S 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 (Shneiderman, 1983). 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 interface and its objects to the visual language and its objects when constructing, viewing and editing a program. With this in mind, this section briefly discusses the motivation and history for some of Pursuit's design decisions.

3.1 Extending Programming to Non-Programmers

We set out to create a visual shell that would enable users to access the underlying power of the computer to help with their tasks, without requiring that they learn complex programming skills, or many special "little languages" or commands. We were particularly interested in providing this power to people who use computers frequently, but who might not want to learn how to program (we refer to these users as "non-programmers"). We began by studying the login files of and shell scripts written by computer scientists at Carnegie Mellon and by informally interviewing some

non-programmers, such as secretaries and administrators. Our goal was to determine the types of programs users wrote and the types of tasks non-programmers do that could be automated. Our studies showed that most shell scripts (including those written by programmers) were very simple, repetitively executing a few commands over a set of files. Often, the files were related in a simple way, such as all being `.tex` files or all being edited on a certain day. A more rigorous study supported these findings (Botzum, in progress). The informal discussions revealed that many of the repetitive tasks that non-programmers do, such as backing up files, were similar in form to the shell scripts.

3.2 Incorporating Principles of Cognition

Our goal became to design a system that would simplify this type of programming. Therefore, we created a language that emphasizes the manipulation of *sets* of objects related in some specific way and that minimizes the use of explicit control constructs such as loops and conditionals, since novice and non-programmers often have difficulty with them (Doane, Pellegrino and Klatzky 1990). In addition, as we describe throughout the paper, the language and editor incorporate some of the same principles of cognition that have made spreadsheets successful (Lewis and Olson, 1987): familiar, concrete representations; immediate feedback; suppressing the inner world of variables and computation; and automatic consistency maintenance. We feel that this will help users learn to understand and use the language.

3.3 Providing Editable, Visual Feedback

Finally, we designed the language to serve as the main form of feedback between the PBD system and the user. Pursuit contains an inference mechanism[1] that can detect loops over sets of data, branches on exit code conditions, and common substring patterns (see Modugno, in progress, for details). Since all inference mechanisms will sometimes be wrong, it is important that users know what the system has inferred. Having a good representation of the program *during* the demonstration gives users full knowledge of the system's inferences at all times in an unobtrusive way. Users can verify these inferences and help guide the PBD system to which features of the examples should be generalized. Furthermore, Pursuit's visual language is not only used to represent programs, but it is also used to indicate 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 (Halbert, 1984); questions and answers (Myers, 1988; Maulsby and Witten, 1989); textual representation of the code (Lieberman 1982); changing the appearance of actual interface objects (e.g. anticipation highlighting (Cypher, 1991); animation (Finzer and Gould, 1984); and sound (Lieberman, 1993). However, our approach has several benefits over these other forms. 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 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.

[1]This research does not focus on improving inferencing. It focuses on other limitations of PBD systems: feedback, representation and editing. The techniques described here are independent of the inference mechanism.

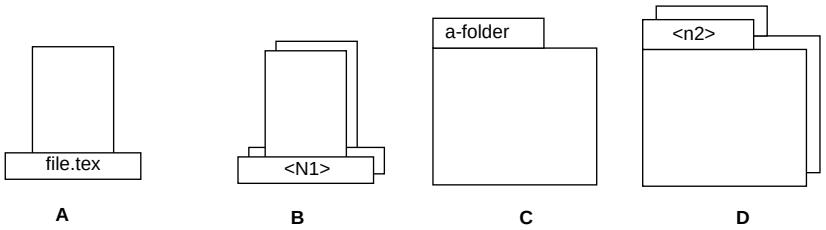

Figure 1: The main data types in Pursuit: a) a file; b) a set of files; c) a folder; d) a set of folders.

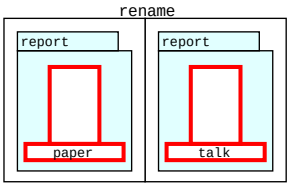

Figure 2: The representation of the operation `rename paper talk`. The first panel shows the icon representing the file `paper` located in the `report` folder before it is renamed. The second panel shows the same file after the `rename` operation. Notice that the icon's name has changed. This change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

4 PURSUIT'S VISUAL REPRESENTATION LANGUAGE

The Pursuit visual language[2] combines elements of the comic strip metaphor (Kurlander and Feiner, 1988) and the visual production system (Furnas, 1991; and Bell and Lewis, 1993). Familiar icons are used to represent data objects, such as files and folders. Sets are represented by overlaying two icons of the same type (see Figure 1). Two panels are used to represent an operation. The *prologue* shows the data objects before the operation and the *epilogue* shows the data objects after (see Figure 2). A program ("visual script") is a series of operation panels concatenated together, along with representations for loops, conditionals, variables and parameters. Because two panels per operation result in long, space inefficient scripts, Pursuit contains space saving heuristics that combine knowledge of the domain with information about operations. These and other features of Pursuit are illustrated in the examples below.

In order to be extensible, Pursuit contains a declarative specification language (see Modugno, in progress, for details) used to specify operations. An operation's specification defines its visual representation, its error conditions and dialog boxes, and how it affects the graphical appearance of data objects. During a demonstration, Pursuit uses this specification to generate a representation of the operation in the visual program.

4.1 Example 1

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

[2]We are not interested in designing the most visually appealing language. Instead, we are exploring the utility of this particular language paradigm for non-programmers. Hence, our visual representations appear primitive.

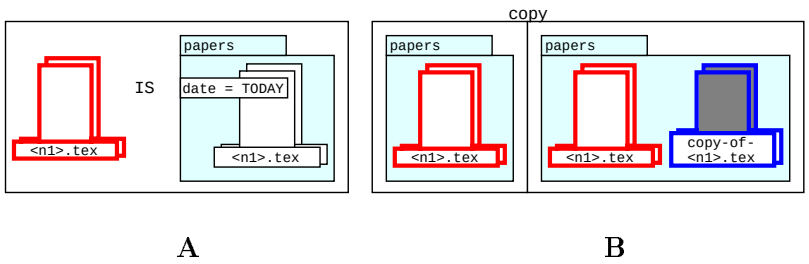

Figure 3: A). A visual declaration binding the set to be all the `.tex` files in the `papers` folder that were edited today. B). The `copy` operation.

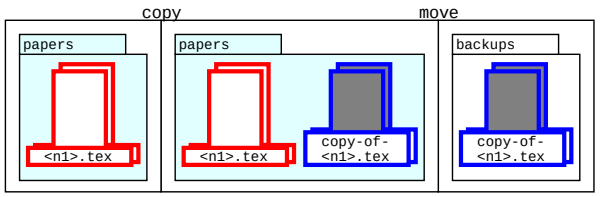

Figure 4: After the user drags (moves) the copies to the `backups` folder, the third panel appears. Notice that in the script the set of copies icon has moved from the `papers` folder to the `backups` folder, reflecting the changes the user has seen in the actual interface when the real copies were moved.

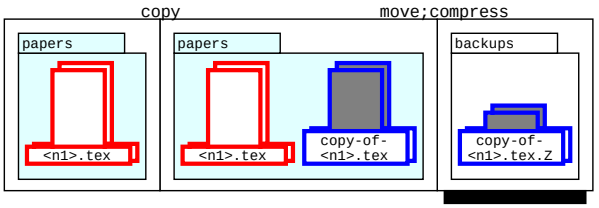

Figure 5: The completed script. The `compress` operation is represented by the difference in the height and the name of the icon for the copies in the second and third panels. This difference is similar to the change in appearance of the icons for the real files that the user would see in the actual interface (see Figure 6), where the `compress` operation replaces a file's icon with a shorter icon and appends a ".Z" to its name. The shadow beneath the third panel indicates that it represents multiple operations. Clicking on it reveals the individual operation panels.

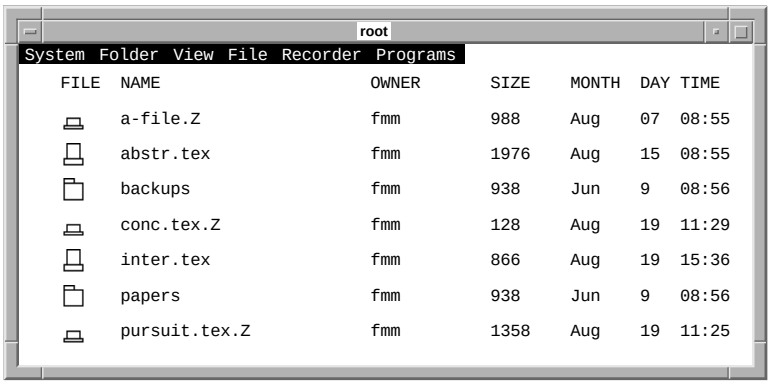

Figure 6: A folder on the Pursuit desktop. The folder contains two different types of file objects, compressed and uncompressed files, as well as folder objects.

Figures 3-5 show the developing visual script *during* the demonstration. Figure 3A is a *visual declaration*. It appears when the user executes the `copy` operation, and defines the scope of the set variable. The icon on the right represents the set of all `.tex` files in the `papers` folder that were edited today. The icon on the left is the icon used in the script to represent this set. The string "date = TODAY" is an *attribute*. It constrains the set to those files edited today. Attributes allow for abstract sets of objects and indicate the PBD system's inferences. In addition, users can directly edit them to specify desired properties of data objects or to fix incorrect inferences. Currently, the attributes Pursuit supports include an object's name, date, location, size, owner and contents. Attribute strings can be simple arithmetic expressions defining a single value or a range or values (e.g. "256 < size < 1024") and can contain variables, system constants such as "TODAY" or "USER", and references to attributes of other objects.

Attributes and sets minimize the need for loops, conditionals and variables. For example, to define the above set 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, thus hiding some inner computations from users.

Figure 3B shows the first two panels as they appear after the user opens the `papers` folder, selects the files to be copied and copies them. After the user moves the copies to `backup` 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. Determining when to combine the prologue of an operation with the epilogue of the previous operation is an example of a Pursuit space saving heuristic.

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

Figure 5 also illustrates an advantage of having icons represent data: icons minimize 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 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 set has the name "copy-of-<n1>.tex.Z" and is in the `backups` folder. Users can tell that the 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. In these languages, a variable is used to denote a data object. Operations manipulate a variable, but affect only the data that the variable represents. The variable name (or its representation) is not changed. However, in Pursuit operations affect the visual representations of icons, which serve as "variables" representing data objects. Color insures that the link between an icon and the data it represents is not lost when the icon's appearance changes.

4.2 Example 2

This example illustrates how Pursuit automatically creates conditionals and loops. When an operation fails, Pursuit cannot construct an epilogue panel. Instead, it creates a *conditional marker* (*e.g.* the black square on the right side of the third panel in Figure 7) and a *branch connector* with an annotation (or predicate) stating the condition for that branch. In this example, the `copy` operation failed because a file with the required output name already existed, so the annotation is "exists" plus a named file icon.

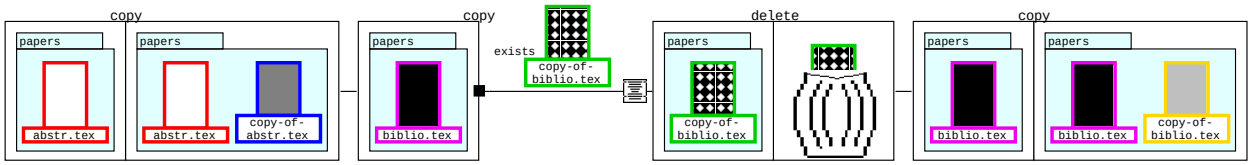

Figure 7: The two examples used to demonstrate a conditional program. The first two panels show a successful execution of the `copy` operation. The remainder of the script shows the unsuccessful execution and corrective actions the user wants the program to take in that case. The black square on the prologue 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 to the user. 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).

To build a conditional program, the user demonstrates the program's actions on two examples – one in which the operation succeeds, and one in which it fails. When the user begins to demonstrate the program on a third example, Pursuit recognizes a loop. It asks the user to verify the two loop iterations by highlighting the panels in Figure 7. It then finishes executing the loop and updates the visual script. The updated script (shown in Figure 8) is an example of an explicit loop containing an explicit conditional.

The panel below the visual declaration states that the loop executes over all the files in the declaration set. The operations in the body of the loop are surrounded by the large loop rectangle. The conditional marker the right edge of the prologue of the `copy` operation indicates that the program branches at this point. The first branch (labeled "no errors") is taken when the `copy` operation executes successfully. The lower branch is taken when the `copy` operation fails because a file with the output file name (`copy-of-<n1>.tex` in this case) already exists. To see the remaining exit conditions of the `copy` operation, users can click on its conditional marker. This helps users interactively consider all possible paths a program may take when executing.

This example also illustrates how Pursuit handles dialog boxes. Applications use dialog boxes to relay messages or obtain information, such as the name of an output file, from users. In Pursuit, applications use a special mechanism to display dialog boxes. If a dialog box appears when recording a program, Pursuit tries to determine if it can compute user responses automatically, so than when the program is executed no user intervention would be required. Pursuit then pops up a "meta" dialog box asking the user to confirm the inferences (if there are any) and to state whether or not the dialog box should appear whenever the program is executed (see Figure 9). To allow users to view or edit dialog box responses, representations of dialog boxes are included in the script. An example is shown on the lower branch of the `copy` operation in Figure 8. By clicking on the dialog box icon, users can see the Pursuit meta dialog box containing both an abstraction of the dialog box that appears when the `copy` operation fails and the user's response to Pursuit's query.

The previous example illustrated how Pursuit handles dialog boxes that relay messages to users. Pursuit uses these same abstraction and inference mechanisms for dialog boxes that obtain information from users. When recording a script, if such a dialog box appears, Pursuit pops up a "meta" dialog box asking users how the information for each field of the dialog box is to be obtained when the program is executed: by being computed by the program, entered by the user, or remaining constant. An icon representing this Pursuit dialog box appears in the script and users can change their responses by clicking on the icon and editing the displayed meta dialog box directly.

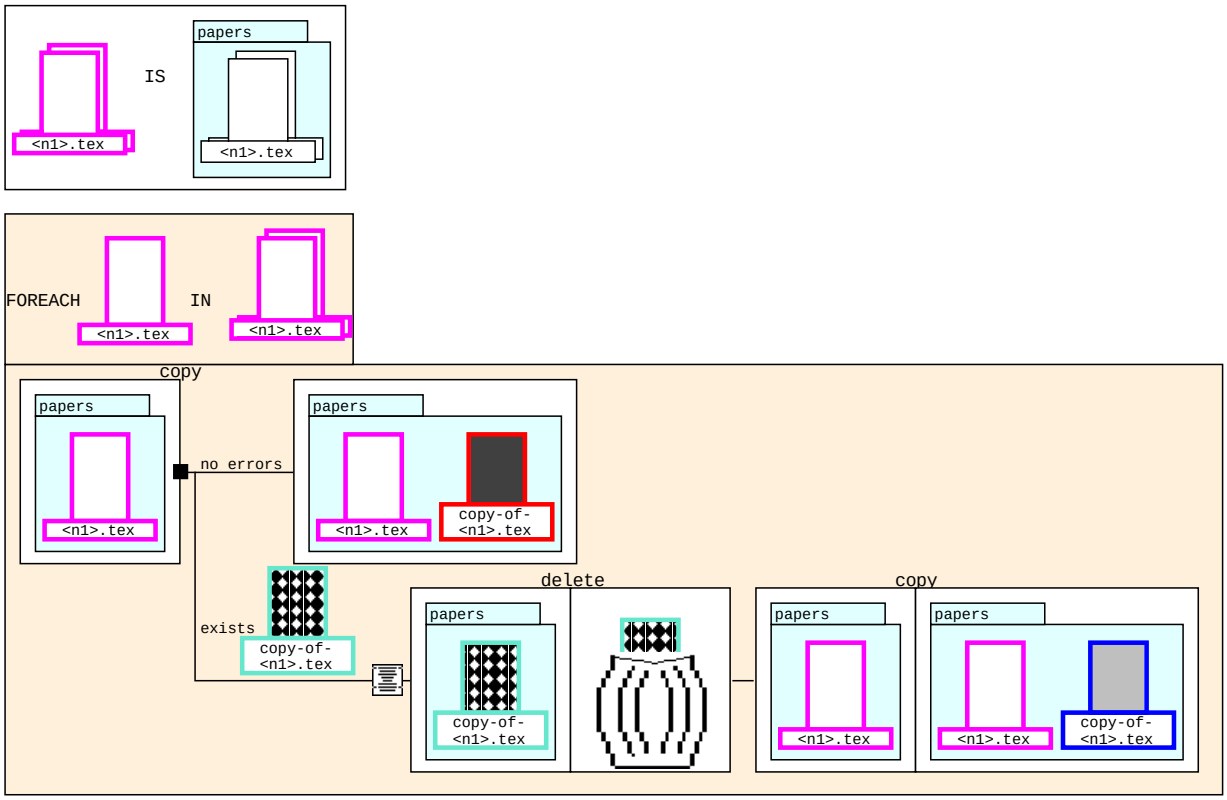

Figure 8: The Pursuit script that copies each `*.tex` file in the `papers` folder. If the `copy` operation fails because of the existence of a file with the output file name, the program deletes that old output file, and re-executes the `copy` operation. Users can see the other possible outcomes of the `copy` operation by clicking on the conditional marker.

4.3 Comparison to Chimera and Mondrian

Although Pursuit's representation language is similar to the editable histories of Chimera (Kurlander and Feiner, 1988), there are many important differences. First, Pursuit's visual language contains *abstractions* that resemble the real interface objects they represent. In contrast, Chimera uses actual screen snapshots in their representations. Also, Pursuit panels contain only the objects affected by the operation, since Pursuit objects can be identified by their icons. On the other hand, Chimera panels contain objects not involved in operations (such as the cursor) in order to provide contextual information to help identify objects and operations. Furthermore, Pursuit's inferences are displayed in the visual script and are always visible, whereas Chimera's inferences are contained in textual supplements and are not visible in scripts. In addition, Pursuit scripts are two dimensional. Information is conveyed from left to right and top to bottom (see figure 8). This makes the language more powerful than Chimera, which use only a linear (left to right) display. Finally, Pursuit scripts visibly represent loops and conditionals, which are inferred automatically. Chimera macros must be edited to contain loops, which have no explicit representation and Chimera contains no mechanism for inferring, adding or representing conditionals.

Mondrian (Lieberman, 1993), a demonstrational graphical editor, also uses a similar representation paradigm. Like Chimera, Mondrian programs are one dimensional focused snapshots of the screen augmented with other objects needed for context. Like Pursuit, Mondrian interactively ob-

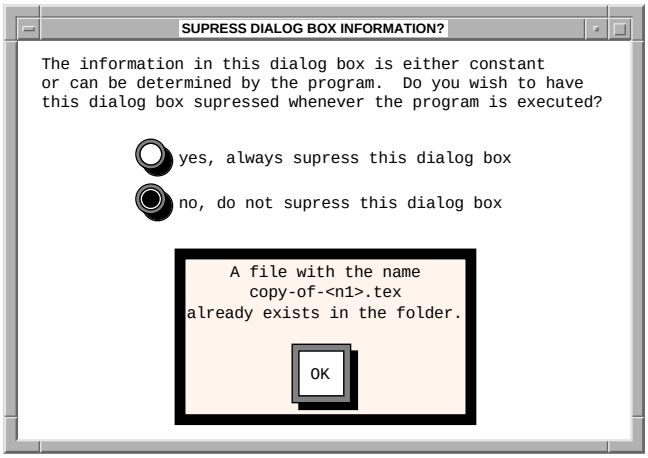

Figure 9: The Pursuit "meta" dialog box displayed when the user clicks on the dialog box icon in Figure 8 The user has indicated that the inner dialog box should not be displayed if the `copy` operation fails during program execution. Pursuit has inferred how the file name in the dialog box is computed and replaced it with the abstract representation "copy-of-<n1>.tex". Pursuit made the inference using information about the dialog box's contents contained in the declarative specification of the `copy` operation.

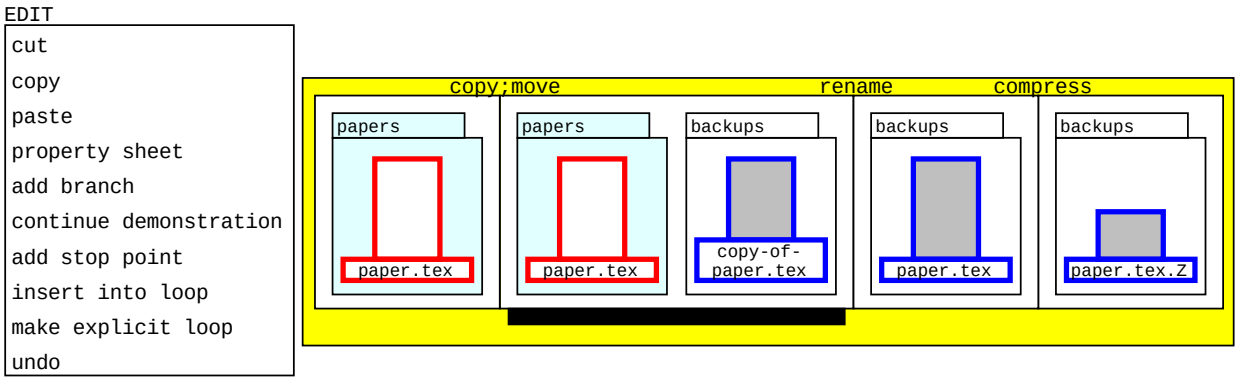

Figure 10: The user has selected a sequence of panels (indicated here by the outer gray rectangle) to wrap in a loop.

tains advice from the user to help with generalizing. Mondrian is unique in its use of speech as a form of feedback, the ability to create recursive procedures by including an iconic representation of the developing program in that program during the demonstration, and the ability to add textual annotations.

5 THE PURSUIT EDITOR

Pursuit contains a visual language editor. This enables users to fix incorrect inferences, add or delete operations, change attributes, add loops and conditionals, select and name parameters, etc. The editor is similar to a direct manipulation text editor. Data objects are selected by clicking on them, and operations are selected by clicking and dragging the mouse across their panels. Once an object is chosen, appropriate editing commands appear in the edit menu (see Figure 10). For example, operations can be cut or copied into the cut buffer and pasted into another section of the program, or they can be wrapped in a loop. File and folder objects can be edited to add, remove or change attributes, or to make them into parameters.

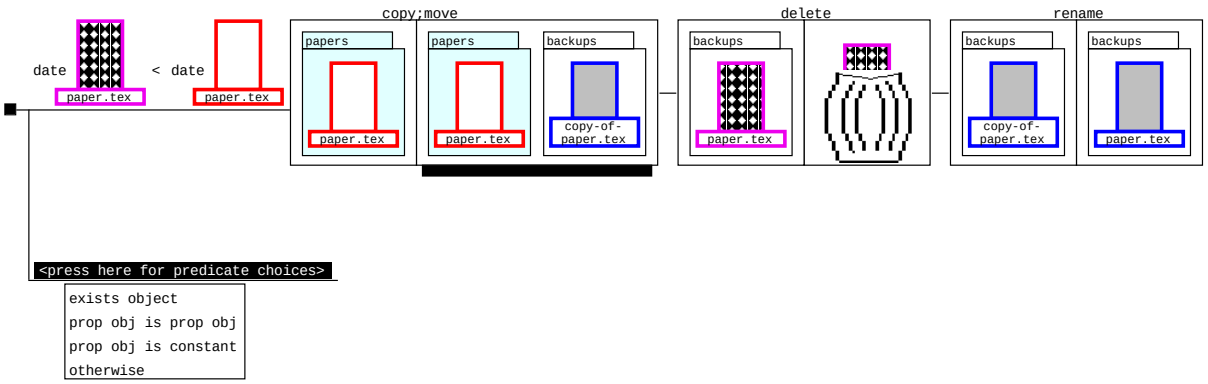

Figure 11: Adding a user defined branch to a script. The upper branch shows the path to take when the file `paper.tex.Z` in the `backups` folder is older than the file `paper.tex` in the `paper` folder. The user is adding another predicate to branch construct by selecting it from the menu of predicate templates. The menu choices provide templates for the user to further edit in order to construct a predicate.

To help maintain consistency, edits are immediately propagated throughout the script. For example, if the user changes the name of a file set, all instances of the set and any members of it are immediately updated. If an operation that produced an output file is deleted, then all subsequent operations that involve that output file are highlighted and the user is informed that deleting the operation can lead to an invalid program.

Users can also select a point in the script and add operations either by copying them from another point in the script or by demonstrating them. Operations can not be drawn from scratch. User defined branches (similar to the Lisp `cond` construct) can also be added to the script by inserting a branch template and constructing the predicates via the predicate menus (Figure 11). Currently, users can add predicates to test for the existence of an object, to compare the properties of two objects, such as their names or date of modification, or to compare the property of an object to a system constant, such as USER or TODAY.

After editing the script, it can be saved. Users indicate parameters by clicking on those objects in the script that represent the actual parameters. For example, clicking on the `papers` folder in Figure 8 indicates that the folder over which the program executes is a parameter to the script. Once saved, a program can be executed by indicating its arguments and selecting the program from the menu of user defined scripts. Programs can also be edited and re-saved, or deleted.

6 INTEGRATING VISUAL REPRESENTATIONS

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.

6.1 Inferring Iterations

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

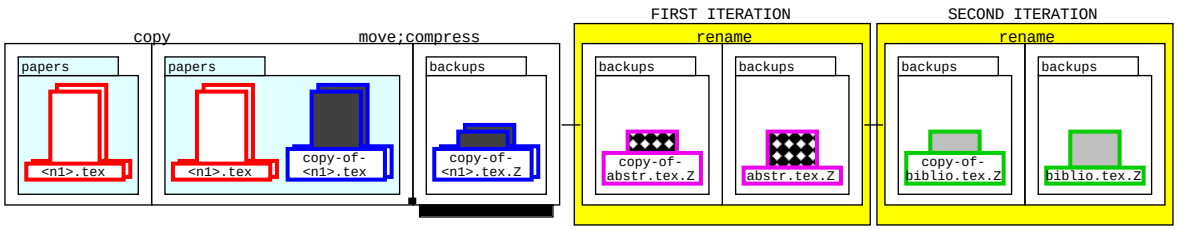

Figure 12: When Pursuit detects a loop, it highlights the operations in the visual program so that users can confirm the inference.

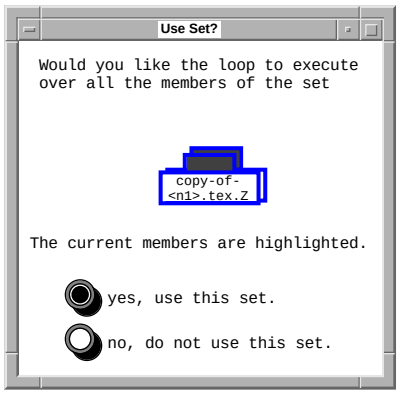

Figure 13: A Pursuit dialog box asking the user to confirm the set to loop over. The icon for the set of files is the same icon found in the third panel of Figure 12. The dialog box appears when the user executes the `rename` operation on two members of the set representing the copied files.

(Figure 12). In this way, users can determine if Pursuit has correctly inferred the loop by looking at the highlighted operations.

Predictive methods like Eager (Cypher, 1991) and Metamouse (Maulsby and Witten, 1990) 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.

6.2 Inferring Sets and Subsets

When Pursuit identifies a set to loop over, it displays a dialog box containing the set's graphical representation (Figure 13). 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 14). In this way, the subset relationship is graphically depicted.

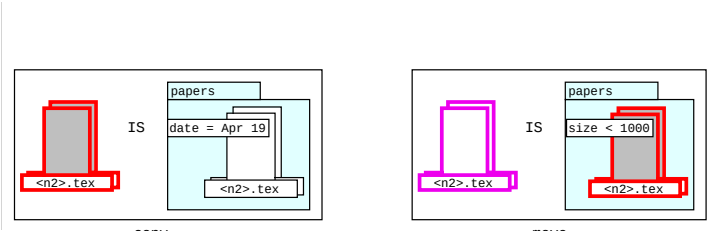

Figure 14: Two Pursuit declarations. The first declaration defines the set icon to represent all the `.tex` files in the `papers` folder that were edited on April 19. The second declaration defines the set icon to be the set containing those files with size less than 1000 bytes in the first file set. Using the first file set icon in the second declaration concisely depicts the subset relationship.

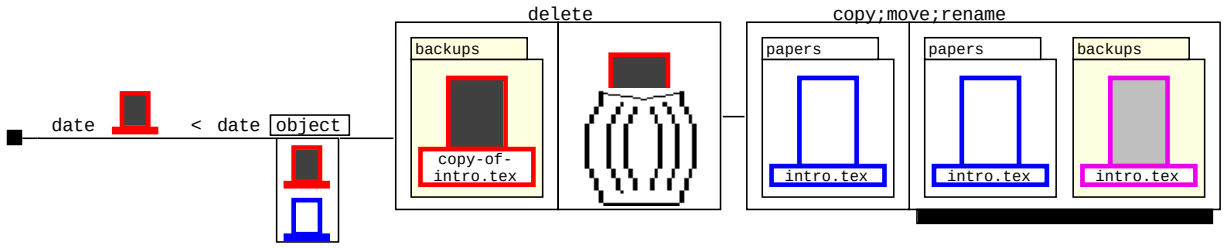

Figure 15: The user is adding a branch before the demonstrated operations. To construct the predicate, the user selected a predicate template and directly edits it using the pull down menus in the template. The menus for objects in the template contain miniature representations of the objects' data icons.

6.3 Editing Programs

Pursuit also 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 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 15). 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.

Dimension	Informal Definition
Viscosity	resistance to change
Hidden Dependencies	important links between entities are not visible
Visibility	ability to view components easily
Diffuseness/Terseness	succinctness of language
Imposed Look-Ahead	constraints on the order of doing things
Closeness of Mapping	closeness of representation to domain
Progressive Evaluation	effort required to meet a goal
Hard Mental Operations	operations that place a high demand on working memory
Secondary Notation	information in means other than program syntax
Abstraction Gradient	types and availability of abstraction mechanisms
Role-Expressiveness	the purpose of a program component is readily inferred
Consistency	similar semantics are expressed in similar syntactic forms

Figure 16: The 12 cognitive dimensions identified by Green and Petre.

7 COGNITIVE DIMENSIONS OF PURSUIT

Cognitive dimensions (Green, 1989 and 1991) of an information artifact provide a framework for a broad-brush assessment of a system's form and structure. By analyzing a system along each dimension, the framework provides insight into the cognitively important aspects of the system's notation and interaction style, and could reveal usability issues that may have been overlooked. In this section, we briefly review the cognitive dimension framework. We then consider Pursuit in light of these dimensions to see how far it is from a region of the design space suitable for its intended purposes. In some cases it scores well; in other cases it scores badly, necessitating changes to the system's design.

In understanding the cognitive dimension framework, it is important to understand that any programming system is composed of its notational structure and its support environment. By notation we mean the the symbols that the user sees and manipulates. By environment we mean the mechanism available to manipulate these symbols. Cognitive dimensions apply to the entire system because the way the user interacts with the system is determined by both the notation and environment for manipulating that notation. One aspect of Pursuit that makes it an interesting case study for cognitive dimensions is that unlike all previous systems studied in which programs are statically defined with a text or visual language editor, Pursuit's PBD system and incrementally evolving program provide a highly interactive and dynamic environment. The process of programming, testing and debugging are intertwined.

7.1 The Dimensions

Green and Petre (in preparation) list 12 cognitive dimensions (see Figure 16) and apply them to a set of contrasting programming languages. In this section we briefly review them for readers not familiar with them. For a more formal discussion of each, the reader is referred to Green (1989 and 1991).

1. Viscosity – how resistant the language is to change. This dimension measures the effects of changing a part of a program. In some systems, changing a part of the program necessitates several other changes in the program. Viscosity measures how difficult these additional changes are to make.

2. Hidden Dependencies – important information links between entities are not visible. Very often the value of a data object depends on the value of another object. Thus changing the value of one object affects the other object and can cause problems if the dependency is not evident.

3. Visibility or Side-by-side-ability – how readily required information can be accessed and whether related components can be viewed simultaneously. Very often users will want to examine different parts of a program at the same time.

4. Diffuseness/Terseness – how succinctly the language can express operations.

5. Order constraints and Imposed Look-Ahead – constraints on the ordering of operations that could force the user to commit too early to a program plan. If there are too many constraints, then users may be forced to make certain choices too soon, locking them into a path that may not be the best or most efficient one.

6. Hard mental operations – operations that have a high mental workload. For example, operations that require users to keep track of hidden information, to compose several operations together, etc. This makes using and understanding the language more difficult.

7. Secondary notation – extra information about the program that is contained in means other than its syntax, such as indenting, name choices, etc. This dimension also includes comments or annotations the user may add to the program. Good secondary notation make understanding programs easier.

8. Abstraction Level – the extent to which abstraction in supported in the system, both at the level of data and operations. This also includes the ability to create subroutines and encapsulate code fragments. Abstraction can help reduce viscosity, increase comprehensibility and help protect from errors.

9. Closeness of Mapping – how close the syntax of the language reflects its domain. A close mapping would enable users to transfer knowledge between the two domains.

10. Gratification Speed and Progressive Evaluation – effort required to meet a goal. This includes the user's ability to execute partially complete programs in order to evaluate how close the program is to the desired solution. This dimension is important particularly for novices, since they often need to evaluate their problem solving progress more frequently than experts.

11. Role-expressiveness – how evident the purpose of a program component is from looking at it. This aids users in recognizing and understanding program code.

12. Consistency – the language express similar semantics in similar syntactic forms. This helps with understanding programs and transferring knowledge between similar syntactic structures.

7.2 How Pursuit Measures Up

There are no hard and fast rules for applying cognitive dimensions. This section represents a compilation of the authors' interpretation and application of the dimensions to Pursuit. In analyzing Pursuit along cognitive dimensions, we found several cognitive pitfalls for which we provide suggestions. Since any notation by itself is never absolutely good, but good only in relation to certain tasks it is important to keep in mind while reading this section that the system was analyzed in

light of the expected user group (non-programmers) and the tasks that these users are expected to do (repetitive programs that execute over a set of related objects at the desktop level).

7.2.1 Viscosity

The Pursuit visual editor makes it relatively simple to cut, copy and paste operations in the program; to add branch and conditional constructs; to add and delete parameters; and to modify the attributes of objects. However, because the main method of program specification is by demonstration, to add new operations to a program users are forced to place "stop points" and re-execute the program on another piece of data. This can place a heavy burden on users who have to place the stop points in the correct position and insure that the state of the desktop is such that the program will execute and follow the desired path. A way to avoid this problem would be to expand the visual editor to include a method to construct an operation's representation. For example, the editor could contain a menu of all system operations. The user can select a particular operation, have a template of its prologue and epilogue appear, and edit the template to contain the correct data objects.

7.2.2 Hidden Dependencies

There are two dependencies in Pursuit: between data objects and between operations. Data objects dependencies define a relationship between two objects based on some shared attribute, which is most often the objects' names. For example, in the program in Figure 8 the output file and the predicate file (*i.e.* the files named `copy-of-<n1>.tex`) both are derived from (depend on) the loop input file (named `<n1>.tex`). Operation dependencies define a relationship between two operations based on some shared data object. For example, both the `move` and the `compress` operations in the program of Figure 5 depend on the `copy` operation.

Both these dependencies can lead to problems when editing the script, since editing the dependency causing objects or operations can affect the dependent objects or operations. For example, suppose the user changes the name attribute of the set of files in Figure 8 to be all `.mss` files. The loop, output and predicate files must also be updated. Similarly, if the user deletes the `copy` operation in Figure 5, then the `move` and `compress` operations would become invalid.

To avoid problems caused by these dependencies, Pursuit contains two features. The first is the automatic propagation of attribute edits. Whenever the user edits an object, all dependent objects are automatically updated. The second feature is automatic notification of invalid operations. Whenever the user deletes a dependency causing operation, Pursuit highlights the dependent operations in the visual script and asks if these operations should also be deleted so that users can see how deleting a single operation affects the entire program.

While these features address the problems of editing dependency causing objects and operations, they do not make users aware of these dependencies until an editing action is taken. A mechanism that shows the dependencies (for example, by highlighting the dependent objects or operations) of an object or operation could help users see the possible effects of their actions before they do anything and could decrease mental load when programming. Consider, for example, the current Pursuit editor and the mental burden on the user in the following scenario. Imagine a fairly long script in which the output of a `copy` operation is not used until several operations later, and the distance between the two operations is such that both are not visible in the program window at the same time. In this case, the dependency between the operations is not visible at any one time. Users must rely on their memory to identify the dependency when scrolling through the program. Similarly, if the user changes the name of the output file so it no longer contains the copied file

name string, then the dependency between the output and input files can only be discovered by tracing the output file icon backwards through (possibly several) operation panels. In both these instances, dependency tracking mechanism would simplify the search and memory requirements for the user.

7.2.3 Visibility, Side-by-Side-ablility and Diffuseness

The entire Pursuit program can be viewed in a scrollable window, making all of the program readily accessible. However, the portion a program that can be viewed at any one time is limited to the width of the program window. Space saving heuristics, such as combining several operations into one panel, increase the amount of the program that is visible at any one time, and the ability to reveal the individual operations of a composite panel insures that the entire program can be viewed at the level of granularity of individual operations. In addition, the ability to pop up a data object's property sheet by clicking on any of its icons in the program allows users to view readily information and attributes of the object. Finally, the use of color to uniquely identify an icon makes it easier to identify and locate data objects when scanning a program.

There are many ways to improve visibility. One way is to allow multiple views of the program so that users can simultaneously view semantically related but distant parts. To simplify accessing information about a data object, it would be helpful to be able to display the object's declaration in a separate window. This could reveal dependencies, such as subsets of file sets, that are not evident in a property sheet. Finally, allowing complete program structures, such as loops and paths of a branch, to be collapsed into a single icon as well as adding the ability of users to select groups of panels to be collapsed into icons would increase the portion of the program visible at any one time, and could provide a global overview of a program's structure.

7.2.4 Order Constraints and Imposed Look-ahead

Pursuit imposes certain order constraints on the programmer. A *component* is a sequence of operations that may contain data dependencies. During a single demonstration, a component can only be developed top-down. This is by nature of the demonstration specification method as discussed above. However, between components that contain no data dependencies with each other, there are no constraints.

To remove constraints between components that contain data dependencies, there are two requirements. First, the user must be able to arrange the state of the desktop so that each component can be successfully demonstrated. Second, the editor must be augmented to allow for two different data objects (*i.e.* two icons of different colors) to be made into the same data object. For example, the user can demonstrate moving and compressing a file as a single component. Then the user can demonstrate copying a file as another component. To make the two components into a single program of copying a file and moving and compressing the output, the user needs a way to indicate that the icon representing the input to the `move` operation should be the same as the icon for the `copy` operation. In this way, the user can demonstrate pieces of the program in any order and then paste them together into a correct program without having to be constrained by the ordering of operations. This would make programming in Pursuit more amenable to the "top-down with deviation" programming process exhibited in other end-user programming domains (Visser, 1990; Davies, 1991).

Constraints Imposed by PBD

It is interesting to note that the demonstrational specification technique both decreases and in-

creases the look-ahead necessary to write a program. Demonstrating a program on existing data objects without considering all possible problems the program may encounter decreases look-ahead. The user simply interacts with the system in the usual way. However, if users desires to construct a program so that it "always" works, then, like all programmers, they must be able to consider all possible data conditions that the program may have to deal with. In the PBD model this requires users to arrange the state of the system so that the demonstration will encounter these situations and construct the program accordingly. Such a look-ahead requirement is burdensome.

To decrease this burden, Pursuit incorporates two features: *exit branch exposition* and *incomplete path exposition*. Exit branch exposition allows the user to view all the possible outcomes of a particular operation by clicking on an exit branch in a visual script. For example, clicking on the conditional marker in Figure 8 displays the predicates for the remaining exit branches of the operation. The user can then demonstrate what the program should do in each case. Similarly, incomplete path exposition displays the predicate for a particular exit branch of an operation when that branch is encountered during execution. When an operation of an executing program has an outcome not included in the program, Pursuit displays the program, adds the undemonstrated branch, highlights it, and asks the user what to do: abort the program execution; abort the execution of this data object; or allow the program to be augmented by the user demonstrating the new path the program should take.

Both exit branch exposition and incomplete path exposition decrease the look-ahead requirement imposed by the PBD specification technique because the user is no longer forced to think of all possible paths the program can take. Instead, the user can demonstrate the program in the current state and then explore ways to augment the program by examining each operation's exit branches. Furthermore, the program need not be edited immediately. Instead, the user can do so any time the program is run and a forgotten path is encountered.

Constraints Imposed by Pursuit

In addition to the constraints imposed by the demonstrational specification technique, the Pursuit programming model of sets and set manipulation also imposes some constraints. Consider the example (section 4.1) in which the user copies, moves and compresses all the `.tex` files in a folder. Suppose that one of the files originally copied was `abstr.tex` and that the `backups` folder contains a file with the name `copy-of-abstr.tex.Z`. Then the `compress` operation will fail on that set member. Thus, the users "plan" to manipulate the set of files in order to construct the program was incorrect. To successfully demonstrate this program, the user must examine the state of the system and notice the problem causing file. Then she must demonstrate the program with two examples - one in which the `compress` operation succeeded and one in which it failed - so that Pursuit could infer the explicit loop (similar to the example in section 4.2). This places a large look-ahead constraint on the user, since for very long programs involving multiple operations in multiple folders, the user would have to carefully inspect the system's state, remembering various data states and operation outputs throughout. Such a search would most likely exceed working memory capacity.

A similar problem arises when the user demonstrates a set of operations on a file set and afterwards realizes that the set selection criteria cannot be expressed via the set attributes, but must be explicitly tested for in a user defined branch. Imagine the frustration as the user exclaims "Darn, I should have used only a single file!". That is, to have correctly demonstrated the program, the user would have had to demonstrate it on a single data object, then edit the program to add the branch and wrap it in a loop. This requires that the user completely understand before hand how to express selection criteria.

To address these two problems, Pursuit needs a mechanism to convert a sequence of set operations to an explicit loop containing the operations. The loop's iteration set would be the data set to the original operations. Such a mechanism should automatically infer an explicit loop whenever an operation applied to a set has different outcomes for different set members, and should be available for users to invoke whenever they need to make a sequence of set operations into an explicit loop. In this way, users are less constrained to examine the system state or to fully know how to express certain selection criteria before a demonstration.

7.2.5 Domain Interpretability or Closeness of mapping

In most language designs a close fit is thought desirable, because novices are expected already to know the domain. We believe that in shell programming novices will not know the domain and that it would be better if Pursuit hid some of the more idiosyncratic features. Pursuit is necessarily driven by some domain requirements, such as the various possible outcomes of operations, but it is mildly abstraction-tolerant: in Figure 5 the third panel probably conforms closer to the user's conceptual structure than the multiple operations that it encloses.

7.2.6 Gratification Speed and Progressive Evaluation

In Pursuit, the program is constructed while it is being executed, so that the user can immediately see its effects. Even when editing or revising a program, users add operations by demonstrating actions so that they can see the results of the program. Incomplete programs are easy to execute – indeed programs as they are being constructed are incomplete programs executing. Thus, the programming process provides (almost) immediate means of evaluating progress and seeing results quickly, making for high gratification.

7.2.7 Hard Mental Operations

Since Pursuit is similar to procedural textual languages such as Pascal and Basic, and since it has been shown that in these languages extracting declarative information from dense conditional structures was not easy (Sime, Green and Guest, 1977), we can assume that Pursuit would suffer the same limitation. We note, however, a study that may suggest that Pursuit's conditionals could be easier to abstract information from. Green (1977) showed that professional programmers could abstract declarative information more easily from a Pascal-like textual language when the branches of the conditional were labeled with "predicate" and "not predicate" rather than "predicate" and "else", and when the conditional ended with "end predicate" rather than just "end". Notice that Pursuit branches, either those based on an exit code or those constructed by the user, contain predicates along each path explaining the condition (e.g., exists *file*) that causes the path to be taken. In addition, the end of a Pursuit branch is communicated visually – the user reaches the end of the panel of operations along the branch. Perhaps this supplies the "visual equivalent" of negated predicates and explicit ends that simplified mentally hard operations in the textual domain.

7.2.8 Secondary Notation

In traditional textual languages, which are by design 1-dimensional, layout on a page (the second dimension) is often used to convey meaning and structure in the program. In Pursuit, as in other VPLs, the second dimension is already incorporated (formally) into the language. Other means are needed to maintain perceptual cues. In Pursuit, the use of different graphical structures, such as a

loop box and branch construct, as well as layout, such as paths of a branch being parallel to one another, are ways that some control information is conveyed.

Other perceptual cues, such as the shadows beneath composite panels and the use of color to identify an object or set element throughout the script, are used to convey information on program structure. In addition, the ability of users to collapse a loop into a loop icon or to choose several panels to collapse into a single icon to identify components can be used to relay some overall structural information about the program (similar to subroutine calls in textual languages).

The most common form of secondary notation in programming languages is the use of comments. Currently, Pursuit provides no way to annotate the program. It would be easy to extend the language to allow a panel to be annotated. However, since the real "action" of a program occurs between panels, and since users would most likely want to annotate a program component, some way to add comments at random points in the program is necessary.

7.2.9 Abstraction Level

There are several types of abstractions in Pursuit. Declarations allow for specifying abstract sets of objects. Property sheets also aid abstraction and provide for a built in form of search and replace (e.g., Consider changing a file name. All dependent files are automatically updated). Automatic panel collapsing provides abstraction at the operation level and future extensions will increase this capability. In addition, parameterizing a script and incorporating into other scripts also provides for operation abstraction.

An interesting feature of Pursuit is that abstraction is done both automatically (by the inference mechanism in creating sets, collapsing operations, etc.) and by the user (parameterizing scripts, editing attributes, etc.).

7.2.10 Role Expressiveness

One of Pursuit's goals is to maximize role expressiveness. The Pursuit visual language was designed so that the graphical properties of a desktop object map to graphical properties of its iconic representation in a visual script. For example, the icons for files and folders in the visual script reflect the icon identifiers for files and folders in the interface (compare Figure 5 and Figure 6). The whole basis for operation representations is depicting the changes operations cause to objects. Theoretically, then, Pursuit is role expressive.

7.2.11 Consistency

We found two inconsistencies in the syntax/semantics relationship in Pursuit. First, an object's name is really an attribute, but is not represented graphically like other attributes such as date. Instead, it has its own unique location and representation. A similar objection was raised to the representation of an object's location attribute (*i.e.* in a folder object, not as a graphical attribute). There are two reasons to support this inconsistency. First, the name of an object and its location serve as both attributes *and* graphical properties that operations can change. Hence, they each serve two roles and should have a representation distinct from attributes only. Second, the location of an object is something that we believe is easier to identify via a good graphical representation (*i.e.* one that is more role expressive) than through an "artificial" graphical one like attribute boxes.

The second inconsistency is how properties of objects are depicted one way in attributes, and another way for user defined predicates. For example, compare how the date attribute is represented in Figure 3A and the date properties is extracted in Figure 11. Both representations refer to the

date property of an object, yet both are represented differently. This inconsistency is defended by the increased understandability the designers hope is gained.

Only user testing can support whether or not these inconsistencies increase or decrease program understandability.

8 STATUS AND FUTURE WORK

A prototype of Pursuit has been implemented using Garnet (Myers, et al, 1990) and is currently operational (Modugno and Myers, 1994). This prototype has been used to evaluate the Pursuit design along the cognitive dimensions. Doing so has already revealed several ways to improve the design, as discussed in the previous section. Using the prototype, we have also done some informal user studies, which have provided important feedback to improve the system. For example, Pursuit initially contained a heuristic that sometimes eliminated the prologue of the first operation of a program. Since several people had difficulty understanding scripts in which this heuristic was applied, we eliminated it. Further work is needed to refine the heuristics for generating attributes and operation panels and to provide ways for making programs more concise.

We are also exploring other ways of using visual representations as feedback, such as with animation. For example, consider the dialog box in Figure 13. 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, user studies are planned to evaluate the visual language itself as well as the entire Pursuit system to determine how well it helps users automate tasks. In these studies, users will construct programs for some tasks using Pursuit. These results will be compared with users doing the same task in the Pursuit visual shell but whose program representation language is an "English-like" textual language similar to the one found in SmallStar. This will help us evaluate whether or not the visual representation really does help simplify the programming process. We also plan to compare the recognizability of programs in both the Pursuit visual language and the "English-like" textual language in the same way that recognizability was evaluated for dataflow languages and textual languages (Green and Petre, 1992; Green, Petre and Bellamy, 1991).

9 CONCLUSION

Pursuit is a visual shell that is designed to provide much of the common programming power currently missing in visual shells in a way that is consistent with the direct manipulation paradigm. By combining the techniques of Programming by Demonstration with an editable, visual representation of programs, users can create abstract programs containing variables, loops, and conditionals. The goal is to enable users to access the underlying power of the computer by interacting with it as much as possible in the way they normally do - by executing real actions on real data objects - thus reducing the gap between between users' mental model and the difficult task of programming.

Analyzing the system along cognitive dimensions provided insight into the strong points of the system and suggested ways to improve some of its weaknesses. In particular, it revealed hitherto unnoticed weaknesses in viscosity, hidden dependencies and (quite unexpectedly to the designer) imposed look-ahead. The results of the planned user studies will further evaluate how close Pursuit comes to its intended goals.

10 ACKNOWLEDGEMENTS

The authors thank T.R.G. Green for his help and guidance with the cognitive dimension analysis. We also thank Mitch Dsouza, Jade Goldstein, David Hendrey, Bonnie John, David Kosbie, David Kurlander, James Landay and Marian Petre for enriching comments on this work.

11 REFERENCES

D. Ackermann and J. Stelovsky (1986). 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 37-52.

Kjell Borg (1990). IShell: A Visual UNIX Shell. In *Proceedings of CHI '90*, pages 201–207.

Keys Botzum (in progress). An Empirical Study of Shell Programs. Technical Report in progress, Bell Communications Research.

Wayne Citrin (1991). Visualization-Based Visual Programming. Technical Report CU-CS-535-91, Unversity of Colorado, Boulder, Colorado.

Allen Cypher (1991). EAGER: Programming Repetitive Tasks by Example. In *Proceedings of CHI '91*, pages 33–40.

Allen Cypher (1993). *Watch What I Do: Programming by Demonstration.* The MIT Press, Cambridge, MA, 1993.

S. P. Davies (1991). Characterising the Program Design Activity: Neither Strictly Top-Down Nor Globally Opportunistic. *Behaviour and Information Technology*, 10(3):173–190.

Stephanie M. Doane, James W. Pellegrino, and Roberta L. Klatzky (1990). Expertise in a Computer Operation System: Conceptualization and Performance. *Human-Computer Interaction*, 5:267–304.

William Finzer and Laura Gould (1984). Programming by Rehearsal. *Byte Magazine*, 9(6):187–210.

George W. Furnas (1991). New Graphical Reasoning Models for Understanding Graphical Interfaces. In *Proceedings of CHI '91*, pages 71–78.

T.R.G. Green (1977). Conditional Program Statements and Their Comprehensibility to Professional Programers. J. Occupational Psychology, 50:93–109.

T.R.G. Green (1989). Cognitive Dimensions of Notations. In A. Sutchiff and L. Macaulay, editors, *People and Computers V*, pages 443–460. Cambridge University Press.

T.R.G. Green (1991). Describing Information Artifacts with Cognitive Dimensions and Structure Maps. In D. Diaper and N. Hammonds, editors, *People and Computers VI*, pages 297–316. Cambridge University Press.

T.R.G Green and M. Petre (1992). When Visual Programs are Harder to Read than Textual Programs. In G.C. van der Verr, M.J. Tauber, S. Bagnarola, and M. Antavolits, editors, *Human-Computer Interaction: Tasks and Organisation* (Proceedings 6th European Conference on Cognitive Ergonomics).

T.R.G. Green, M. Petre, and R.K.E. Bellamy (1991). Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the 'Match-Mismatch' Conjecture. In *Empirical Studies of Programmers: Fourth Workshop.*

P.E. Haeberli (1988). ConMan: A Visual Programming Language for Interactive Graphics. In *ACM SIGGRAPH*, pages 103–111.

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

Tyson R. Henry and Scott E. Hudson (1988). Squish: A graphical shell for unix. In *Graphics Interface*, pages 43–49.

Edwin L. Hutchins, James D. Hollan and Donald A. Norman (1986). Direct Manipulation Interfaces. In D. Norman and S. Draper, editors, *User Centered System Design*, pages 87-124.

Branka Jovanovic and James D. Foley (1986). 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.

David Kurlander and Steven Feiner (1988). Editable Graphical Histories. In *Workshop on Visual Languages*, pages 127–134, Pittsburgh, PA 15213.

Clayton Lewis and Gary M. Olson (1987). Can Principles of Cognition Lower the Barriers to Programming? In *Empirical Studies of Programmers: Second Workshop*, pages 248–263.

Henry Lieberman (1982). Constructing Graphical User Interfaces By Example. In *Graphics Interface '82*, pages 295–302, Toronto, Ontario, Canada.

Henry Lieberman (1993). Mondrian: A Teachable Graphical Editor. In *Proceedings of InterCHI '93*, page 144.

David L. Maulsby and Ian H. Witten (1989). Inducing Programs in a Direct-Manipulation Environment. In *Proceedings of CHI '89*, pages 57–62, Austin, Tx.

Francesmary Modugno (in progress). *Pursuit: Adding Programming in the Interface.* PhD thesis, Carnegie Mellon University, expected December 1994.

Francesmary Modugno and Brad A. Myers (1994). Pursuit: Graphically Representing Programs in a Demonstrational Visual Shell. To appear in *Proceedings of CHI '94.*

Brad A. Myers (1988). *Creating User Interfaces by Demonstration.* Academic Press, Boston, Massachusetts.

Brad A. Myers (1992). Demonstrational Interfaces: A Step Beyond Direct Manipulation. *IEEE Computer*, 25(8):61–73.

Brad A. Myers et al (1990). Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. *IEEE Computer*, 23(11):71–85.

M.E. Sime and T.R.G. Green and D.J. Guest (1977). Scope Markings in Computer Conditionals – A Psychological Evaluation, *Int. J Man-Machine Studies*, 9:107–118.

Ben Shneiderman (1983). Direct Manipulation: A Step Beyond Programming Languages. *Computer*, 16(8):57–69.

W. Visser (1990). More or Less Following a Plan During Design: Opportunistic Deviations in Specification. *Int. J. Man-Machine Studies*, 33(3):247–278.