In: G. Cockton, S. W. Draper and G. R. S. Weir (Eds.) People and Computers IX, Proc. of HCI'94, Glasgow. Cambridge: Cambridge University Press, 1994, pp 91-108.

Visual Programming in a Visual Domain: A Case Study of Cognitive Dimensions

Francesmary Modugno[1] T. R. G. Green[2] Brad A. Myers[1]

[1]Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213
{fmm,bam}@cs.cmu.edu

[2]MRC Applied Psychology Unit
15 Chaucer Road
Cambridge CB2 2EF, UK
thomas.green@mrc-apu.cam.ac.uk

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 analyzed the language and its editor along cognitive dimensions. The assessment provided insight into both strengths and weaknesses of the system, prompting a number of design changes. This demonstrates how useful such an analysis can be.

KEYWORDS: Cognitive Dimensions, End-User Programming, Programming by Demonstration, Visual Language, Visual Shell, Pursuit.

1. Introduction

A visual shell (or desktop) is a direct manipulation interface to a file system. Examples include the Apple Macintosh desktop and the Xerox Star. Although such systems are easy to use, most do not support end-user programming. Pursuit is a visual shell aimed at providing 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 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 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.

There are three properties we believe will make it possible for novices and non-programmers to learn to construct and recognize Pursuit programs. First, programs are specified by executing real actions on real data. The Pursuit PBD system 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. Finally, 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 in Pursuit.

This paper reports our experience using cognitive dimensions (Green, 1989 and 1991) to determine how well Pursuit meets its design criteria, and gives a designer's perspective on how useful such an analysis is. The assessment provided support for several design decisions and revealed potential weak areas of the system, prompting a number of system changes.

2. RELATED WORK ON VISUAL SHELLS

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.*, Haeberli, 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

The goal of Pursuit is to add end-user programming into a visual shell in a consistent way. This section briefly reviews some motivation and history for our 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. 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. 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

[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.

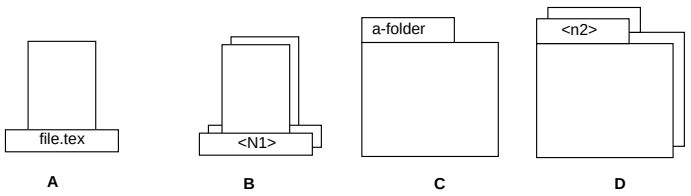


Figure 1: The main data types in Pursuit: a) a file; b) a set of files; c) a folder; d) a set of folders.

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.

4. PURSUIT'S VISUAL REPRESENTATION LANGUAGE

Visual shells are easy to use because of the constantly visible, concrete, familiar representations of data objects and the illusion of concretely manipulating these objects. 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.

The Pursuit visual language[2] combines elements of the comic strip metaphor (Kurlander and Feiner, 1988) and the visual production system (Furnas, 1991). 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 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 programs, 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 following examples.

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

[2]Our current focus is not on 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. However, we recognize that the visual presentation will necessarily influence the acceptance of our language, and thus deem it important to explore that avenue. Currently, however, we do not have the resources to invest in graphic design.

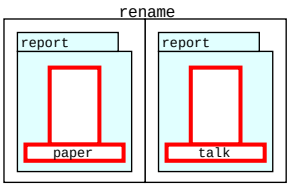
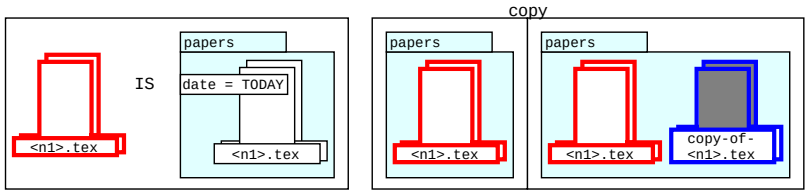


Figure 2: The representation of the operation **rename paper talk**. The first panel shows the icon representing the file **paper** located in the **report** folder before it is renamed. The second panel shows the same file after the **rename** operation. Notice that the icon's name has changed. This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



A **B**

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.

on a particular set of files. During the demonstration, the underlying PBD system constructs a program.

Figures 3-5 show the developing program *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 program 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 properties of data objects or to fix incorrect inferences. 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.

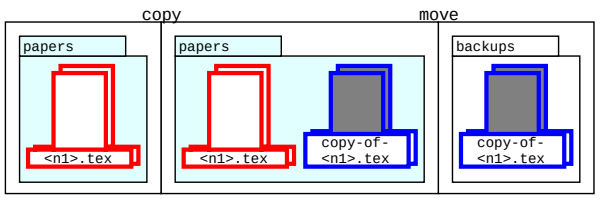


Figure 4: After the user drags (moves) the copies to the **backups** folder, the third panel appears. Notice that in the program the set of copies icon has moved from the **papers** folder to the **backups** folder, reflecting the changes the user has seen in the actual interface when the real copies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

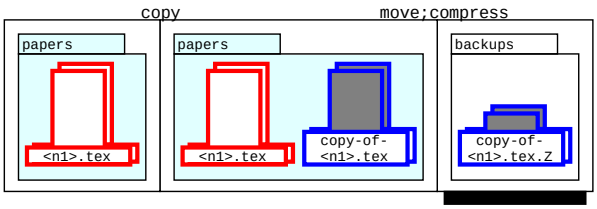


Figure 5: The completed program. The `compress` operation is represented by the difference in the height and the name of the icon for the copies in the second and third panels. This difference is similar to the change in appearance of the icons for the real files that the user would see in the actual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

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 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. Clicking on it reveals 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 program, Pursuit assigns it a unique color. Although an icon's appearance may change throughout the program, 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.

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 6) 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

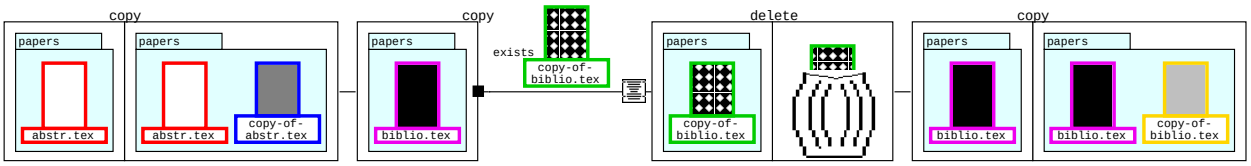


Figure 6: The two cases used to demonstrate a conditional program. The first two panels show a successful execution of the `copy` operation, demonstrated on the `abstr.tex` file. The remainder of the program shows the unsuccessful execution, demonstrated on the `biblio.tex` file, and corrective actions the user wants the program to take in that case. Between the second and third panels is a gap, indicating that the user is manipulating a different data object. 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).

"exists" plus a named file icon.

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 6. It then finishes executing the loop and updates the program. The updated program (shown in Figure 7) is an example of an explicit loop containing an explicit conditional.

The panel below the visual declaration (FOREACH .. IN ..) 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 on 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.

5. THE PURSUIT EDITOR

Pursuit contains a visual language editor so that users can 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 8). 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.

To help maintain consistency, edits are immediately propagated throughout the program. 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

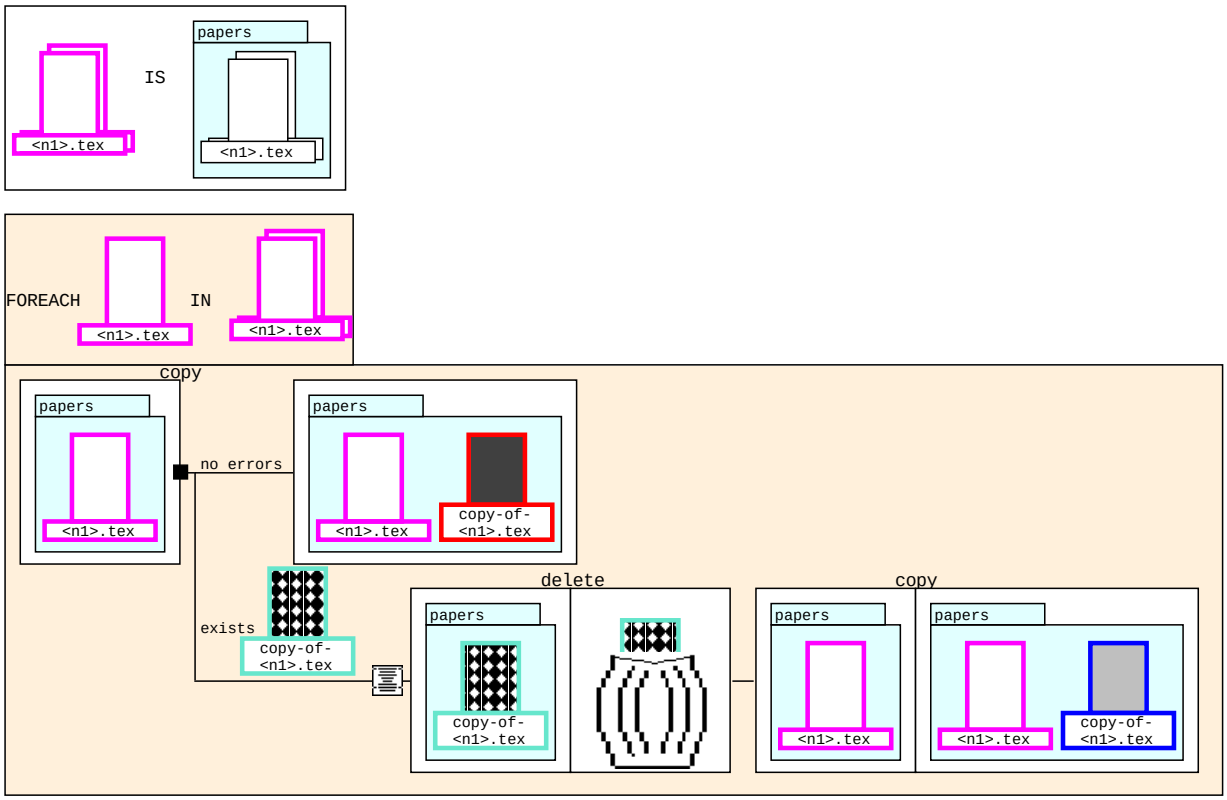


Figure 7: The Pursuit program that copies each `*.tex` file in the `papers` folder. If the `copy` operation fails because of the existence of a file with the output file name, the program deletes that old output file, and re-executes the `copy` operation. Users can see the other possible outcomes of the `copy` operation by clicking on the conditional marker.

operation can lead to an invalid program.

Users can also select a point in the program and add operations either by copying them from another point in the program 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 program by inserting a branch template and constructing the predicates via the predicate menus (Figure 9). The types of predicates users can construct represent some of the common predicates found in our informal study of shell scripts, as well as more general predicates we as designers and users felt were important to have in a language.

After editing the program, it can be saved. Users indicate parameters by clicking on those objects in the program that represent the actual parameters. For example, clicking on the `papers` folder in Figure 7 indicates that the folder over which the program executes is a parameter to the program. Once saved, a program can be executed by indicating its arguments and selecting the program from the menu of user defined programs. Programs can also be edited and re-saved, or deleted.

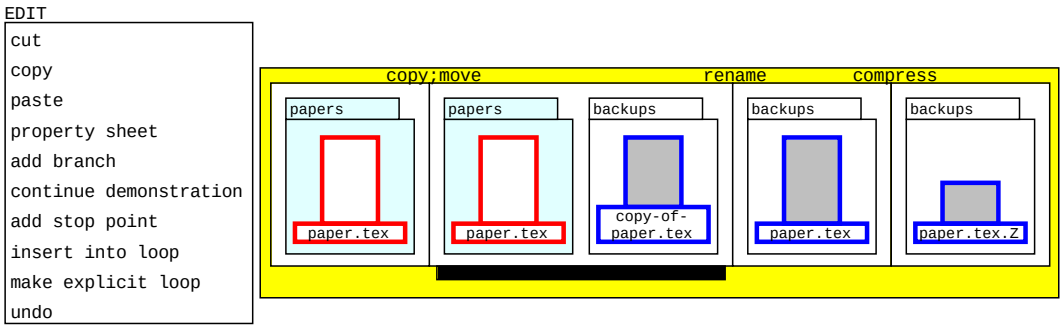


Figure 8: The user has selected a sequence of panels (indicated here by the outer gray rectangle) to wrap in a loop.

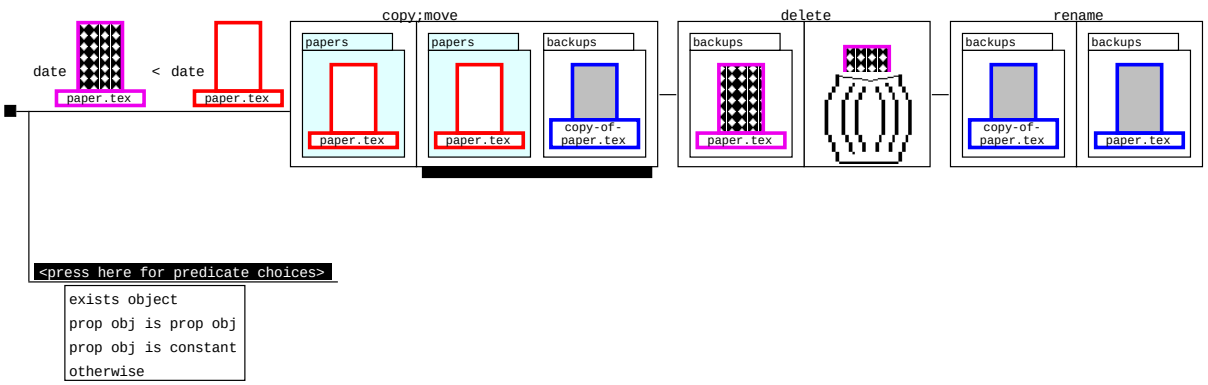


Figure 9: Adding a user defined branch. The upper branch shows the path to take when the file `paper.tex` in the `backups` folder is older than the file `paper.tex` in the `paper` folder. The user is adding another predicate to the branch construct by selecting from the menu of predicate templates. The menu choices provide templates for the user to further edit in order to construct a predicate.

6. 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 overlooked usability issues.

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 processes of programming, testing and debugging are intertwined.

Green and Petre (submitted) list 12 cognitive dimensions (see Figure 10) and apply them to a set of contrasting programming languages. We considered Pursuit in light of these dimensions to see how far it was from a region of the design space suitable for its intended purposes. In some cases it scored well; in other cases it scored badly, prompting changes to the system's design.

6.1. History of the Analysis

Before presenting our analysis, it is beneficial to explain how the analysis was done. The first author (and designer/implementer of Pursuit) began by reading Green (1989 and 1991) and an early version of Green and Petre (submitted). She then spent a day thinking about how each dimension applied to some typical programs users might create with Pursuit. Finally, she spent the next few days writing an early version this paper, and sent it via electronic mail to the second author.

The second author and his colleague David Hendry, both experts in cognitive dimension, perused paper descriptions of Pursuit and the Pursuit video (Modugno and Myers, 1994). After watching the video continuously through, they ran it again, stopping frequently to ask questions such as "How can the user change that bit of program?" and "How does the user know what cases will be needed to set up a demonstration?". They then went over the list of dimensions, asking for each one how it applied to the programming exercise they had watched. Because they were already experienced in this activity, they needed very little time over and above the time taken to read the papers and watch the video.

The second author agreed with the analysis done by the first author, and added only the closeness of mapping analysis. Our experience shows that cognitive dimensions are a powerful, easy to use tool. The first author was able to do a detailed assessment in less than a day, without ever having used the technique before. Furthermore, the technique provides both experts (the second author) and novices (the first author) the ability to examine an artifact, and proceed quickly to a high level (e-mail!) discussion of it. We attribute this to the compact shared vocabulary of cognitive dimensions not found in other evaluation techniques. Finally, the technique provided several insights

Dimension	Informal Definition
Viscosity	resistance to change
Hidden Dependencies	important links between entities are not visible
Visibility and Side-by-Side-ability	ability to view components easily
Diffuseness/Terseness	succinctness of language
Imposed Guess-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	extra 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 10: The 12 cognitive dimensions identified by Green and Petre.

to the designer, even though she had been working with the system for over two years!

6.2. The Analysis

To save space, we present only the highlights of our analysis, practically unchanged from the original version.

6.2.1. 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 could select a particular operation, have a template of its prologue and epilogue appear, and edit the template to contain the correct data objects.

6.2.2. Hidden Dependencies

There are two dependencies in Pursuit: between data objects and between operations. Data object 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 7 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, the `move` and `compress` operations in Figure 5 depend on the `copy` operation.

Both these dependencies can lead to problems when editing programs, 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 7 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 dependency problems, 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 program 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 program 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 cases, dependency tracking mechanism would simplify the user's search and memory requirements.

6.2.3. Visibility, Side-by-Side-ability and Diffuseness/Terseness

The entire Pursuit program can be viewed in a scrollable window, making all of the program readily accessible. However, the portion of 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.

6.2.4. Imposed Guess-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. However, between components that contain no 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 the 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 is 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).

It is interesting to note that the demonstrational specification technique can both decrease and increase 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 the user 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, the state of the system to be arranged so that the demonstration will encounter these situations. 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 users to view all possible outcomes of a particular operation by clicking on a conditional marker. For example, clicking on the conditional marker in Figure 7 displays the predicates for the remaining exit branches of the operation. The user then demonstrates what the program should do in each case. 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.

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 beforehand 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 for 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.

6.2.5. 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. For example, the user probably believes that when compressing a file the output is still the same file, only its contents are compressed. In reality, compressing a file produces an entirely different file whose contents are the compressed form of the original file's contents, and removes the original file from the system.

6.2.6. 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. RELATED WORK ON EVALUATION TECHNIQUES

There has been much work on evaluating designs for interfaces; Green and Petre (submitted) compare cognitive dimensions (CDs) with several other techniques. The nearest in spirit to CDs appears to be Heuristic Evaluation (HE) (Nielsen and Molich, 1990), which, like CDs, attempts to substitute a very small number of general principles for the huge volumes of detailed guidelines in common use. The 9 heuristics of HE, such as "use simple and natural dialogue," "speak the user's language," "minimize user memory load," etc., are to be applied by a small team, carefully considering all aspects of an interface and searching for problems.

While both HE and CDs use the same processes (sit and think) there are many notable differences: (i) The focus of HE is on *interacting* with an application, whereas CDs focus on users *building an information structure* (*e.g.*, a program, a musical score, or even a long-distance telephone number). (ii) HE is best used by HCI specialists, but CDs are designed to make sense to ordinary "choosers and users." As noted above, in this instance CDs were usefully applied by the first author, a computer scientist with no specialist HCI background. (iii) Like other evaluation techniques, HE focuses on surface details, thus restricting its applicability; In contrast, CDs can be used to assess the information structure of a design long before any coding takes place. (iv) Heuristics like "be consistent" are likely to be useful only at a low level (e.g. screens A and B are different), whereas "look for imposed guess-ahead" deals with structural problems of a very different order. (v) But the most telling difference is that the 9 heuristics of HE are simply based on experience, whereas the CDs are based on a model of cognitive processes in design; in consequence, (a) the CDs address problems that are overlooked by the HE – e.g. the need to rebuild information structures is a part of the design process, now recognized to be at least partly opportunistic; (b) the terms of the CDs can in principle be grounded in analyses of information structure or in psychological theories of, say, parsing; and (c) the CDs address the problems of trade-offs between choices, such as what happens if viscosity is reduced by adding abstractions.

8. STATUS AND FUTURE WORK

A prototype of Pursuit has been implemented using Garnet (Myers, et al, 1990). 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. 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 program 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.

In addition, 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. This would enable us to evaluate how easy it will be to write a program by modifying an existing one, to identify a program long after it is written, or to share programs between users. Indeed, if a system like Pursuit becomes popular, one can imagine that few programs will be written from scratch, but will instead be copied and modified from existing programs. The ability to recognize and tailor programs would thus become very important.

9. CONCLUSION

Pursuit is a visual shell 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 the way they normally do - by executing real actions on real data objects - thus reducing the "programming" skills they need to learn.

The aim of the cognitive dimensions framework is to provide a vocabulary for discussion to assist designers and evaluators. It readily revealed potential improvements to Pursuit, and so in the present case it certainly achieved its goal. In particular, it revealed unnoticed weaknesses in viscosity, hidden dependencies and (quite unexpectedly to the designer) imposed guess-ahead; a convincing demonstration of how useful the framework can be.

10. ACKNOWLEDGEMENTS

The authors thank Mitchum D'Souza, Bill Hefley, David Hendry, David Kosbie, David Kurlander, James Landay and Marian Petre for enriching comments on this work. We also thank the reviewers for their insightful comments. The first author is supported by NSF grant IRI-98020089 and by a grant from AAUW.

11. REFERENCES

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

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, (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, (submitted). Cognitive Dimensions as Discussion Tools for Programming Language Design.

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.

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 and Brad A. Myers, (1994). Pursuit: Graphically Representing Programs in a Demonstrational Visual Shell. *Proceedings of CHI '94 Video Program.*

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.

Jacob Nielsen and R. Molich (1990). Heuristic Evaluation of User Interfaces. In *Proceedings of CHI '90*, pages 249–256.

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