EVALUATING PROGRAM REPRESENTATION IN A DEMONSTRATIONAL VISUAL SHELL

Francesmary Modugno *Albert T. Corbett* *Brad A. Myers*
Carnegie Mellon University
Pittsburgh, PA 15213
fmm@cs.cmu.edu ac21@andrew.cmu.edu bam@cs.cmu.edu

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

ABSTRACT
For Programming by Demonstration (PBD) systems to reach their full potential, some way of representing programs is necessary so that programs can be viewed, edited and shared. How to represent programs, especially in a way that is easy for non-programmers to understand, is an open question. We designed and implemented two representation languages for a PBD desktop similar to the Apple Macintosh Finder. The first language is based on the comic-strip metaphor and graphically depicts the *effects* of a program on data. The second language is text-based and describes the *actions* of a program. A user study revealed that both languages enabled non-programmers generate and comprehend programs containing loops, variables and conditionals. Moreover, the study revealed that the comic-strip language doubled users' abilities to accurately generate programs. Trends suggest that the same language was also easier for users to comprehend. These findings suggest that it is possible for a PBD system to enable non-programmers to construct programs and that the form of the representation can impact the PBD system's effectiveness.

1. INTRODUCTION AND MOTIVATION

A visual shell, such as the Macintosh Finder, is a direct manipulation interface to a file system. Files and directories are represented as icons and the user specifies operations by directly manipulating the icons. For non-programmers in particular, the illusion of manipulating data rather than issuing textual commands makes interacting with computers more concrete and therefore simpler. However, the cost of this simplicity has been the decreased power available to the end user. Most visual shells provide no mechanism for users to automate even the simplest repetitive tasks. When programming is introduced, for example in macro languages like QuicKeys, the "conceptual simplicity" that makes visual shells popular is often sacrificed: programming is done off-line in a textual programming language. Users must develop two very different bodies of knowledge: concrete, visual notations to interact with the system, and abstract, textual notations to program it.

Some visual shells (e.g., Haeberli, 1988; Borg, 1990) have attempted to overcome this problem by providing a graphical programming language. These languages are often based on the data-flow model, with icons representing utilities and connecting lines representing data paths. Unfortunately, they contain no way to depict abstractions or control structures. Furthermore, programming is done off-line by wiring icons together, which is different from the way users normally interact with the system. Thus, while "visual" like the desktop, these languages still require that users learn a new and very different programming language. More importantly, users must learn how to program from scratch since they cannot transfer skills and knowledge of the interface to the programming task.

The Pursuit visual shell is exploring ways to provide end-user programming capabilities, especially to non-programmers, *in a way that is consistent with the direct manipulation paradigm*. Pursuit contains a Programming by Demonstration (PBD) system (Cypher, 1993) to construct file manipulation programs. In PBD environments, the system infers a program as the user executes actions on real data (Myers, 1992). PBD systems have shown promise in enabling non-programmers to automate tasks (e.g., Cypher, 1991; Halbert, 1984; Maulsby and Witten, 1989), but there is one well-known shortcoming: how to represent the resulting program to end users. Without a representation there are two problems. First, inferential ambiguities often arise during program generation and PBD systems typically resolve them by querying the user (Myers, 1988; Maulsby and Witten, 1989) or highlighting the expected user action (Cypher, 1991). However, users sometimes find these interactions confusing and often simply select the default option (Cypher, 1991). Second, users are unable to recall what old programs do, view programs written by others, or make modifications to existing programs. Even simple changes require rewriting a program from scratch. Without a static program representation, PBD systems will never realize their full potential.

To address this problem with PBD systems, we are investigating ways to represent the evolving program *while the user is demonstrating it*. In this way, users know immediately what the system has inferred (by observing the growing program representation) and can interactively and incrementally learn the syntax and semantics of the representation language. By allowing the program to be edited and saved, users can have an artifact to later examine, edit and share.

In this paper, we present two graphical program representation languages and describe an empirical evaluation of them. The purpose of the study was twofold. First, we wished to see if Pursuit met its goal of enabling non-programmers to create programs containing loops, variables and conditionals. Second, we wanted to see what effect, if any, the representation language had on a user's ability to generate and comprehend programs in the PBD system.

Each of the representation languages incorporates data icons from the visual shell, enabling the user to transfer knowledge of the desktop icons to the representation language when constructing, viewing and editing a program. The first language employs a comic strip metaphor (Kurlander and Feiner, 1988) for operations and graphical representations of control structure. The second language, similar to SmallStar (Halbert, 1984), is essentially textual with a conventional verb/argument structure for operations and keywords for control constructs.

Although Pursuit is the first Programming by Demonstration system to explore different forms of program representations, previous studies have compared textual and graphical program representations to determine whether different representations might prove useful for different audiences and different situation. Some have found visual representations to be better than textual representations (e.g., Cunniff and Taylor, 1987), some have found them to be worse (e.g., Green, Petre and Bellamy, 1991), and some have found them to be the same (e.g., Moher, et al., 1993). What we can conclude from these studies is that neither text nor graphics is in itself inherently superior; rather, the extent to which a particular notation supports users in their tasks depends on the context in which the language is employed[1]. Further research is needed to determine which characteristics of a representation language make it suitable for a particular type of task in a particular domain.

Our work differs from these studies in that we examine two equivalent *graphical* languages that differ in their use of text and graphics: one language emphasizes graphics and uses text to support its syntax, while the other language emphasizes text and uses graphics to enhance its syntax. In essence, our work is exploring two different points in the design space of potential graphical languages for this particular domain. One language attempts to tap into the more graphical nature of the desktop metaphor and users' familiarity with the graphics, while the second language attempts to tap into users' familiarity with more textual domains (such as reading, recipes and lists of instructions).

Moreover, our work differs from previous work in that we test both comprehension *and* generation. Our study attempts to understand whether the differences in the languages affect both program construction and comprehension. Finally, ours is a study of an entire programming environment, not just of the syntax or micro-structures of the representation languages.

2. THE TWO REPRESENTATION LANGUAGES

To construct a program in Pursuit, users enter `record` mode and demonstrate the program's functionality on actual pieces of data. For example, consider a program that places a compressed copy of all the `.tex` files in the `papers` folder that were edited today into the `backups` folder. To create this program, the user (1) selects all the `.tex` files in the `papers` folder that were edited today, (2) selects `copy` from a pull-down menu, (3) selects all the resulting copies, (4) selects `compress` from a menu, (5) reselects the copies, and (6) drags them to the `backups` folder. As the user executes each operation (`copy`, `compress`, `move`), Pursuit presents the evolving program in a special program window on the desktop (details of how Pursuit works and how users interact with the system can be found in Modugno, 1995). Figures 1 and 2 are two representations of this program.

2.1. Language 1 – A Comic-Strip Graphical Language

The representation language in Figure 1 is based on the comic-strip metaphor (Kurlander and Feiner, 1988). Figure 1a displays a declaration for the set of `.tex` files edited today. When Pursuit infers a set, it generates a declaration showing any set attributes. In this comic-strip language, sets are represented by overlapping two file icons, and attributes are represented as graphical constructs attached to the set.

Figure 1b-e shows the operations performed on the declared file set. Two panels are used to represent an operation. The first panel shows the data icons before the operation and the second panel shows the data after. A program is a series of operation panels concatenated together, along with representations for loops, conditionals, variables and parameters. Control constructs are represented graphically. Loops are represented with a shaded rectangle enclosing the abstract representation of the loop operations along with a definition of the loop parameter. Branches are represented with a black square initiating diverging lines. Figure 3 shows an example of a complex program containing an explicit loop and conditional based on the outcome of an operation (`copy` in this case).

The comic-strip language explicitly displays the state of data objects before and after each operation.

[1] For an excellent discussion of this issue, see Petre, 1995

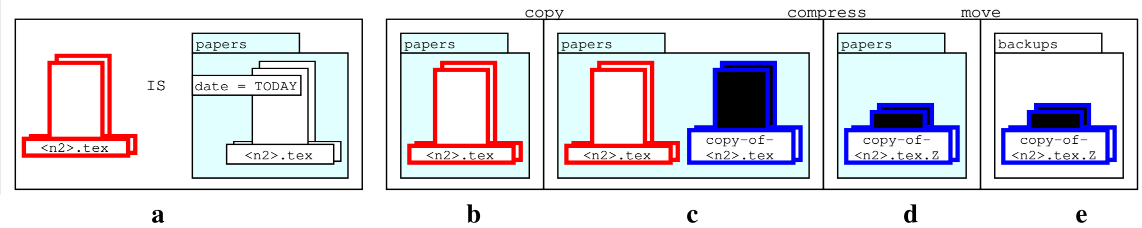

Figure 1: **A Simple Program in the Comic-Strip Language**
a) A declaration defining the set to be all `.tex` files in the `papers` folder that were edited TODAY. The program copies all those files (panels b and c); compresses the copies (panel d); and moves the copies to the `backups` folder (panel e). The `compress` operation is represented by the difference in the height and name of the icon for the copies in panels c and d. These differences are similar to the changes seen in the actual interface, where `compress` replaces a file's icon with a shorter one and appends a ".Z" to its name. The equivalent program in the text-based language is shown in Figure 2.

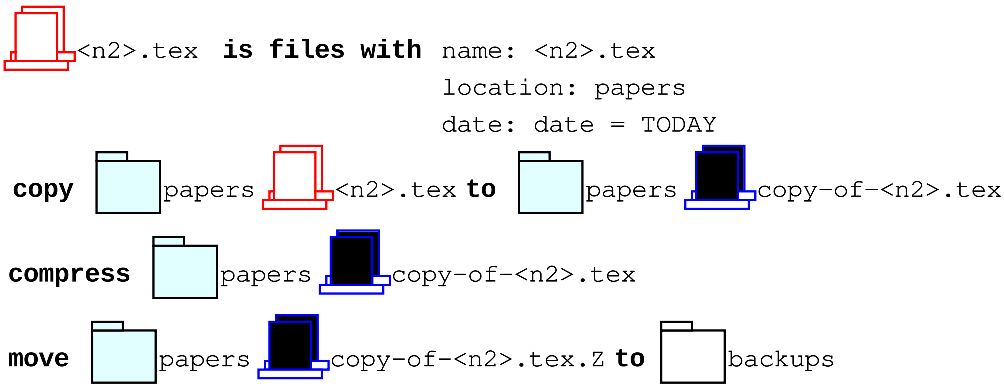

Figure 2: **A Simple Program in the Text-based Language**
The program in Figure 1 represented in the text-based language. Notice the textual attributes (`name`, `location` and `date`) attached to the declaration.

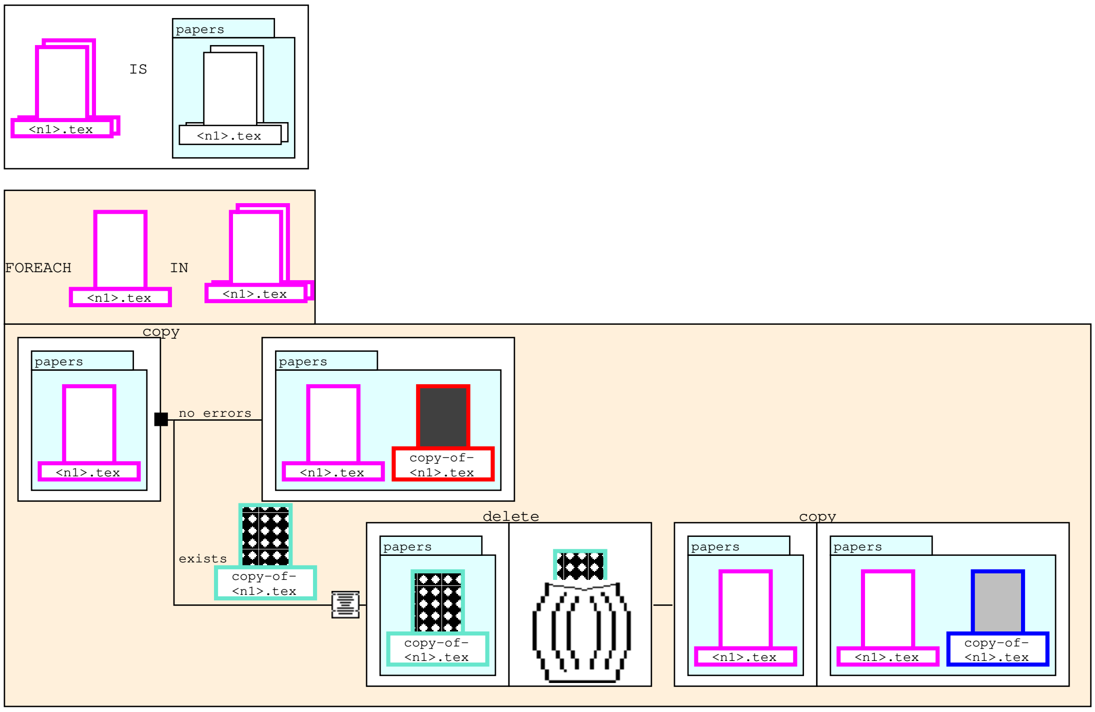

Figure 3: **A Complex Program in the Comic-Strip Language**

A complex program in the comic-strip language containing an explicit loop and branch. The program copies each `.tex` file in the `papers` folder. The initial copy operation either succeeds (the upper branch) or fails (the lower branch). This conditional is depicted graphically by the branch (i.e., the little black square) and predicates after the first panel for the `copy` operation. The upper branch represents the successful `copy`, while the lower branch represents the unsuccessful one. If the `copy` 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. Pursuit generated this program as the user demonstrated the actions on two actual file objects: one in which the `copy` executed successfully, and one in which the output file object already existed in the `papers` folder. The loop construct is represented by the surrounding rectangle enclosing the loop operations.

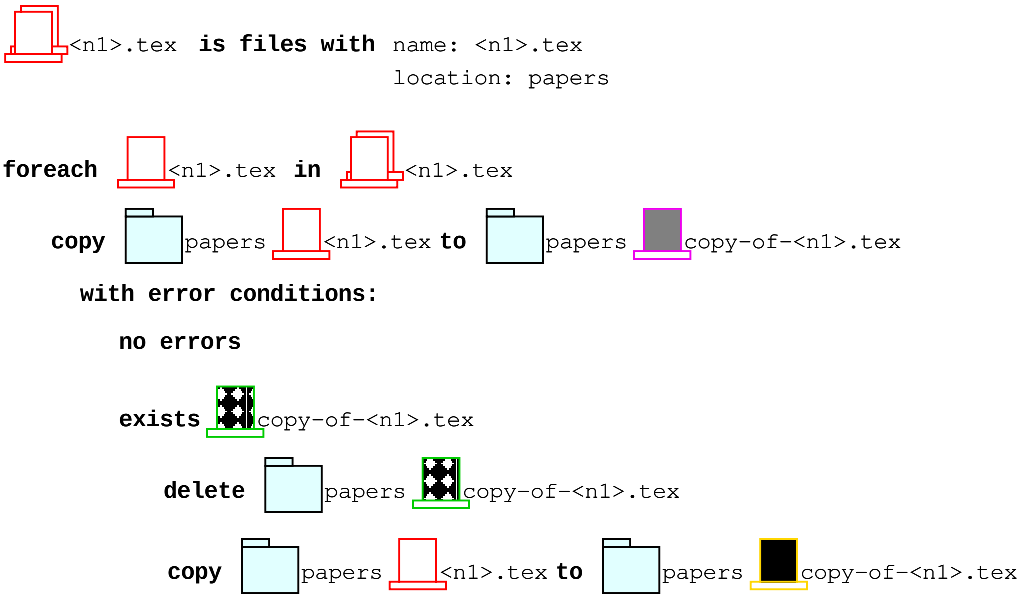

Figure 4: **A Complex Program in the Text-Based Language**

The program in Figure 3 represented in the text-based language. Notice the representation of the loop through a combination of text (`foreach`) and indentation of loop operations. A similar combination is used to represent the branch on the outcome of the `copy` operation (`exists...`).

To identify objects throughout a sequence of operations, icons are assigned unique colors. Although an icon's name, etc. may change throughout the script, its color remains the same (e.g., the icon shown dark blue on the screen and black in Figure 1c-e).

Related Languages. The comic-strip language is similar to the editable histories of Chimera (Kurlander and Feiner, 1988) and Mondrian (Lieberman, 1993), but there are many important differences. First, the comic-strip language contains *abstractions* that resemble the real interface objects they represent. In contrast, Chimera and Mondrian use actual screen snapshots in their representations. Also, Pursuit's inferences are displayed in a program, and are always visible, whereas Chimera's inferences are contained in textual supplements and Mondrian's are contained in speech feedback. Finally, Pursuit's comic-strip language visibly represents loops and conditionals, which are inferred automatically, whereas Chimera and Mondrian have no explicit loop and conditional representations and neither automatically infers conditionals.

2.2. Language 2 – A Text-Based Language with Icons

The second language is based on the "English-like" language of SmallStar (Halbert, 1984). It might also be considered visual, since data objects are represented with uniquely colored icons, and sets are represented by overlaying two icons of the same type (see the top of Figure 2). However, its representation of operations is not through before and after pictures of data objects. Instead, commands are represented by a string name (e.g., `move`, `compress`) followed by the icon of one or more data objects and any necessary connecting words, such as `to` and `from`, that further illuminate the effects of a command or any output produced. A program is a series of commands listed one after another along with representations of loops, conditionals, variables and parameters. Control constructs, such as loops and conditionals, are not represented graphically, but are represented with words suggestive of their action together with indentations to define their scope.

Because this language uses text for operations and keywords, we refer to it as "text-based." Figures 2 and 4 show the text-based versions of the programs displayed in Figures 1 and 3. Notice that like the comic-strip language, an icon is assigned unique colors so that users can easily identify it throughout a sequence of operations, even if its name, etc., changes throughout the program.

A Related Language. The text-based language version of Pursuit is based on SmallStar (Halbert, 1984), but there are many differences. First, Pursuit contains an inference mechanism that generalizes the user's actions automatically. In contrast, SmallStar records exactly what the user does – only the object that is pointed to can be a parameter and the transcript consists of a straight-line sequence of commands. To generalize the transcript, users must edit it after the demonstration via a menu of editing commands. Second, Pursuit's text-based language contains explicit representations for branches based on the exit code of an operation (see Figure 4), user defined predicates, and visible declarations. In SmallStar, these mechanisms are "hidden" in property sheets. Finally, the text-based language contains abstract representations of sets that can be manipulated (see Figure 2) as a single object. SmallStar does not contain this feature.

2.3. Comparing the Comic-Strip and Text-Based Languages

Although the comic-strip and text-based languages are visibly different, they are functionally equivalent. That is, the languages were designed so that there is a one-to-one mapping between their respective commands and between their respective constructs. For example, both contain icons for data objects and declarations for abstract sets. However, in the comic-strip language attributes are attached to set icons while in the text-based language they appear as text next to the icons. Thus, to learn each language users had to learn the same number of constructs.

Furthermore, the actions performed in demonstrating programs are identical across languages, and the interaction techniques for editing the languages are also the same. For example, assume that Pursuit incorrectly inferred the file set name in Figures 1 and 2, or that the user wants to change the program to work on all `.mss` instead. To correct the inference error or create a new program, the user just edits the file set name directly in either representation. As a second example, assume the user wishes augment the program in Figures 1 and 2 so that it removes the "copy-of-" prefix from all the backup copies. In either version of Pursuit, the user simply adds a stop point after the `compress` or `move` operations and executes the program. When the program reaches the stop point, Pursuit automatically enters record mode, and the user can continue demonstrating the operations to add to the program. Other editing features, such as cutting, copying, and pasting operations, wrapping operations in a loop, adding user-defined branches and selecting parameters, and how the users employ them, are the same for both versions of Pursuit. Thus, the concepts and techniques users need to learn to use the entire Pursuit system do not vary with representation language.

Despite these equivalences, the representation languages are conceptually different. The comic-strip language explicitly represents data objects and implicitly represents operations by changes to the data objects. The language was designed to reflect the state of the main data object in the interface and the changes users see in this object as it is manipulated. The comic-strip language visibly reflects *what* the program does to the main data object[2]. On the other hand, the text-based language explicitly represents both data and operations. The language was designed to specify how the data is manipulated so that programs are a series of action-object statements. The text-based language describes *how* the program works.

3. EVALUATION STUDY

To evaluate the efficacy of each representation language, we implemented a version of Pursuit using Garnet (Myers, et al., 1988) for each representation language and then compared the two versions. PBD actions and program editing features were identical across environments. The comparative study had three parts. The first part involved doing a self-paced tutorial that introduced Pursuit. In the second part, users constructed new programs with Pursuit's PBD facilities. The final part assessed users' comprehension of the program representation. The study was done on a Sun SPARC I color workstation with X11 windows.

3.1. Subjects

Sixteen subjects were each paid $50 to participate in this experiment. Each subject had at least 3 years experience using the Apple Macintosh, used the Macintosh at least every other day, and had no programming experience. All subjects were university students, either seniors majoring in English, history or graphics design, or graduate students in English or history (with English or history undergraduate degrees). Subjects were randomly divided into two 8-person groups (with

[2] Initially, the comic-strip language reflected the changes to all data objects affected by the operation. However, such a depiction took up considerable screen real estate. Thus we streamlined the language to emphasize only the main data object. Informal evaluations with paper-and-pencil mockups of example programs showed no difference in program comprehension between the full comic-strip version of the language and the more streamlined version. For details of the language and ways in which it was altered to be more concise, see Modugno, 1995

the constraint that each group contained the same number of people with a given academic major background). The comic-strip group used the comic-strip version of Pursuit and the text-based group used the text-based version.

3.2. General Materials – Program Categories and Tasks

For the purpose of this study, programs were divided into two categories: simple and complex. *Simple* programs were straight-line programs containing no explicit loop or branch constructs (e.g., Figures 1 and 2). *Complex* programs contained an explicit loop and either a branch based on the exit condition of an operation or a user-constructed branch with predicates (e.g., Figures 3 and 4). In this way, we were able to determine if a user's ability was affected by both program representation and complexity.

In order to derive the tasks for this study, we randomly selected a set of scripts from a larger set of C-shell scripts gathered for an informal survey at CMU (Botzum, in progress). We then studied the structure of the scripts, in terms of loops, conditionals, variables, inputs, etc., and classified them according to our simple/complex scheme. We then chose four scripts that represented the most common structures in our classification and modified them as necessary to fit the operations available in Pursuit. In this way, we insured that the sample tasks were representative of the form of programs users tended to construct in a traditional shell environment.

3.3. General Procedure

Subjects were tested individually in two sessions. The first session lasted approximately 3 hours. It consisted of a 1.5 to 2 hour tutorial on Pursuit, which included walk-through construction of 4 example programs, followed by the program construction task. The comprehension task was presented in a second session the following day. This session lasted approximately 2 hours. Upon completion of the entire study, subjects were asked to fill out a 7 page evaluation questionnaire.

3.4. Part 1 – The Pursuit Tutorial

The first part of the study introduced subjects to the Pursuit interface, general programming concepts, Programming by Demonstration, and the Pursuit representation language. It provided subjects with the background needed in the remaining parts of the study. The tutorial consisted of an 80 page description of Pursuit. In addition to explaining all the commands and menus, the tutorial contained 4 walk-through examples (2 simple and 2 complex) of program construction that explained the general programming concepts of loops, variables and conditionals, the representations of data, operations and program constructs, and all the interaction and editing techniques. The tutorials for both groups differed only in their explanation of program representations. Interaction and editing techniques, as well as explanation of general programming concepts were identical. An analysis of variance of the time spent on each tutorial example showed no difference between the two groups.

3.5. Part 2 – Program Generation

The second part of the study examined whether the representation language had any effect on the user's ability to construct programs in Pursuit.

Part 2 Materials. Materials for Part 2 included an instruction page for the generation part of the task and a one-page description of each of the four tasks for which the users were to construct

	Percent Correct		Time in Minutes		User Evaluation	
	Comic-strip	Text-based	Comic-strip	Text-based	Comic-strip	Text-based
Simple	87.5	63.5	12.2	12.1	2.23	2.88
Complex	56.0	12.5	19.2	21.3	3.97	4.47

Table 1: **Summary Results for the Generation Study**

Percent of Programs Correctly Generated (out of 2), Average Time in Minutes Spent on a Generation Task, and Average User Rating of Task Difficulty (1 = very easy, 5 = not easy) per subject in each group divided by task complexity. Users in the comic-strip group were significantly more accurate in generating programs, F(1,28)=13.00, p<.002. Users in both groups were significantly faster, F(1,28)=12.9, p<.002, and more accurate, F(1,28)=9.31, p<.005, when constructing simple programs and rated simple programs significantly easier to construct, F(1,28)=28.9, p<.0001. There were no other reliable main effects or interactions.

programs. The four generation tasks were similar in structure to the four examples in the tutorial (i.e., they contained similar program constructs such as sets, loops, error conditions and user defined branches). The subjects had to generate two simple and two complex programs. The first three tasks did not *require* any editing (that is, the user could demonstrate the program in such a way that Pursuit made all the correct inferences, although there were other ways to demonstrate the programs that would make editing necessary). The fourth task required users to copy and paste operations, wrap operations in a loop, select a loop parameter and edit the parameter's name.

Each task description contained a high level verbal description of the task, such as "backup all the `.tex` files in the `papers` folder," along with two pictures. The first picture (initial state) depicted the desktop as the user saw it upon beginning the task. The second depicted the desktop as it would look if the program the user wrote were executed in the initial state. The user could thus verify if a program worked by resetting the desktop state (which was a special feature added to Pursuit for the purpose of this study), executing the program, and checking to see that the final state matched the one shown in the task description. The task descriptions, as well as the general instructions for this part of the study, were identical for both groups.

Part 2 Procedure. Subjects were given the generation task instructions along with the four task descriptions and asked to construct programs for the tasks.

Part 2 Results. For each group, Table 1 shows the percent of correctly generated programs, average time spent on each task, and user rating of task difficulty. A two-way ANOVA was performed on each of the data sets. Subjects in the comic-strip group were twice as accurate (72% vs 37%) in generating programs as those in the text-based group, F(1,28)=13.00, p<.002. In addition, subjects in both groups were twice as accurate when constructing simple programs compared to complex programs, F(1,28)=9.31, p<.005. The interaction between representation language and complexity was not significant.

Not surprisingly, subjects in both groups were reliably faster in constructing the simple programs, F(1,28)=12.9, p<.002, and user ratings indicated that simple programs were easier to construct than complex program, F(1,28)=28.9, p<.0001. Although the comic-strip group performed faster and rated the tasks somewhat easier, these differences were not significant and the interactions between representation language and complexity were not significant.

3.6. Part 3 – Program Comprehension

The purpose of this part of the study was to investigate the effect of representation on three programming scenarios:

- "I know I have a program that does X. Is this the right program for task X?"
- "This program has a bug. It's supposed to do X. Where is the bug?"
- "I wrote this program to do X. Now I need a program to do a similar task, Y. How can I change program X to do Y?"

The comprehension part of the study required true/false decisions. For each decision, the subject was presented a task description and a program and asked if the program did the task. If so, the subject was given a modified task description and asked to explain how to modify the program to do the new task. If the program did not do the task, the subject was asked to explain why. Thus, this part of the study tested whether users could identify if a program implemented a particular task, whether users could identify why a program did not implement a particular task, and whether users could modify a given program to do a slightly modified task.

Part 3 Materials. Eight task descriptions were used, four simple and four complex. A sequence of programs was presented for each task until a correct program was encountered. The first program presented was the correct program for two simple and two complex tasks. The second program was correct for one simple and one complex task and the third program was correct for the remaining two tasks. Thus, users made 14 true/false decisions, with 8 "correct" programs and 6 "buggy" programs.

Since we were investigating the effects of program representation on identifying certain kinds of bugs, only two types of bugs were introduced into "buggy" programs, and each type of program (simple or complex) contained only one type of bug. Simple programs contained "state" bugs. Examples include incorrect or missing operations, incorrect or missing data objects, and out of order operations. Complex programs had "structural" bugs, such as missing branches, operations outside of loops, etc.

Similarly, we requested only a limited number of modifications, such as adding and deleting operations, attributes and control constructs, so that we could determine if the graphical constructs were easier to modify than their textual counterparts.

Part 3 Procedure. Subjects were given a brief tutorial containing 3 walk-through examples to familiarize them with the interface and interaction techniques for this part of the study. Users entered their true/false responses by clicking on a button. Subjects were given feedback on the correctness of each true/false decision, after which they were asked for an additional response (regardless of whether the true/false decision was correct or incorrect). In the case of a buggy program, the subject was asked why the program did not work. In the case of a correct program, the subject was asked how to modify the program to complete a slightly modified task. In each case, subjects typed their responses in a simple text editor. Upon completion of this part of the study, subjects were asked to do a think aloud protocol of 3 additional tasks description/program pairs in order for us to gain insight into their thought processes during this part of the study.

	% Correct		Time (Min)		% Bugs Found		% Mods Found	
	State	Text	State	Text	State	Text	State	Text
Simple	73.1	71.4	1.48	1.14	70.7	62.3	66.8	65.8
Complex	71.4	41.0	1.62	1.30	59.0	54.0	74.0	61.0

Table 2: **Summary Results for the Comprehension Study**
Percent of Programs Correctly Identified (out of 7), Average Time In Minutes to Classify a Program, Percent of Bugs Correctly Identified (out of 3), and Percent of Correct Program Modifications Identified (out of 4). The text-based group was significantly faster, F(1,28)=6.44, p<.02. There were no other reliable main effects or interactions.

Part 3 Results. Table 2 summarizes the four measures of task performance: (1) percent of programs correctly classified[3], (2) average time to make the classification decision, (3) percent of bugs identified, and (4) percent of successful program modifications.

Although the comic-strip group made about 25% fewer errors in classifying programs, this difference is not significant in an ANOVA. On the other hand, the text-based group spent 25% *less* time classifying a program, and this difference was significant, (F(1,28)=6.44, p<.02). No other accuracy or time effects were significant.

In scoring both the bug identification task and the program modification task, we defined each solution(s) in terms of a set of addition and/or deletion operations. An addition (deletion) operation is defined to be the action of adding (deleting) an operation, attribute or predicate together with a specification of the operation's parameters and a location for the action. For example, if a program modification required moving a set of files to a particular folder, the correct answer would be something like this: "Add move of `.tex` file set to `backups` folder after `compress` operation." A subject's response was then scored by adding up all the missed or incorrect components of additions and deletions. For bug identification, the maximum number of bugs missed was 3 for simple tasks and 3 for complex tasks. Modifications missed could be as high as 4 for both simple and complex. Although the comic-strip group consistently did better, an ANOVA showed there were no significant main effects or interactions for program representation and complexity.

4. DISCUSSION

The purpose of this study was to see if non-programmers would be able to generate and comprehend Pursuit programs. Although we are careful to state that the results of this study are only suggestive, they are nonetheless very promising. Given the difficulty that introductory students have learning to program, it is unlikely that non-programmers with only two hours of training could write programs like the ones in this study in any traditional language such as C-shell. Thus, this study demonstrates that combining PBD with an editable program representation does indeed help non-programmers gain access to the power of programming. The study further suggests that care must be taken to maximize the potential of this combination. We further explore that idea in this section.

[3] Because there were unequal numbers of correct programs and incorrect programs, percent correct = (percent correct true responses + percent correct false responses)/2,

4.1. Program Generation

As discussed in Section 3.5, the comic-strip language significantly enhanced users' ability to accurately generate programs in the PBD system. This result is interesting because the user actions in constructing and editing programs are absolutely identical across the two program representations. Thus, what users need to know about *how* to construct a program (in terms of actions and concepts) is constant across the two conditions. Therefore, this result suggests that the program representation significantly affected users' abilities to use the PBD system.

Although we have no answer as to why the comic-strip group performed so much better, we conjecture that the graphical representation of control constructs in the comic-strip language may play an important role. The fact that users in the text-based group were rarely able to construct complex programs indicates that they did not fully understand when and how to use loops and conditionals. Since the explanations of the concepts underlying these constructs were identical in both tutorials, it is possible that the graphical representations of loops and conditionals found in the comic-strip representation further enhanced users' understanding of the programming concepts.

This hypothesis is further supported by the fact that in Part 3 (the comprehension study) users in the text-based group performed about the same as users in the comic-strip group *for simple programs*. However, for complex programs, users in the text-based groups performed much worse. They made twice as many errors in classifying a program. Moreover, the talk aloud protocols for the comprehension study showed that users in the comic-strip group tended more often to differentiate between programs with loops and conditionals and those without. This further supports our conjecture.

To examine our conjecture, we hope to perform a future study that establishes the users' understanding of the underlying programming concepts and representations as they progress through the Pursuit tutorial. For instance, after each tutorial example users could be asked to answer some questions about basic programming concepts. The questions could be repeated after the generation study to see how well the users retained the information. Such a study might reveal that one of the representation languages helps users form a better mental model of programming concepts.

4.2. Program Comprehension

Although the ANOVAs showed that there were no significant differences between the two groups, subjects in the comic-strip group consistently performed at least 10% better in the two comprehension studies. For complex programs, this difference was as great as 30%. Moreover, a T-test performed on the accuracy results for the complex programs in Part 3 of the study showed that the difference between the two groups was significant, ($t(1,14)=1.85$, $p<.04$), although a T-test performed on the timing data showed no reliable difference between the groups[4].

There are several reasons to expect differences in comprehension. First, the comic-strip language with its graphical representation of control constructs could provide users with a better overview of program structure and thus, program function. This theory is supported by the talk aloud protocols we gathered after Part 3. These protocols revealed that very often users read a task description and formed a model of the program structure for the task (for example, "I know I need a loop," or "Ok, I have to check for an exit condition"). Then they scanned the program looking for that structure. Because program constructs in the comic-strip language have a graphical representation, they may

[4]We are careful to mention these results only in the discussion because they cannot be interpreted as a firm result. They merely indicate that further investigation is required.

be easier to locate than their corresponding textual counterparts in the text-based language. We hope to further explore this hypothesis by performing an additional within-subject study, similar to the one found in Green, Petre, and Bellamy, 1991, to see if differences in the micro-structures of the two languages do indeed affect users' speed of comprehension.

In addition, the graphical aspects of the comic-strip language may make it easier to see the *scope* of control constructs. This may make various parts of a program easier to discriminate in that language. For example, in looking at Figure 3 it is fairly clear which operations are in the lower branch. It is more difficult and requires greater attention to details to identify these same operations in Figure 4.

Finally, the comic-strip language graphically depicts control *flow* and relationships between components, whereas the text-based language uses whitespace for this purpose. These concrete representations may make it easier for users to decipher program code in the comic-strip language, since control flow may be easier to follow.

4.3. Informal Results

In addition to the formal hypotheses tested by this user study, more informal results were gathered via evaluation questionnaires, discussions with subjects, and verbal protocols. For example, users in both groups generally reported a positive experience with the system and stated that it would be something they would find useful.

Additional responses provided feedback on how to improve the actual system and languages. For example, some users noted that the file icons on the `exists` branches of conditionals did not have folders. To find the folders, users had to scan forward to find the next instance of the file icon. This omission will be fixed in future versions. A brief examination of the unsuccessful attempts from the generation part of the study indicated that more guidance is needed to help users learn when and how to apply different strategies for different tasks. As one user stated about a failed attempt to generate a program: "I knew I needed some type of loop, but I couldn't figure out how to do it." Furthermore, user errors indicate that much more structure is needed in the system, although users seemed to desire more control. Several users noted that they wanted to edit the program *while* they were still demonstrating it, rather than after as the current system enforces. If such power were given to the user, care must be taken to insure that user edits do not create a state that makes continuing the program demonstration impossible.

Finally, some of the fixes to usability problems found by doing a cognitive dimensions analysis (Modugno, Green, and Myers, 1994), such as automatically decomposing a program that manipulates a set as a whole into an explicit loop, helped users during the construction task.

5. CONCLUSION

We have designed, prototyped and evaluated two graphical representation languages for a demonstrational visual shell. The languages were designed to have equivalent functionality and a one-to-one mapping between constructs. The main difference between the languages was the conceptual representation that the languages attempted to convey. The comic strip language used graphical components to depict the *effects* of a program on data. A program in the comic-strip language is represented by *what* it does to the main data object . On the other hand, the text-based language used more textual components to describe the *actions* of a program. In the text-based language, a program is represented by *how* it manipulates the main data object.

The empirical evaluation showed that Pursuit mets its goal of enabling non-programmers to construct and comprehend programs containing loops, variables and conditionals. Moreover, the comic-strip graphical language, with its emphasis on state and data, was more effective in helping users to construct programs than the essentially textual language. There was also a trend suggesting that the comic-strip language improved comprehension.

Although not conclusive, this research indicates that this new paradigm for visual languages, namely the comic-strip comic strip metaphor, could prove more successful than traditional text-based programming languages for increasing the power of PBD systems for a visual shell environment, and warrants further research. In particular, the strong result found for program generation suggests that not only does representing a program assist users in program construction, but how the program is represented can dramatically affect users' success.

6. ACKNOWLEDGMENTS

This work was partially funded by NSF grants IRI-9020089 and IRI-9319969, the Hertz Foundation and AAUW. We thank Maria Ebling, Jade Goldstein, T.R.G. Green, Tom Moher, Amy Zaremski and the anonymous reviewers for the help with presenting this work. We also thank Wayne Gray for his support in preparing this manuscript.

7. REFERENCES

Borg, K. (1990). *IShell: A Visual UNIX Shell*. In **Proceedings of CHI '90**. Seattle, Washington, pp. 201–207.

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

Cunniff, N. and Taylor, R. P. (1987). *Graphical vs. Textual Representation: An Empirical Study of Novices' Program Comprehension*. In **Empirical Studies of Programmers: Second Workshop**, edited by G. M. Olson, S. Sheppard, and E. Soloway. Washington, D.C., pp. 114–131.

Cypher, A. (1991). *EAGER: Programming Repetitive Tasks by Example*. In **Proceedings of CHI '91**. New Orleans, LA, pp. 33–40.

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

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

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

Halbert, D. C. (1984). *Programming by Example*. Ph. D. Thesis, Computer Science Division, University of California, Berkeley, CA.

Kurlander, D. and Feiner, S. (1988). *Editable Graphical Histories*. In **IEEE Workshop on Visual Languages**. Pittsburgh, PA 15213, pp. 127–134.

Lieberman, H. (1993). *Mondrian: A Teachable Graphical Editor*. In **Proceedings of InterCHI '93**. Amsterdam, The Netherlands, pp. 144.

Maulsby, D. L. and Witten, I. H. (1989). *Inducing Programs in a Direct-Manipulation Environment*. In **Proceedings of CHI '89**. Austin, TX, pp. 57–62.

Modugno, F., Green, T., and Myers, B. A. (1994). *Visual Programming in a Visual DomaIn A Case Study of Cognitive Dimensions*. In **HCI '94**. Glasgow, Scotland.

Modugno, F. (1995). *Extending End-User Programming in a Visual Shell with Programming by Demonstration and Graphical Language Techniques*. Ph. D. Thesis, Carnegie Mellon University, Pittsburgh, PA 15213, March 1995. Also available as *Technical Report CMU-CS-95-130*.

Moher, T. B. et al. (1993). *Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets*. In **Empirical Studies of Programmers: Fifth Workshop**. Ablex Publishing Corporation.

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

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

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

Petre, M. (1995). *Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming*. **Communications of the ACM**, 38(6), pp. 33–44.