

Graphical Representation of Programs in a Demonstrational Visual Shell—An Empirical Evaluation

FRANCESMARY MODUGNO

University of Washington

and

ALBERT T. CORBETT and BRAD A. MYERS

Carnegie Mellon University

An open question in the area of Programming by Demonstration (PBD) is how to best represent the inferred program. Without a way to view, edit, and share programs, PBD systems will never reach their full potential. We designed and implemented two graphical representation languages for a PBD desktop similar to the Apple Macintosh Finder. Although a user study showed that both languages enabled nonprogrammers to generate and comprehend programs, the study also revealed that the language that more closely reflected the desktop domain doubled users' abilities to accurately generate programs. Trends suggest that the same language was easier for users to comprehend. These findings suggest that it is possible for a PBD system to enable nonprogrammers to construct programs and that the form of the representation can impact the PBD system's effectiveness. A paper-and-pencil evaluation of the two versions of the PBD desktop prior to the study supported these findings and provided interesting feedback on the interaction between usability evaluations and user studies. In particular, the comparison of the paper-and-pencil evaluation with the empirical evaluation suggested that nonempirical evaluation techniques can provide guidance into how to interpret empirical data and, in particular, that PBD systems need to provide support for programming-strategy selection in order to be successful.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming; D.1.7 [**Programming Techniques**]: Visual Programming; D.m [**Software**]: Miscellaneous—*software psychology*

General Terms: Experimentation, Human Factors

Additional Key Words and Phrases: Programming by Demonstration, Pursuit

This research was funded by NSF grants IRI-9020089 and IRI-9319969 and by grants from the Hertz Foundation and the AAUW.

An earlier version of this article appears in *Empirical Studies of Programmers 6th Workshop*, Jan., 1996, pp. 131–146.

Authors' addresses: F. Modugno, Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98115; A. Corbett and B. A. Myers, Human-Computer Interaction Institute, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

© 1997 ACM 1073-0516/97/0900-0276 \$03.50

1. INTRODUCTION

Programming by Demonstration (PBD) systems [Cypher 1993] have shown promise in enabling nonprogrammers to automate tasks in direct-manipulation environments (e.g., Cypher [1991], Halbert [1984], Maulsby et al. [1993], and Maulsby and Witten [1989]). In a PBD system, the user executes concrete actions on actual data objects, and the underlying system uses a set of predefined heuristics to infer a more general procedure [Myers 1992]. Such systems are considered “easy to use” because users construct programs in the same way they interact with the system: by directly manipulating data objects.

However, most PBD systems have one well-known shortcoming: they do not represent the resulting program for the end-user. This poses several problems. First, users cannot examine a program’s contents. This makes it difficult for users to verify whether the inferred program is correct, to recall a program’s function at a later date, or to share programs with other users. Moreover, without an *editable* representation, it is impossible for users to directly edit the generated code. Thus, to enable users to edit programs, the system would have to rely on other techniques such as incremental learning. In some cases, this can be more difficult or time consuming than simply changing the program directly.

Finally, if a PBD system does not provide users with an incremental representation of the developing program during the demonstration, then the system must find some other way to disambiguate and communicate its inferences to users. Most PBD systems achieve this by questions and answers (e.g., Peridot [Myers 1988]), by dialog boxes (e.g., MetaMouse [Maulsby and Witten 1989]), or by highlighting the expected user action (e.g., Eager [Cypher 1991]). However, users sometimes find these interactions disruptive and confusing, and often they simply select the default option [Cypher 1991].

Thus, without a way to communicate unobtrusively with users and without a way for users to view, modify and share programs, any PBD system will never realize its full potential.

This article discusses the results of an empirical evaluation of Pursuit, a PBD system for a desktop domain (or “visual shell”) similar to the Apple Macintosh Finder. The goal of Pursuit was to provide end-user programming to nonprogrammers without their having to acquire expert programming skills.

In designing Pursuit, we explored two alternative graphical program representation languages. Each language 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.

The purpose of the user study was twofold. First, we wished to see if Pursuit met its goal of enabling nonprogrammers to create programs containing loops, variables, and conditionals. Second, we wanted to determine whether the form of the representation language would have any impact on Pursuit's effectiveness.

To explore these questions, we performed a three-part user study. The study involved 16 nonprogrammers, eight of whom used the comic-strip version of Pursuit and eight of whom used the text-based version. The first part of the study consisted of a self-paced tutorial that introduced the users to one of the versions of Pursuit. In the second part, users constructed new programs with Pursuit's PBD facilities. The final part assessed users' comprehension of program representations. The study was performed on a Sun SPARC I color workstation with X11 windows.

The results of the study show that users in both groups were able to successfully generate and comprehend programs in Pursuit. Hence, Pursuit attained its original goal. Moreover, users in the comic-strip group were more than twice as likely to construct an accurate program than users in the text-based group. This difference was significant. In addition, trends indicate that users were better able to comprehend comic-strip programs than text-based programs. Therefore, this study suggests that PBD techniques can successfully assist users in programming and that seemingly small details of the system can greatly alter the system's effectiveness.

In addition, we found another very interesting result by comparing the outcome of the empirical evaluation with an earlier paper-and-pencil evaluation of the systems using Cognitive Dimensions (CDs) [Green 1989; 1991]. In particular, by analyzing the log files from the study in light of the CD analysis, we discovered that despite the fact that PBD helps nonprogrammers to construct programs, the technique needs to provide more support for helping users determine *how* to construct such programs (i.e., PBD systems need to help users identify and select the particular programming strategy needed to solve a given problem). During the CD analysis, we discovered this "strategy-selection problem" and even devised a solution to one instance of this problem. Analysis of the log files showed that two-thirds of the subjects encountered and successfully used our solution. This points to the utility of such a mechanism within PBD systems. However, the log files also revealed instances in which other strategy-selection problems arose. This suggests the need for future research into devising other ways to support programming-strategy selection in demonstrational systems.

The remainder of this article is organized as follows. We begin by briefly introducing the two versions of Pursuit. We then discuss the actual study and its results. All materials used in the study as well as a comprehensive listing of the study's results are found in the appendices of Modugno [1995]. Finally, we discuss how the results of the user study relate to an earlier paper-and-pencil evaluation of the systems using cognitive dimensions. We conclude with the lessons learned, as well as suggestions for future work.

2. THE PURSUIT PBD DESKTOP

The popularity of the Macintosh and more recent desktop interfaces, such as Microsoft Windows, speaks to the success of the desktop metaphor, especially among typical users (i.e., other than computer scientists). The illusion of directly manipulating data objects rather than issuing abstract textual commands makes interacting with computers more concrete and therefore simpler [Shneiderman 1983]. However, it is well recognized that desktop interfaces or “visual shells,” unlike command-based shells such as Unix, are limited in the programming power available to end-users. In particular, most visual shells provide no way for users to automate even the simplest repetitive tasks. The goal of our research was to find a way to incorporate the desired programming capabilities into the desktop in a way that is consistent with the direct-manipulation paradigm.

The success of PBD systems in providing end-user programming in other domains, and in particular the success of SmallStar [Halbert 1984] in providing this capability for the Star [Smith et al. 1982] desktop, suggested that the visual-shell domain might benefit from the technological developments in the PBD community. This seemed especially promising because with PBD programs are specified by directly manipulating data objects. In contrast with traditional program specification, in which users first write a program in some abstract programming language and then execute the program, specifying programs by demonstration seemed more consistent with the principles of the desktop interface.

At the same time, the broadness of the visual-shell domain suggested it might provide a good test bed for investigating the previously unexplored question of program representation for PBD systems.

To construct a file manipulation program in Pursuit, the user demonstrates its actions on real data, and Pursuit infers a general procedure. For example, a Pursuit user can create a simple program to make a compressed backup copy of all **.tex** files in the **papers** folder that were edited today by (1) selecting the existing **.tex** files edited today (for example, **a.tex**, **b.tex**, and **c.tex**) in the **papers** folder, (2) executing the **copy** command, (3) selecting the output files (the copies), and (4) executing the **compress** command. As the user executes each command (**copy**, **compress**), Pursuit presents the evolving program in a special program window on the desktop (see Modugno [1995] for details of how Pursuit works and how the user interacts with the system).

Figures 1–4 are actual screen snapshots of the developing program in each of the two representation languages. The programs also illustrate some of the data primitives found in each version of Pursuit: single file and folders, sets of files and folders, and attributes such as dates, names, and user ids.

2.1 The Comic-Strip Representation Language

The representation language in Figures 1–2 is based on the comic-strip metaphor [Kurlander and Feiner 1988]. The top panel of Figure 1 displays

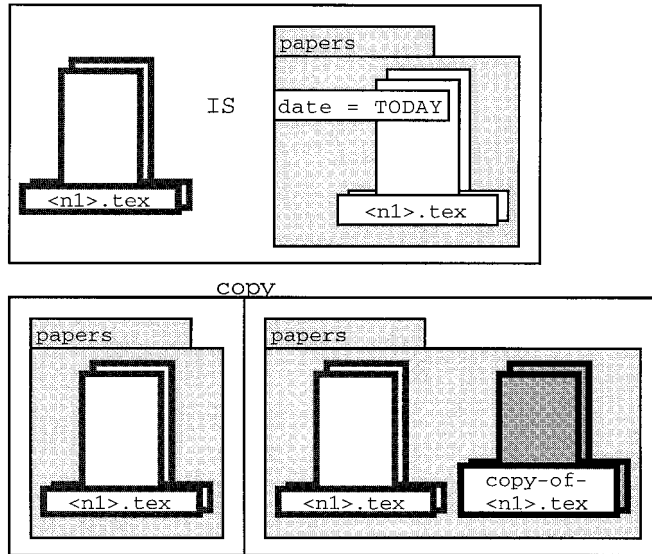


Fig. 1. The initial operation of example (1)—selecting the existing `.tex` files edited today in the `papers` folder—in the comic-strip language. To begin, the user makes a copy of all the `.tex` files in the `papers` folder that were edited today. The upper panel is a declaration defining the set of files that the program manipulates. It represents an inference made by Pursuit. The `copy` operation is depicted by the changes in the icons between the first and second panels: a new file set icon is added to the `papers` folder to represent the output of the `copy`.

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 2 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 an enclosing rectangle, and diverging lines represent branches. Figure 5 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. 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 in dark blue on the screen and black in Figure 2).

2.2 The Text-Based Representation Language

The second language is based on the “English-like” language of SmallStar [Halbert 1984]. It might also be considered visual, because data objects are

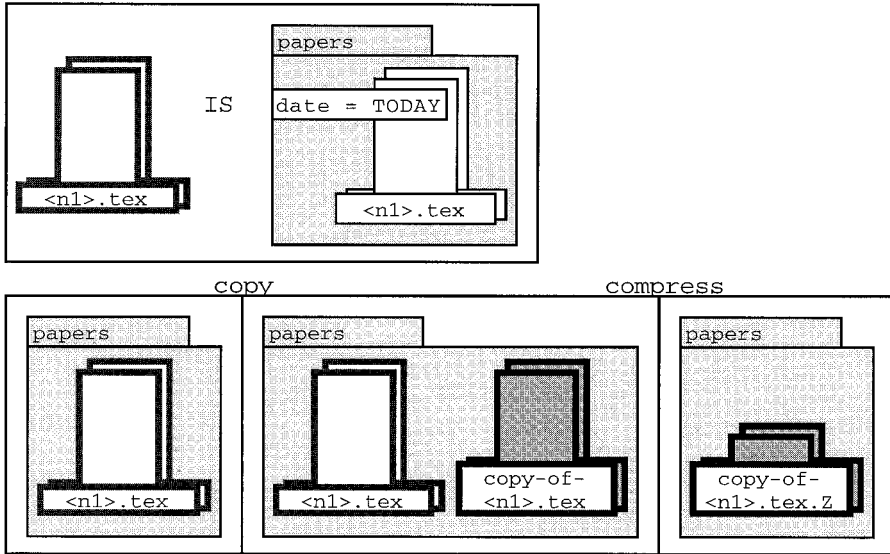


Fig. 2. The complete program of example (1)—selecting the existing **.tex** files edited today in the **papers** folder—in the comic-strip language. After the user compresses the copies, a new panel is added to the program. Only one panel is added because the epilogue of the **copy** can serve as the prologue of the **compress**. The program is a sequence of actions just like a comic strip.

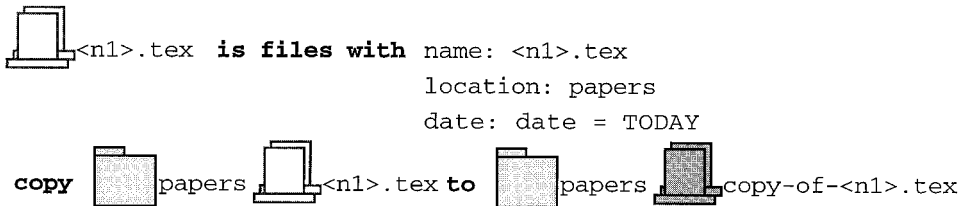


Fig. 3. The initial operation of example (1)—selecting the existing **.tex** files edited today in the **papers** folder—in the text-based language. After the user selects all the **.tex** files in the **papers** folder that were edited today and copies them these “lines” appear. The first three lines are the declaration defining the set that the program manipulates. The last line is the representation of the copy operation. Compare this representation to the comic-strip representation in Figure 1.

represented with uniquely colored icons, and sets are represented by overlaying two icons of the same type (see the top of Figure 3). 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., **copy**, **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 a representation of loops, conditionals, variables, and parameters (see Figure 6). Control constructs, such as loops and conditionals, are not represented graphically, but are

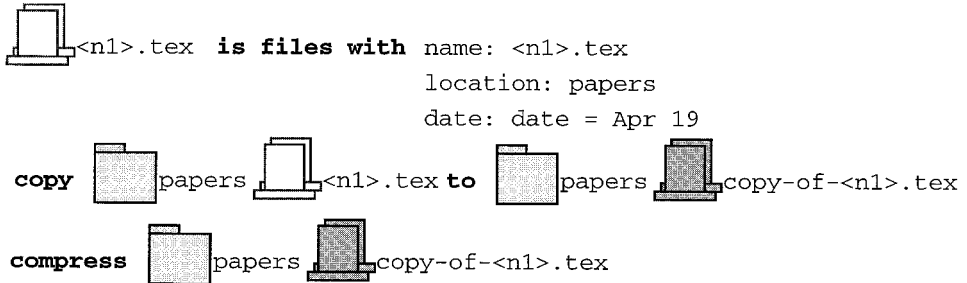


Fig. 4. The complete program of example (1)—selecting the existing **.tex** files edited today in the **papers** folder—in the text-based language. After the user compresses the copies, a new line is added to the program. The program in the text-based language is a sequence of commands similar to more traditional textual languages. Compare this representation with the comic-strip representation in Figure 2.

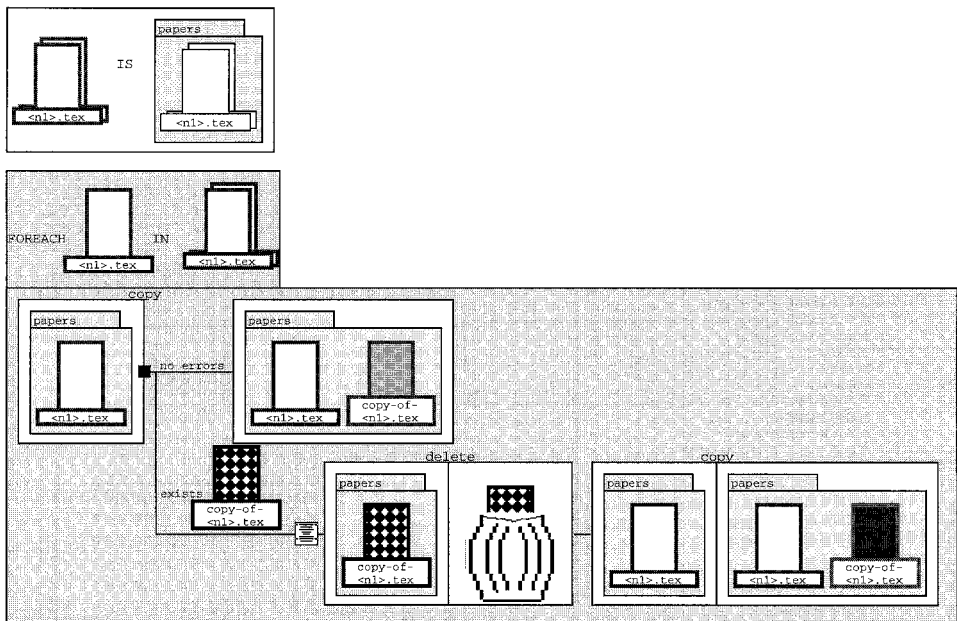


Fig. 5. An explicit loop with conditional in the comic-strip language. The figure shows a program that makes a copy of all the **.tex** files in the **papers** folder. If a file with the output copy name already exists in the **papers** folder, then the program deletes that file before making the copy. Notice the representation of the loop through the large outer rectangle enclosing the loop operations. The conditional is indicated by the conditional marker with two branches each containing an annotation (**no errors** and **exists**) explaining under what conditions the program will execute the branch's operations.

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 3, 4, and 6 show the text-based versions of the programs displayed in Figures 1, 2, and 5. Notice that like the comic-strip

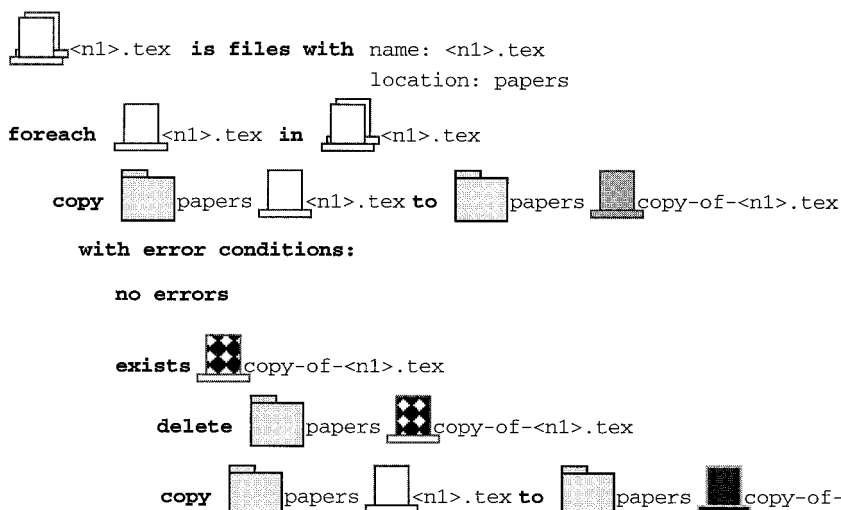


Fig. 6. An explicit loop with conditional in the text-based language. The figure shows the program in Figure 5 represented in the text-based language. Notice that the representation of the loop through the keywords **foreach** . . . and the intended operations. The conditional is indicated by the indented conditional keywords **with error condition:** along with the indented branch annotations (**no errors** and **exists** . . .) explaining under what conditions the program will execute the indented branch operations.

language, an icon is assigned a unique color so that users can easily identify it throughout a sequence of operations, even if its name, etc., changes throughout the program.

2.3 Editing Programs

In order to enable users to change a program, Pursuit contains a graphical language editor. This enables users to fix incorrect inferences, add or delete operations, change data attributes, add loops and conditionals, select and name parameters, etc. The editor is similar to a direct-manipulation text editor, and its functions and interaction techniques are identical for both representation languages. For example, users select a data object by clicking on its icon anywhere in the program. An operation is selected by clicking and dragging the mouse across its representation. In the comic-strip language, the user must drag the mouse through all the *panels* that constitute the chosen operation. In the text-based language, the user must drag the mouse through all the *lines* that constitute the chosen operation.

Once the user selects an object, appropriate editing commands appear in the edit menu. 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. For attribute edits, Pursuit does simple range checking to ensure correctness. For example, when the user edits a date attribute, the resulting edit is checked to ensure it is a valid date. To help maintain consistency, edits are immediately

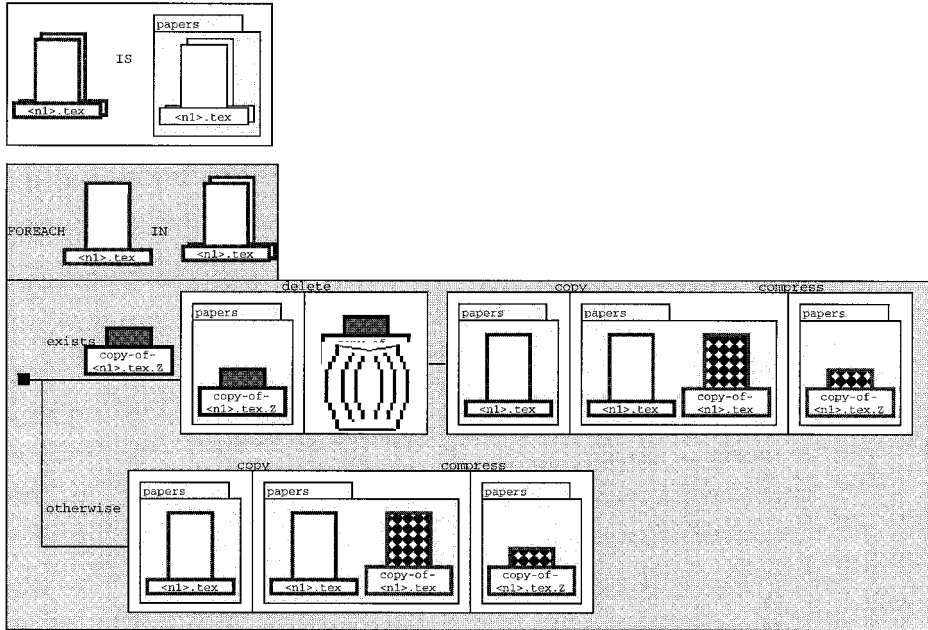


Fig. 7. A program containing a user-defined branch in the comic-strip language. The figure shows a program that copies and compresses all the `.tex` files in the `papers` folder. If a compressed copy already exists in the `papers` folder, the program first deletes that copy. The branch is indicated by the little black square and diverging lines. Note that the program could also have been written with the `compress` operation factored out into the main line of the program. In that case, there would be a reverse branch coming from the epilogues of the `copy` operation, joining and leading into the prologue of the `compress` operation.


propagated throughout the program. For example, if the user changes the name of a file set icon, all instances of the icon are immediately updated.












Users can also add their own user-defined branches (similar to the Lisp `cond` construct) to programs. Examples are shown in Figures 7 and 8. Users construct branches through a series of menus that incorporate icons from the program for easy identification. The menus, and hence the method of construction, are the same for both languages. Although the representations of the branch constructs differ between the languages, the representations of the test predicates are the same. Users can add predicates to test for the existence of an object, to compare the properties of two objects, such as their names or date of modification, or to compare the property of an object to a system constant, such as `USER` or `TODAY`.







2.4 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 among their respective commands and among their respective constructs. For example, both contain icons for data objects and declarations for abstract sets. However, in the comic-strip

```

 <n1> is files with name: <n1>
                                location: papers

foreach  <n1> in  <n1>
  case one of:
    exists  copy-of-<n1>.Z
    delete  papers  copy-of-<n1>.Z
    copy  papers  <n1> to  papers  copy-of-<n1>
    compress  papers  copy-of-<n1>

  otherwise
    copy  papers  <n1> to  papers  copy-of-<n1>
    compress  papers  copy-of-<n1>

```

Fig. 8. A program containing a user-defined branch in the text-based language. The figure shows the program in Figure 7 represented in the text-based language. The branch is depicted by the **case one of** keywords along with the predicates and indentation of operations indicating the scope of each branch. Note that the program could also have been written with the **compress** operation factored out into the main line of the program. In that case, the indentation for the **compress** operation would be the same as the **case one of** statement.

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 must 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 2 and 4 or that the user wishes to change the program so that it works on all **.mss** files instead. To correct an inference error or to create a new program, the user just edits a file set name directly in either representation. Thus, the concepts 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 data in the interfaces and the changes users see in the data as it is manipulated. The comic-strip language visibly reflects *what* the program does, as well as what the user does when constructing the program. It visibly *shows* program actions. 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, as well as how the user constructed the program. It *describes* program actions.

3. RELATED WORK

In this section, we briefly review some of the work that has influenced Pursuit. In particular, Pursuit draws on techniques from the areas of visual programming languages and Programming by Demonstration. We begin, however, by reviewing some of the results of previous studies comparing textual and graphical program representations.

3.1 Empirical Studies of Graphical Languages

Although Pursuit is the first PBD 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 situations. 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 [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 microstructures of the representation languages.

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

3.2 Visual Representation Languages for Visual Shells

The idea of using a graphical representation language for a visual shell is not new. Other systems (such as in Jovanovic and Foley [1986], Haeberli [1988], and Henry and Hudson [1988]) have employed iconic command languages based on the *dataflow* metaphor. In the dataflow model, programs are depicted as a collection of icons that explicitly represent *commands*. Command icons are connected to show the data path through the program. This approach has several problems though.

Programs can be cumbersome and space inefficient. Furthermore, users must learn to associate a symbol with each operation. If the pictures for operations differ from the way in which operations are represented in the interface, users must learn a mass of detail that is very different from the knowledge they have already acquired by interacting with the system.

IShell [Borg 1989], a design for a Unix visual shell, is typical of the dataflow model of visual shell languages. The IShell interface contains iconic representations of each application (operation) that represent “machines” to which the user feeds data.

IShell is limited because users specify programs off-line by constructing them with the IShell editor. They must learn the syntax and the semantics of the representation language from the start. Moreover, IShell represents each application with a unique icon. Not only does this take up a lot of space on the desktop, but also users must learn the meaning of every icon and its functionality before they can transfer this knowledge to the programming language. Finally, because an IShell machine can have only one behavior, different machines are needed to specify only slightly different behaviors. This places a burden on the user to differentiate between different machine behaviors, and it contributes to the desktop clutter.

3.3 Visual Representation Languages for PBD Systems

Programming by Demonstration [Cypher 1993] is more appealing than the off-line specification of programs because it allows users to specify a program in the same way in which they invoke operations.

3.3.1 *SmallStar*. The SmallStar system [Halbert 1984] introduced demonstration as an alternative method for specifying command language programs in a visual shell. SmallStar is a prototype of the Xerox Star system, which pioneered the desktop metaphor and the direct-manipulation interface [Smith et al. 1982]. SmallStar contains a macro recorder that produces an “English-like” transcript of user actions. In the transcript, icons are used to represent files and folders, and keywords are used to represent operations. The user can generalize the transcript by editing it via a menu of commands. For example, in order to add a loop, the user must wrap the code in a loop construct. Conditionals can also be explicitly added to a transcript after it is recorded. To generalize an object, the user edits its “data description,” a property sheet that describes ways in which a user can select an object.

Pursuit's text-based language is based on SmallStar, 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, 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 as a single object. SmallStar does not contain this feature. Despite these limitations, SmallStar illustrated that demonstration can have practical applications in a commercial system. This provides evidence that a more sophisticated system like Pursuit could have greater value.

3.3.2 Chimera. Chimera [Kurlander and Feiner 1988] is a graphical editor that creates an editable, graphical history of user actions. As the user edits a picture, Chimera produces a series of panels, similar to a comic-strip, in a separate window. The panels are focused snapshots of the screen that graphically depict important events in the editing history. A panel depicts an object both before and after one or more editing operations. Using this representation, users can review, edit, and generalize a program. For example, to edit a macro, the user either returns to a point in the history and edits the original drawing or edits the history directly.

Although Pursuit's comic-strip representation language is similar to the editable histories of Chimera, there are many important differences. First, Pursuit's visual language contains *abstractions* that resemble the real interface objects they represent. In contrast, Chimera uses actual screen snapshots in its representations. Also, Pursuit panels contain only the objects affected by the operation, because Pursuit objects can be identified by their icons. On the other hand, Chimera panels contain objects not involved in operations (such as the cursor) in order to provide contextual information to help identify objects and operations.

Furthermore, Pursuit's inferences are displayed in the visual program and are always visible, whereas Chimera's inferences are contained in textual supplements and are not visible in its histories. In addition, Pursuit scripts are two dimensional. Information is conveyed from left to right and top to bottom (see Figure 5). This makes the language more powerful than Chimera, which use only a linear (left to right) display. Finally, Pursuit programs visibly represent loops and conditionals, which are inferred automatically. Chimera macros must be edited to contain loops, which have no explicit representation, and Chimera contains no mechanism for inferring, adding, or representing conditionals.

3.3.3 Other PBD Systems. Mondrian [Lieberman 1992], a demonstrational graphical editor, also uses a similar representation paradigm. Like Chimera, Mondrian produces a storyboard history of user edits. Each pair

of panels of the storyboard represents a single operation and contains a focused snapshot of the screen augmented with other objects needed for context. A program is a one dimensional sequence of panels. The user cannot edit the panels. Instead, Mondrian creates a Lisp representation of the constructed program for the user to edit. Of course, this requires the user to know the syntax and semantics of Lisp, which differs significantly from the syntax and semantics of the storyboard language.

Kidsim [Smith et al. 1994] is a toolkit that enables children to construct and modify simulations by programming their behavior. That system employs graphical rewrite rules and PBD to enable users to create programs for some types of simulations. The graphical rewrite rules are similar to the prologue/epilogue ideas in Pursuit.

4. THE STUDY

In this section we present the details of the evaluation study of Pursuit. The tested versions of Pursuit were implemented in the Garnet toolkit [Myers et al. 1990] running on a Sun SPARC I color workstation with X11 windows. The study took place over a six-week period in the Summer of 1994.

4.1 Subjects

Sixteen subjects were each paid U.S. \$50 to participate in this experiment. Each subject had at least two years experience with 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, music, or history (with B.A.'s in English, music, or history). Subjects were randomly divided into two eight-person groups (with 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.

4.2 Program Categories

For the purpose of this study, we divided programs into two categories: simple and complex. *Simple* programs are straight-line programs containing no explicit loop or branch constructs (e.g., Figures 2 and 4). *Complex* programs contain 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 5 and 7 and Figures 6 and 8).

By dividing programs into these two categories, we were able to determine whether program complexity affected users' abilities to generate and comprehend programs. In addition, we were able to see whether there were any interactions between program complexity and representation language.

4.3 General Procedure

Subjects were tested individually in two sessions. The first session lasted approximately three to four hours. It consisted of a tutorial (about one and three-quarter hours) on Pursuit, which included walk-through construction of four example programs, followed by the program generation task. The comprehension task was presented in a second session the following day. That session lasted approximately two to three hours. Upon completion of the study, subjects filled out a six-page evaluation questionnaire. Subjects then took two tests to evaluate their visual perception and recognition skills.

4.4 Part 1: The Pursuit Tutorial

The first part of the study was a tutorial that introduced subjects to the Pursuit interface, general programming concepts, Programming by Demonstration, and the Pursuit representation language. It provided subjects with the background needed for the remaining three parts of the study.

The tutorial consisted of both written material and a set of interactive exercises. The written material was an 80-page description of Pursuit (the complete tutorials can be found in Appendix A of Modugno [1995]). The first 18 pages of the written material were identical for both groups. They explained all the desktop commands and menus, such as selecting and manipulating files and folders, responding to dialog boxes, and using pull-down menus.

The remaining part of the written material contained four walk-through examples (two simple and two complex) of program construction. These examples 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. These explanations were accompanied by directions that led the subjects through the actual program generation in Pursuit. This helped subjects to learn the Pursuit representation language and interaction techniques as well as the general concepts behind the system.

Each subject received his or her own copy of the written tutorial material and was allowed to make notes in a binder.

The tutorials for both groups differed only in their explanations of program representations. The explanations of general programming concepts were identical. Moreover, *how* the user constructed a program, i.e., the interaction techniques and editing commands, were the same in both tutorials. An analysis of variance of the individual times users spent on each tutorial example revealed that subjects took longer to work through the complex examples than the simple examples: 21.78 minutes versus 17.10 minutes, $F(1,28) = 8.67$, $p < 0.007$. However, the main effect of representation language and the interaction of complexity and representation language were not significant.

Table I. Summary Description of the Generation Tasks

Example	Description	Program Type
1 (simple)	copy report.tex ; rename, compress, and move the copy to the backups folder.	Straight Line
2 (simple)	copy all .tex files; rename, compress, and move copies to the backups folder.	Implicit loop
3 (complex)	same as for 2, but handle the exists... error condition of copy .	Explicit Loop with Branch on Error Conditions
4 (complex)	Same as for 2, but only backup those files modified since last backups were made	Explicit Loop with User-Defined Branch

The tasks are similar to the four walk-through examples the users constructed in the tutorial. The first two tasks are simple because they do not contain an explicit loop or conditional. The last two tasks are complex because they contain both an explicit loop and a conditional.

4.5 Part 2: Program Generation

The purposes of this part of the study were to determine whether users could generate programs in Pursuit and to see whether the representation language had any effect on users' abilities to generate programs.

4.5.1 Part 2's Materials. Materials for Part 2 (all on paper) included a general instruction page, a one-page description of each of the four tasks for which the users were to construct programs, and four brief questionnaires.

The general instruction page explained the form of the task descriptions and the user's goal for this part of the study: to construct programs for the four tasks. The instructions also (1) reminded users that they could refer back to the tutorial at any time and (2) asked them to fill in the questionnaire after completing each task.

The four generation tasks were similar in structure to the four examples in the tutorial (i.e., their programs contained similar program constructs such as sets, loops, error conditions, and user-defined branches). Hence subjects had to generate two simple and two complex programs. Brief descriptions of the four tasks are shown in Table I.

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 figures. The first figure (initial state) depicted the desktop as the user saw it upon beginning the task. The second figure depicted the desktop as it would look if the program the user wrote were executed in the initial state. The user could verify that a program worked by resetting the desktop state, executing the program, and checking to see that the final state matched the second figure shown in the task description. The reset mechanism was a special feature added to Pursuit for the purpose of this study.

After each task, the user filled out a four-question questionnaire. The first question asked the user to rate the difficulty of the task on a scale of 1 (very easy) to 5 (not easy). The remaining questions asked the user to

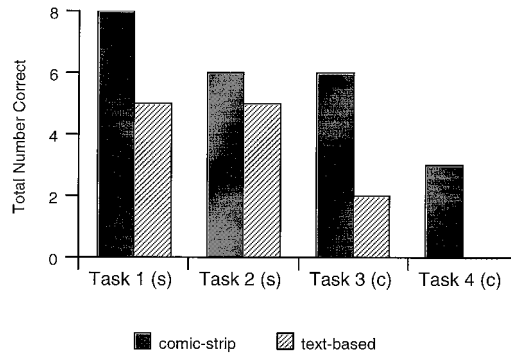


Fig. 9. Accuracy comparison for the generation study. The figure shows a comparison of the total number of subjects (out of eight) who correctly generated a program for each task in the comic-strip group and the text-based group (no user in the text-based group correctly generated the program for task 4). Subjects in the comic-strip group were reliably more accurate in generating programs as those in the text-based group, $F(1,28) = 13.00$, $p < 0.002$. Subjects in both groups were reliably more accurate when constructing simple programs than when constructing complex programs, $F(1,28) = 9.31$, $p < 0.005$. There were no significant interactions between representation language and program complexity.

describe missing features of Pursuit that would be helpful, any information missing from the tutorial that would be useful, and any additional thoughts about Pursuit.

The general instructions, task descriptions, and questionnaires used in this part of the study were identical for both groups.

4.5.2 Part 2's Procedure. Each subject was given a binder containing the generation study instructions followed by the four task descriptions and questionnaires. The binder also contained the Pursuit tutorial so that the user could refer to it at any time during the study.

4.5.3 Part 2's Results. The charts in Figures 9–10 provide a visual summary of the results of the generation study. Figure 9 compares the total number of correctly generated programs per task for both user groups. Users in the comic-strip group outperformed users in the text-based group for all tasks. Figure 10 compares the average time in minutes per task for the two groups. As shown in the figure, there was no correlation between time and user group.

Table II summarizes the percent of programs correctly generated, average time spent on each task, and user rating of task difficulty for each of the two groups. A two-way ANOVA was performed on each of the three measures. Subjects in the comic-strip group were twice as accurate (72% versus 37%) in generating programs as those in the text-based group, $F(1,28) = 13.00$, $p < 0.002$. In addition, subjects in both groups were twice as accurate when constructing simple programs as when constructing complex programs, $F(1,28) = 9.31$, $p < 0.005$. The interaction between representation language and program complexity was not reliable.

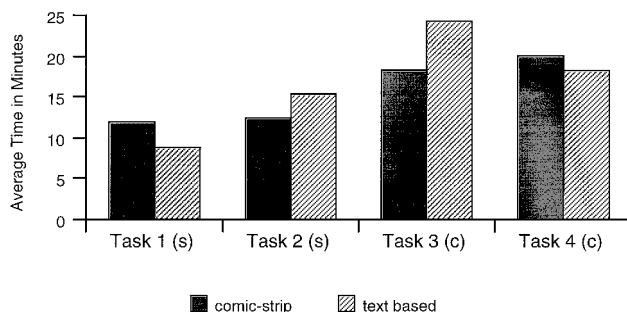


Fig. 10. Time comparison for the generation study. The figure shows a comparison of the average time a subject spent attempting to construct a program for a given task between the comic-strip group and the text-based group. Subjects in both groups were reliably faster when constructing simple programs than complex programs, $F(1,28) = 12.9$, $p < 0.002$. There was no main effect of representation language and no significant interaction between representation language and program complexity.

Table II. Summary Results for the Generation Study

	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

Percent of programs correctly generated (out of two), 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. Users in both groups were significantly faster and more accurate when constructing simple programs and rate simple programs significantly easier to construct. There were no other reliable main effects or interactions.

Not surprisingly, subjects in both groups were reliably faster in constructing the simple programs, $F(1,28) = 12.9$, $p < 0.002$. User ratings indicate that simple programs were easier to construct than complex program, $F(1,28) = 28.9$, $p < 0.0001$. Although the comic-strip group performed faster and rated the tasks somewhat easier, these differences were not significant. There was also no reliable interaction between representation language and program complexity for both time and user rating.

In summary, the comic-strip group was twice as successful in generating programs, but spent no more time on each task.

4.6 Part 3: Program Comprehension

The purpose of this part of the study was to determine whether users could read and understand Pursuit programs. Specifically, we wanted to investigate whether users would be able to handle three situations typically encountered when creating programs:

—“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 (or Pat gave me 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 was asked whether the program implemented the task. If the task was implemented, the subject was given a modified task description and asked to explain how to modify the program to implement the new task. If the program did not implement the task, the subject was asked to explain why. Thus, this part of the study tested whether users could identify whether 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 implement a slightly modified task.

This part of the study focused on areas that might show differences between the representation languages. There are two differences between the languages that might lead to differences in comprehension. First, because the comic-strip language emphasizes the state of data, it could be easier for users to comprehend programs, because tasks are often described in terms of the state of the data (“a copy of the file should be placed in the **backups** folder”) rather than in terms of the actions needed to achieve the task (“copy the file and then move the copy to **backups** folder”). In addition, the state emphasis of that language could enable users to identify more quickly particular types of program bugs, such as incorrect or missing operations, incorrect or missing data objects, and out-of-order operations, because these bugs can be identified by the conflicting state information that they reveal (for example, one cannot move the output of an operation until the operation has executed).

The second difference between the two languages that could affect comprehensibility is the way program control is depicted. The comic-strip language has graphical indications of program flow (i.e., lines connecting panels), whereas the text-based language uses only ordering and indentation to indicate flow. For more complex programs containing loops and conditionals, it might be easier to trace program paths in the comic-strip language than in the text-based language because of the former’s concrete representations of control syntax. For the same reason, bugs in control structure—such as missing branches and paths in the program, operations outside of loops, etc.—could be easier to identify in the comic-strip language.

4.6.1 Part 3’s Materials. Materials for Part 3 included a general instruction page and a brief tutorial to familiarize users with the interface and interaction techniques for the study.

Subjects were shown descriptions of eight tasks: four simple and four complex. For each task description, subjects viewed a series of programs, last of which was the correct one. For two simple and two complex tasks,

the first program was correct. For one simple and one complex task, the second program was correct. For one simple and one complex task, the second program was correct. For the remaining tasks, the third program was correct. Thus, each user made 14 true/false decisions, viewing a total of eight “correct” and six “buggy” programs.

The order of the programs was arranged to control the average position of simple and complex programs in the sequence. Recall that there were two types of simple programs: those that operated over a single file (single) and those that manipulated a file set (set). There were also two types of complex programs: those that contained a branch based on the exit condition of an operation (exit) and those that contained a user-defined branch (user). For each complexity type (simple or complex) there were four tasks with two tasks of each subtype.

To minimize the possible effect of bug type on user comprehension, 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, such as incorrect or missing operations, incorrect or missing data objects, and out-of-order operations. Complex programs contained “structural” bugs, such as missing branches, missing predicates, or incorrectly ordered predicates. By limiting the types of bugs in programs, we hoped to determine whether either representation language facilitated identification of particular bug types.

Similarly, we requested only a limited number of modifications, such as adding and deleting operations, attributes and control constructs, in order to see whether users understood how to modify a program for one task so that it implements another task.

4.6.2 Part 3’s Procedure. Subjects were given a brief tutorial containing three walk-through examples to familiarize them with the interface (Figure 11) 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 into a simple text editor.

Upon completion of this part of the study, subjects were asked to provide a think-aloud protocol of three additional tasks description/program pairs in order for us to *informally* gain insight into their thought processes.

4.6.3 Part 3’s Results. Four analyses were performed on the data for this part of the study. First, we compared the number of users in each group who correctly classified a program for a given task. Users in the comic-strip group performed as well as or better than users in the text-based group on all but three of the 14 tasks, although these differences were not significant in an ANOVA. We also compared the average time in

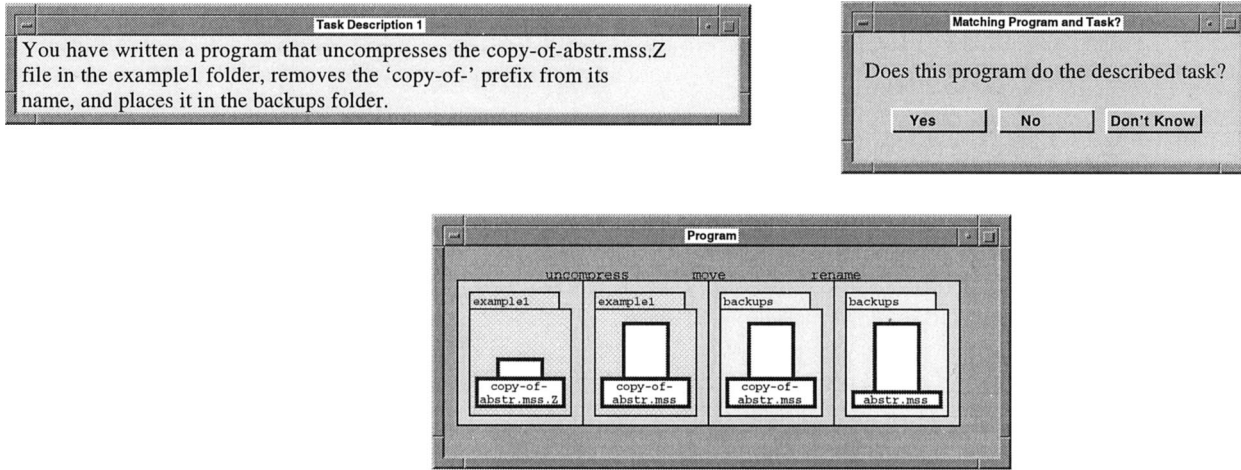


Fig. 11. The user interface for the comprehension study. The figure shows the interface for the comic-strip group in Part 3 of the study. The window in the upper left-hand corner contains a task description. The user indicates whether the program in the lower program window implements the described task by clicking on one of the buttons in the window in the upper right. The interface for the text-based group is the same except for the representation of the program in the program window.

Table III. Summary Results for the Comprehension Study

	Percent Correct		Time in Minutes		Percent of Bugs Found		Percent of Mods Found	
	Comic	Text	Comic	Text	Comic	Text	Comic	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

Percent of programs correctly identified (out of seven), average time in minutes to classify a program, percent of bugs correctly identified (out of three), and percent of correct program modifications identified (out of four). The text-based group was significantly faster, $F(1,28) = 6.44$, $p < 0.02$. There were no other reliable main effects or interactions.

minutes to classify each program. With the exception of one program, users in the text-based group were always faster, and these differences were significant, $F(1,28) = 6.44$, $p < 0.02$. Additionally, we compared the total number of users who correctly identified the bug in each buggy program. Overall, users in the comic-strip group identified more bugs, but this difference was not significant. Finally, we compared the total number of modifications correctly identified for each task. Again, the comic-strip group generally outperformed the text-based group, but the difference was not significant. There were no main effects of program complexity and no interaction between program complexity and representation language in any of the measures.

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

Because of the unequal number of correct programs (8) and buggy programs (6), we performed the comparative statistical analysis based on the *percent* of programs correctly identified per user. We computed each user's percentage as follows:

$$(T_i/8 + F_i/6)/2$$

where T_i is the number of correct true responses (i.e., the number of correct programs that the user said implemented the task), and F_i is the number of correct false responses (i.e., the number of buggy programs that the user said did not implement the task). Therefore, $T_i/8$ is the percent of correct true responses, and $F_i/6$ is the percent of correct false responses. The average of the two numbers is the percent of correct responses. In this way, we adjusted individual scores to minimize biases in random guessing. For example, assume a user always responded that a program implemented a task, without even reading the task. Because of the uneven number of correct and buggy programs, this user would answer eight questions correctly. With a simple accuracy count, the user would score a 57%. However, with the adjusted percent correct, the user would score a 50% ($[8/8 + 0/6]/2$).

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 as 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 correctly identified components of additions and deletions. For bug identification, identifying *any* part of the solution counted as identifying the bug. The maximum number of bugs missed was three for simple tasks and three for complex tasks (there were six buggy programs). For modifications, users had to identify *all* parts of the solution in order to receive a full credit (partial answers were given partial credit). Total modifications missed could be as high as four for both simple and complex programs (one for each program of each type). This is because we wanted to identify whether or not the user knew the program was correct.

In summary, users in both groups did well. The only reliable difference was in speed: the text-based group spent 25% less time classifying a program. Although trends indicate that the comic-strip group consistently outperformed the text-based group, no other main effects or interactions were significant.

4.7 Poststudy Evaluations

After completing the comprehension study, users filled out a short evaluation questionnaire. The purpose of the questionnaire was to help provide insight into the learnability and usability of Pursuit. Moreover, we hoped that the questionnaire would help us gain insight into the intuitiveness of the representation languages. Users were asked to rate both the entire Pursuit system and portions of the system and representation languages on a scale of 1 to 5, where 1 represented very easy, very useful, very intuitive, etc., and 5 represented not easy, not useful, not intuitive, etc.

In general, users in both groups rated Pursuit easy to use, easy to learn, and intuitive. For individual concepts, such as recording a program and programs of a particular type, users in the comic-strip group rated that version of Pursuit somewhat easier to learn. Both users rated simple programs easier to construct than more complex programs. Consistent with the results of the generation study, the comic-strip users rated programs easier to construct than the text-based users.

As a whole, the text-based users rated the representation language somewhat easier to learn and somewhat more intuitive. However, for individual parts, such as the representation of files, folders, operations, etc., users in the comic-strip group rated that representation language somewhat easier to learn and easier to understand. Moreover, for complex constructs such as loops and conditionals, users in the comic-strip group

indicated that the representations more closely matched what they would expect such a concept to look like.

An interesting difference between the two groups was the features of the representation languages that users found most intuitive. In the comic-strip group, users cited branches (four users), loops (two users), and operations (two users) as the most intuitive. Users in the text-based group cited files and folder (five users), declarations (two users), and predicates (one user). Notice that the comic-strip group chose program constructs, while the text-based group chose data items: the only objects in the text-based language that have explicit graphical representations. In general, the results of the evaluation questionnaire supported the findings of the study.

5. DISCUSSION

The purpose of this study was to see whether nonprogrammers 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 nonprogrammers with only two hours of training could write programs like the ones in this study² in any traditional language such as C-shell. Thus, this study demonstrates that combining PBD with an editable program representation does indeed help nonprogrammers 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.

5.1 Program Generation

As shown in Table II, the comic-strip language significantly enhanced users' ability to generate programs accurately in the PBD system. This result is surprising because user actions in constructing and editing programs are absolutely identical across the two program representations. That is, 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 have several hypotheses. First, the comic-strip language more closely reflects the original user interface, so that the users may transfer knowledge of how the interface works to how the programs are constructed. Thus it is really the problem domain that dictates the best representation language. If this were true, then in a version of Pursuit with an interface that was more similar to the text-based language, we would expect the text-based group to perform better.

²The tasks in this study were based on the set of C-shell scripts found in an informal survey at Carnegie Mellon University [Botzum 1995].

Additionally, we cannot eliminate the possibility that the syntax of the particular text-based language we designed was the real cause of the differences. For example, the text-based language contains key words such as **case one of** and **foreach**, which may be too arcane, thus making it difficult for nonprogrammers to comprehend their functionality. Additionally, the layout of the language might make it more difficult to understand. For example, the indentation of nested statements may not have been wide enough to enable the users to sufficiently identify the change in program flow. A “user-friendlier” text-based language might overcome these difficulties and prove to be as good or better than the comic-strip language.

Another reason that the comic-strip group performed better may be because of the graphical representation of control constructs in the comic-strip language. 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. Because 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.

To determine whether the graphical representation of control constructs found in the comic-strip language helps users better conceptualize the concepts they represent, it is necessary to perform a 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.

5.2 Program Comprehension

Although the ANOVAs showed that there were no significant differences between the comic-strip and the text-based 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, $t(1,14) = 1.85$, $p < 0.04$, although a T-test per-

formed on the timing data showed no reliable difference between the groups.³

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 be easier to locate than their corresponding textual counterparts in the text-based language. A way to further explore this hypothesis is to perform an additional within-subject study, similar to the one found in Green et al. [1991], to see whether differences in the microstructures 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 7 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 8.

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

While the last two hypotheses seem reasonable, they directly contradict one result from Part 3 of the study: the *text-based* group was significantly faster in deciding whether or not a program was correct, although that group performed no better or worse in classifying programs. One possible reason for this may be because the program specifications were written in *text*, which is closer in representation to the text-based language than the comic-strip language. Thus, the text-based group required less effort to translate between the problem statement and the actual program representation. The within-subject study of the microstructures of the two languages may provide more insight into this issue.

5.3 Informal Results

In addition to the formal hypotheses tested by this user study, we gathered more informal results 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 Pursuit

³We are careful to mention these results in the discussion because they cannot be interpreted as a firm result. They merely indicate that further investigation is needed.

Table IV. Summary of Incorrect Strategy Selection for the Generation Study

User	Comic Strip Users		Text-Based Users	
	Task 3	Task 4	Task 3	Task 4
1	√*			√
2		√	√	√
3		√	√	√
4	√*		√	√
5		√	√*	√
6	√	√*	√	√
7				
8	√	√	√	√

A check (√) indicates users who initially selected an incorrect strategy to construct a program for Tasks 3 and 4 (the complex tasks). For each task, users who started the task over and then constructed a correct program (using a correct strategy) are denoted with an asterisk. All but one user in each group selected an incorrect strategy at least once. Most users (72%) who selected an incorrect strategy never constructed a correct program. This suggests that Pursuit—and PBD systems in general—needs some way to help users initially select a correct problem-solving strategy.

would be something they would find useful. However, a few subjects expressed difficulty understanding programming concepts, such as when to use a loop or conditional. This was also supported by our pilot studies, which contained minimal explanation of programming concepts and constructs. Those trials showed that users could not learn the minimal skills needed to construct and comprehend programs just by interacting with the PBD system. Instead, they needed some explanation of the concepts. We therefore augmented the tutorial used during the actual study so that it contained explanations, via analogies, of general programming concepts such as iteration and conditional branching. Even with these explanations, which were pilot tested prior to the study, users sometimes had difficulty choosing the correct strategy to demonstrate a program. By strategy we mean how the user needed to demonstrate the program in order for the PBD system to infer the correct procedure. As shown in Table IV, of the 16 users, all but one chose the incorrect strategy at least once during the generation study. In only 18% of these cases did the user eventually create a correct program: *by starting the programming task over and employing another demonstration strategy*. Thus, once a user chose the incorrect strategy, he or she was unlikely to correctly construct a program for the task.

Other users noted that they had difficulty remembering and differentiating between the programming terminology introduced in the tutorial (loop, branch, predicate, etc.), although they understood these concepts. For example, one user pointed out that the word “predicate” was new and confusing. We could have used the word “test” in the same way with (potentially) less confusion. Another user pointed out that he had trouble understanding an exit predicate, and specifically the word “exit,” because to him the word “exit” implied something “final.” He did not seem to understand that the exit predicate described the condition that caused the

operation to (finally) stop executing. Verbal protocols showed that users often invented their own terminology for such concepts (for example, one user referred to a branch as “acknowledging” a particular system state; another referred to declarations as “specifying” files). Further work is needed to understand how to present these terms to users.

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 locate the next instance of the file icon. A brief examination of the unsuccessful attempts from the generation part of the study indicated that users needed more guidance in learning 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 indicated that Pursuit needs much more structure, 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 afterward as the current system requires. If Pursuit were to give such power to the user, it would also need to ensure that user edits did not create a state that made continuing the program demonstration impossible.

6. LESSONS LEARNED FROM A PAPER-AND-PENCIL EVALUATION

Prior to performing the empirical study just discussed, we performed a paper-and-pencil analysis [Modugno et al. 1994] of Pursuit using Cognitive Dimensions [Green 1989; 1991]. The purpose of such an analysis is to provide a fast, more cost-effective way to uncover potential usability problems before embarking on an expensive user study.

Cognitive Dimensions are one of the many evaluation techniques currently being explored by the human-computer interaction community. The Cognitive Dimensions of an information artifact provide a framework for a broad-brush assessment of a system’s form and structure. Green and Petre [1994] have identified twelve Cognitive Dimensions. By analyzing a system along each dimension, the analyst gains insight into the cognitively important aspects of the system’s notation and interaction style and could uncover overlooked usability issues.

After reading several papers on CDs [Green 1989; 1991; Green and Petre 1994], the first author (who is the designer and implementer of Pursuit and at the time had little background in HCI evaluation techniques) performed the analysis in about a day. Both Pursuit prototypes were analyzed along each of the 12 dimensions to determine how far each was from the region of the design space suitable for its intended purposes. In some cases the prototypes scored well; in other cases they scored badly, prompting changes to their designs. Some of these changes were incorporated into the prototypes of Pursuit that were tested in the comparative study. Due to time limitations, however, several of the suggested changes remain only in Pursuit’s design.

Not only did the analysis provide insights into each of the two designs, it also provided a way to characterize the differences between the two representation languages as well as clarified the tradeoffs between these differences. For example, the comic-strip language is more “role expressive,” meaning it more closely reflects what the user sees and manipulates on the desktop. On the other hand, the text-based language is more “terse,” meaning more of it appears in the program window. The cost of greater role expressiveness is less terseness and vice versa.

6.1 The Strategy-Choosing Problem and Solution

A detailed discussion of the analyses can be found in Modugno [1995]. However, it is worth discussing an interesting limitation of Pursuit revealed by the analyses. One of the 12 CDs is *imposed look-ahead*. Imposed look-ahead represents constraints on the order of actions that the system imposes on the user. For example, to select a menu item the user must first expose the menu. Order constraints often require the user to do some planning *before* taking any actions. The more planning involved, the greater the burden (i.e., look-ahead) on the user.

One source of imposed constraints in Pursuit is its model of sets and set manipulation. Consider the simple program shown in Figures 2 and 4 in which the user copies and compresses all the **.tex** files in the **papers** folder that were edited today. Suppose that one of the files originally copied was **abstr.tex** and that the **papers** folder contains a file with the name **copy-of-abstr.tex.Z**. Then the **compress** operation will fail on that set member. Thus, the user’s “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 the user must demonstrate the program with two examples, one in which the **compress** operation succeeds and one in which it fails, so that Pursuit can infer the explicit loop (this is precisely how the program in Figures 5 and 6 is demonstrated). This places a large lookahead constraint on the user, because for very long programs involving multiple operations in multiple folders the user must carefully inspect the system’s state and remember various data states and operation outputs. Such a search would most likely exceed the user’s working-memory capacity. Note that this problem is independent of the representation language.

A similar problem arises when the user demonstrates a set of operations on a file set and afterward realizes that the set selection criteria cannot be expressed via the set attributes, but must be tested explicitly by a user-defined predicate. Imagine the frustration as the user exclaims “Darn, I should have used only a single file!”. That is, to correctly demonstrate the program, the user must demonstrate it on a single data object, edit the program to add a branch, and finally wrap it in a loop. This requires that the user completely understand beforehand how to express selection criteria. Notice again that this problem is independent of the representation language.

Table V. Summary of Users Employing Automatic Loop Conversion

User	Comic Strip Task 3	Text Based Task 3
1	√*	√
2	√*	
3	√*	
4		
5	√*	√*
6		√
7	√*	√
8		

When attempting Task 3, nine of the 16 users (denoted by a checkmark) accidentally happened upon Pursuit's feature that automatically converts an implicit loop to an explicit loop with multiple exit branches (it was not documented in the tutorial). The feature was added after completing the CD analysis described in Modugno et al. [1994]. Users that successfully constructed the program are denoted with an asterisk. When using automatic loop conversion, 67% of users successfully completed the program as compared to an 18% recovery rate in general (Table IV). This illustrated the utility of performing such an analysis prior to conducting actual user trials.

These two examples illustrate the problem imposed by the Pursuit model of set manipulation: users are forced to determine a priori *how* to demonstrate the program. That is, the user must determine the specification strategy, which involves thoroughly examining the state of the desktop and inferring state changes that may result from intermediate program actions. We refer to this problem as the *Strategy-Choosing Problem*. As we discussed above, the user studies brought to light just how difficult a problem this is for users: almost every user chose the wrong strategy for at least one of the tasks in the construction study.

To address this strategy-choosing problem, we added an automatic conversion mechanism to both versions of Pursuit after uncovering it with the CD analysis. Whenever an operation applied to a set has different outcomes for different set members, Pursuit (both versions) automatically converts the single implicit loop to an explicit loop whose loop iteration set is the data set for the original operations. The user can then illustrate the two (or more) different paths of the program by demonstrating the subsequent operations on single members of the set. This feature proved extremely useful during the user study. More than half of the subjects *accidentally* happened upon it (it was not a documented feature of Pursuit), and most of them went on to successfully demonstrate the program. Table V summarizes this result.

This finding is interesting because, as indicated by Table IV, once a user chose the incorrect demonstration strategy, he or she was unlikely to construct a correct program. However, when a user incorrectly chose a strategy that caused him/her to encounter the automatic loop decomposition, the user was very likely to construct a correct program. That is, 67% of students successfully recovered from an error after stumbling on the feature, compared to an 18% recovery rate more generally. This suggests

that the Cognitive Dimension analysis had a noticeable effect on the usability of Pursuit.

The automatic conversion of an implicit loop to an explicit loop reduces lookahead because the user is less constrained to examine the system state or to fully know how to express certain selection criteria before a demonstration. If the user does choose the wrong strategy in certain cases, Pursuit will automatically change it.

6.2 Relating the Two Evaluations

The strategy-choosing problem and the impact of the mechanism we incorporated into Pursuit as a result of the CD analysis attest to the utility of performing a paper-and-pencils evaluation prior to undertaking the more costly (in terms of time and money) user studies. Indeed the additional cost of a few days of reading about CDs and applying them to Pursuit more than paid off in the increased success of the system, which enabled us to gain further insight into other problems of Pursuit.

Moreover, although we uncovered the strategy-choosing problem by analyzing Pursuit, similar problems can be found in other demonstrational systems. This suggests that designers of PBD systems need to consider ways to assist users with strategy selection. That such support is possible was demonstrated by the successful mechanism we incorporated into Pursuit. That such support is not easy was demonstrated by the fact that our solution was not 100% successful. Further study and design are needed to address this problem.

In addition, the CD analysis influenced *how* we analyzed the empirical data. Because the CD analysis revealed the strategy-choosing problem, we looked for confirmatory evidence in the data logs. We might not have looked for this problem otherwise. We thus might have missed a stumbling block for users (at least prior to the empirical study), might have incurred greater cost to fix it after the study in terms of additional user testing, and would not have learned as much from the user study. Moreover, the analysis of the data logs as a result of the CD analysis not only showed us how our solution to the strategy-choosing problem helped users, it also revealed particular instances where it failed and suggested future research into devising mechanisms for handling different types of strategy-selection problems in demonstrational systems.

Finally, the CD analysis was not only useful for revealing potential usability problems prior to the user studies, but also for understanding different design tradeoffs and their potential impact on usability and our study results. For example, the difference in role expressiveness and terseness between the representation languages is a plausible explanation for the success of the comic-strip language over the text-based language in the program generation study. We can test this hypothesis with further studies.

7. CONCLUSION

We have designed, prototyped, and evaluated two 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 this language is represented by *what* it does to data. On the other hand, the text-based language used more textual components to depict the *actions* of a program. In the text-based language, a program is represented by *how* it manipulates data.

The empirical evaluation showed that Pursuit met its goal of enabling nonprogrammers 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 than the essentially textual language in helping users to construct programs. There was also a trend suggesting that the graphical language improved comprehension. These results suggest that this paradigm for visual languages—namely, the 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 they warrant 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 effect users' success.

Finally, the empirical study validated some of the finding of the paper-and-pencil evaluation of the prototypes and points to the utility of such an evaluation prior to embarking on a more costly user study.

ACKNOWLEDGMENTS

We gratefully acknowledge the anonymous reviewers, whose comments greatly improved the content and presentation of this article.

REFERENCES

- BORG, K. 1989. Visual programming and UNIX. In the *IEEE Workshop on Visual Languages*. IEEE, New York, 74–79.
- BOTZUM, K. 1995. An empirical study of shell programs. Tech. Rep., 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: 2nd Workshop* (Washington, D.C., Dec.). G. M. Olson, S. Sheppard, and E. Soloway, Eds. 114–131.
- CYPHER, A. 1991. EAGER: Programming repetitive tasks by example. In *Proceedings of CHI '91* (New Orleans, La., Apr.). ACM, New York, 33–40.
- CYPHER, A. 1993. *Watch What I Do: Programming by Demonstration*. The MIT Press, Cambridge, Mass.
- GREEN, T. 1989. Cognitive dimensions of notations. In *People and Computers V*, A. Sutcliffe and L. Macaulay, Eds. Cambridge University Press, Cambridge, Mass., 443–460.

- GREEN, T. 1991. Describing information artifacts with cognitive dimensions and structure maps. In *People and Computers VI*, D. Diaper and N. Hammonds, Eds. Cambridge University Press, Cambridge, Mass., 297–316.
- 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: 4th Workshop*. Ablex, Norwood, N.J.
- GREEN, T. R. G. AND PETRE, M. 1994. Cognitive dimensions as discussion tools for programming language design. Tech. Rep., M. R. C. Applied Psychology Unit, Cambridge, England.
- HAEBERLI, P. 1988. ConMan: A visual programming language for interactive graphics. In *ACM SIGGRAPH (Atlanta, Ga.)*. ACM, New York, 103–111.
- HALBERT, D. C. 1984. Programming by example. Ph.D. thesis, Computer Science Division, Univ. of California, Berkeley, Calif.
- HENRY, T. R. AND HUDSON, S. E. 1988. Squish: A graphical shell for UNIX. In *Graphics Interface*. 43–49.
- JOVANOVIC, B. AND FOLEY, J. D. 1986. A simple graphics interface to UNIX. Tech. Rep. GWU-IIST-86-23. Inst. for Information Science and Technology, The George Washington Univ., Washington, D.C.
- KURLANDER, D. AND FEINER, S. 1988. Editable graphical histories. In the *IEEE Workshop on Visual Languages (Pittsburgh, Pa., Oct.)*. IEEE, New York, 127–134.
- LIEBERMAN, H. 1992. Dominoes and storyboards: Beyond “Icons On Strings”. In the *IEEE Workshop on Visual Languages*. IEEE, New York.
- MAULSBY, D. L., GREENBERG, S., AND MANDER, R. 1993. Prototyping an intelligent agent through Wizard of Oz. In *Proceedings of InterCHI '93*. 277–284.
- MAULSBY, D. L. AND WITTEN, I. H. 1989. Inducing programs in a direct-manipulation environment. In *Proceedings of CHI '89 (Austin, Tex., Apr.)*. ACM, New York, 57–62.
- MODUGNO, F. 1995. Extending end-user programming in a visual shell with Programming by Demonstration and graphical language techniques. Ph.D. thesis, Tech. Rep. CMU-CS-95-130. Carnegie Mellon Univ., Pittsburgh, Pa.
- MODUGNO, F., GREEN, T., AND MYERS, B. A. 1994. Visual programming in a visual domain: A case study of cognitive dimensions. In *Proceedings of Human Computer Interaction '94*.
- MOHER, T. B., MAK, D. C., BLUMENTHAL, B., AND LEVENTHAL, L. M. 1993. Comparing the comprehensibility of textual and graphical programs: The case of Petri Nets. In *Empirical Studies of Programmers: 5th Workshop*. Ablex, Norwood, N.J.
- MYERS, B. A. 1988. *Creating User Interfaces by Demonstration*. Academic Press, Boston, Mass.
- MYERS, B. A. 1992. Demonstrational interfaces: A step beyond direct manipulation. *IEEE Comput.* 25, 8 (Aug.), 61–73.
- MYERS, B. A., GUISE, D., DANNENBERG, R., VANDER ZANDEN, B., KOSBIE, D., PERVIN, E., MICKISH, A., AND MARCHAL, P. 1990. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Comput.* 23, 11 (Nov.), 71–85.
- PETRE, M. 1995. Why looking isn’t always seeing: Readership skills and graphical programming. *Commun. ACM* 38, 6 (June), 33–44.
- SHNEIDERMAN, B. 1983. Direct manipulation: A step beyond programming languages. *Computer* 16, 8 (Aug.), 57–69.
- SMITH, D., CYPHER, A., AND SPOHRER, J. 1994. Kidsim: Programming agents without a programming language. *Commun. ACM* 37, 7, 54–67.
- SMITH, D. C., IRBY, C., KIMBALL, R., AND HARSLEM, E. 1982. Designing the Star user interface. *Byte* 7, 4 (Apr.), 242–287.

Received July 1996; accepted June 1997