Journal of Visual Languages and Computing (1997) **8,** 491–522

Visual Programming in a Visual Shell—A Unified Approach

FRANCESMARY MODUGNO* AND BRAD A. MYERS†

**Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98115, U.S.A., fm@cs.washington.edu, †Human–Computer Interaction Institute, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, U.S.A., bam@cs.cmu.edu*

Received 25 July 1996; revised 11 May 1997; accepted 15 May 1997

Pursuit is a desktop interface designed to enable non-programmers to construct programs that automate routine repetitive tasks in a way that is consistent with the direct manipulation paradigm. Pursuit combines a Programming by Demonstration (PBD) interface with an editable, visual program representation language. The representation language differs from existing visual languages because it explicitly represents data objects and implicitly represents operations by changes to the data objects. In addition, the language provides concise ways to handle and represent error conditions and dialog boxes.

1. Introduction

A DESKTOP interface or 'visual shell' is a direct manipulation interface to a file system. Files and directories are represented as icons and operations are specified by directly manipulating the icons. The Macintosh Finder was the first popular visual shell, and the idea has gained widespread acceptance among end users. The 'desktop' metaphor has spread to a variety of environments, notably PC Windows and even UNIX (e.g. NeXT). For non-programmers, in particular, the illusion of manipulating data objects rather than issuing textual commands makes interacting with computers more concrete and therefore simpler [25].

Unfortunately, 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, e.g. in macro languages like QuicKeys[2] [3], 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.

Pursuit [18] is a visual shell aimed at providing end-user programming capabilities, especially to non-programmers, *in a way that is consistent with the direct manipulation paradigm.*

Pursuit achieves its goal by combining Programming by Demonstration with an editable, graphical representation of programs. As discussed in Section 2.1, Programming by Demonstration enables users to specify programs in the same way that they

1045-926X/97/050491+32 $25.00/0/vl970049

normally interact with the system—by direct manipulation. This makes the programming process more concrete. Moreover, by representing the program in a visual language that reflects the objects and actions in the interface, users can transfer knowledge of how an object looks and behaves in the interface to how its representation looks and behaves in a program.

This article describes the design and implementation of Pursuit and explains how Pursuit attempts to provide a unified approach to programming in a visual domain using visual language techniques. Previous papers have described the ideas for an initial language design [21], ways to extend the techniques presented here to handle interface objects other than files and folders [17] and evaluations of the system [19, 20]. This article presents the first comprehensive explanation of the different features of the language and interface, as well as the details of the architecture and implementation. We begin (Section 2) by giving an overview of the main idea behind Pursuit—creating a unified visual shell by combining visual language techniques. Section 3 reviews other systems that have influenced the design of Pursuit. In order for the reader to get an idea of the main features of Pursuit and how the user interacts with the system, Section 4 contains four detailed examples. Sections 5–9 discuss Pursuit's implementation. We conclude with the lessons learned in this research and some suggestions for possible future work directions.

2. The Pursuit Visual Shell—An Overview

To bridge the gap between the concrete, visual notation of the direct manipulation visual shell interface and traditional abstract, textual programming notations, Pursuit combines two visual language techniques: Programming by Demonstration and an editable, graphical program representation.

2.1. A Programming by Demonstration Interface

To enable users to construct file manipulation programs, Pursuit contains a Programming by Demonstration (PBD) system [5]. In PBD environments, the user executes actions on real data and the system infers a general procedure [23]. For example, a Pursuit user can create a simple program to make a compressed backup copy of all .tex files in the papers folder by (1) selecting the existing .tex files (e.g. 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. Pursuit generalizes the demonstration from the specific set of .tex files (a.tex, b.tex, c.tex) to the more general description of *all* .tex files in the papers folder. In addition to generalizing over data objects, Pursuit generalizes over sequences of operations in order to identify loops and conditionals based on the outcome of operations. Section 7 details Pursuit's inferencing capabilities.

Although PBD systems have shown promise in enabling non-programmers to automate tasks (e.g. Eager [4], SmallStar [9] and MetaMouse [16]), they have well-known shortcomings: PBD systems can generalize incorrectly, most contain no static representation of the inferred program, their feedback is often obscure or missing and few provide editing facilities. These limitations pose several problems. First, because PBD systems are heuristically based, inferential ambiguities often arise during program

generation. PBD systems typically resolve these ambiguities by querying the user with questions and answers (e.g. Peridot [22]) or dialog boxes (e.g. MetaMouse [16]) or by highlighting the expected user action (e.g. Eager [4]). However, users sometimes find these interactions disruptive and confusing, and often they simply select the default option [4]. Furthermore, because most PBD systems have no way to represent programs, it is difficult for users to recall a program's function or to share programs with other users. Finally, without an *editable* representation, it is impossible for the user to directly edit the generated code.

2.2. An Editable, Graphical Representation

To address the limitations of PBD, Pursuit represents the evolving program in an editable, visual language *while the user is demonstrating it*. In this way, the user knows 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 editing and saving the program representation, the user can have an artifact to later examine, edit and share.

Pursuit's visual language is unique in that, unlike other visual languages, which explicitly represent operations and leave users to imagine the data in their heads, Pursuit's visual language explicitly represents data objects using icons and implicitly represents operations by the visible changes to data icons.

The Pursuit visual language employs a comic strip metaphor [13] for operations and graphical representations for control structure. Data objects, such as files and folders, are explicitly represented with familiar icons. An operation is represented with two panels: the *prologue* shows the data icons before the operation and the *epilogue* shows the data icons after the operation. The operation is depicted implicitly by the changes to data icons between the prologue and epilogue. Control constructs are represented by graphical objects, such as enclosing rectangles for loops and diverging lines for branches. A program is a series of operation panels concatenated together, along with representations for loops, conditionals, variables and parameters. Essentially, programs are a static representation of the dynamic changes to data objects over time.

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.

3.1. Visual Representation Languages for Visual Shells

The idea of using a graphical representation language for a visual shell is not new. Other systems (e.g. ConMan [8], Squish [10] and Jovanovic and Foley [11]) 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. 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. Finally, programming is done off-line by wiring icons together, which differs from the concrete way users normally interact with the system.

3.1.1. IShell

IShell [1], a design for a unix visual shell, is typical of the dataflow model of visual shell languages. In the IShell visual shell design, the interface contains iconic representations of each application (operation). These icons represent 'machines' to which the user feeds data. In IShell programs, an application is represented by its familiar interface machine icon. To construct a program, the user forms a program graph by connecting application icons.

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 also contributes to the desktop clutter.

3.2. Visual Representation Languages for Programming By Demonstration Systems

Although there are situations in which off-line specification is useful and even preferable, e.g. if the user wants to demonstrate a program for a situation that rarely occurs, there are other times in which Programming by Demonstration [5] can be better. In the visual shell domain, Programming by Demonstration 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—via direct manipulation. Moreover, because data objects are available and easily manipulated in the desired way, the program can be readily demonstrated. Similarly, because the domain is limited, the PBD system is more likely to generalize correctly from the demonstration, thus helping the user with the programming task.

3.2.1. SmallStar

The SmallStar system [9] 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 [26]. 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

while keywords are used to represent operations. The user can generalize the transcript by editing it via a menu of commands.

Although Pursuit is based on SmallStar, 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 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, Pursuit's representation 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.

3.2.2. Chimera

Chimera [13] 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 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 modified 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 generalizations 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. As discussed in Section 4.3, in Pursuit information is conveyed from left to right and top to bottom. This makes the language more powerful than Chimera, which uses 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 Systems

Mondrian [15], 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 [27] is a toolkit that enables children to construct and modify simulations by programming their behavior. It 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. Examples

To illustrate some of the major features of Pursuit, we give four examples of program construction. To construct a program in Pursuit, the user enters record mode and demonstrates the program's functionality on actual pieces of data. That is, the user constructs a program by manipulating icons in the interface the way she normally manipulates them when executing each operation. In Pursuit, this manipulation is very similar to the menus and actions of the Apple Macintosh desktop. The only difference is that during recording, Pursuit makes a transcript of the operations and presents the evolving program in a special program window on the desktop. In addition, Pursuit contains an inference mechanism (discussed in Section 7) that attempts to create a general procedure from the operations by generalizing over both the properties of the data and the sequence of operations.

4.1. Example 1—A Simple Program

The simplest programs that users can write consist of a straight-line (i.e. no loops or conditionals) sequence of operation executed over a single data object. For example, assume the user is editing the report file in the papers folder, and periodically wishes to make a compressed backup copy of the file. To create a program to automatically do this, the user (1) enters record mode by selecting start from the pull-down **Recorder** menu in any window; (2) copies the report file using the copy command from the pull-down **File** menu in the papers folder window; (3) selects, drags and drops the copy-of-report file into the backups folder; (4) compresses the copy-of-report file using the compress command from the **File** pull-down menu, which replaces the original file with the compressed version (copy-of-report.z) and (5) exits record mode by selecting stop from the pull-down **Recorder** menu in any window. Figures 1–3 show the evolving program as the user demonstrates it.

4.1.1. Depicting Operations by Data Changes

Figure 1 illustrates the main idea of the Pursuit visual language: operations are represented implicitly by explicit changes to data icons. These graphical changes reflect the changes to the real data objects in the interface caused by the operation. In the first panel of Figure 1 the papers folder only has one file: report. In the second panel the

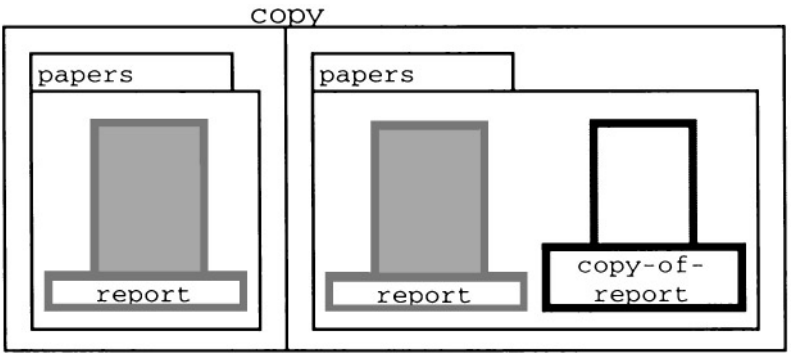

Figure 1. The representation of the operation copy report that appears in the program window after the user executes the copy operation on the report file

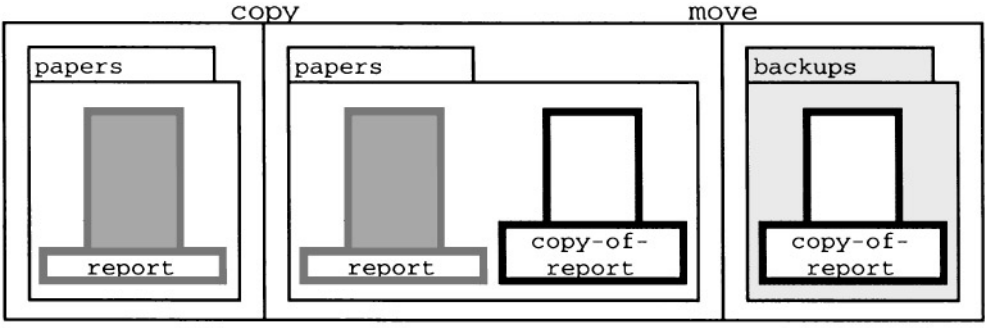

Figure 2. After the user drags (moves) the copy to the backups folder, the third panel appears. Notice that in the visual script the icon for the copy has moved from the papers folder to the backups folder, reflecting the changes the user has seen in the actual interface when the real copy was moved

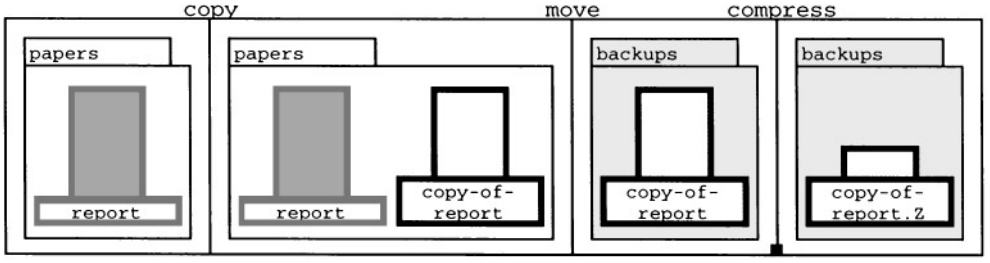

Figure 3. The visual representation of the program to place a compressed copy of the reports file into the backups folder

papers folder has two files: report and copy-of-report. The change in the folder's contents between the two panels implies the copy operation. These differences reflect the change in appearance of the real desktop folder when the copy operation is executed: a new file with the name copy-of-report appears in the papers folder.

4.1.2. Incrementally Displaying Programs

Figures 1–3 illustrate another principle behind Pursuit: programs are displayed incrementally as the user constructs them. In this way, users learn interactively how data, operations and control constructs (i.e. program syntax) are represented, and how these objects fit together to form a meaningful program. Moreover, by interactively constructing the program, the user learns the semantics of the representations by associating them with the corresponding interface actions. For example, demonstrating two or three iterations of a loop helps users understand the semantics of the loop construct. This is important because although users do not need to learn the details of the representation language in order to construct programs, they must still be able to read and understand program representations.

4.1.3. Using Color for Visual Variables

Using icons to represent data has two advantages: icons minimize the use of explicit variables and remove a level of indirection that variables introduce. To identify an icon in a script, Pursuit assigns it a unique color (in this document, color is denoted by different shades of gray). Although an icon's appearance may change throughout the script, its color remains the same. For example, in the second panel of Figure 3 the icon representing the copy is tall, has the name 'copy-of-report' and is in the **papers** folder. In the final panel, the same file is short, has the name 'copy-of-report.Z' and is in the **backups** folder. Users can tell that the two icons represent the same file because they have the same color.

4.1.4. Saving and Parameterizing Programs

After demonstrating a program, the user can save it. The user indicates a program's parameters by clicking on those objects in the program that represent the actual parameters. For example, clicking on the **reports** file in the first panel of Figure 3 indicates that the file that is copied is a parameter to the program. Henceforth, the user can make a backup of any file with the program simply by selecting the desired file and executing the saved program. Saved programs are added to the menu of user-defined programs and can be executed, edited and re-saved, or deleted.

4.2. Example 2—Manipulating a Set of Objects

This next example is similar to the first example, except that instead of manipulating only a single file, the program manipulates a set of files as a whole. The user constructs a program to backup all the **.tex** files in the **papers** folder that were edited today. The user's actions are similar to those taken in Example 1, except that when selecting input to an operation, the user selects a *set* of files and then executes the operation.

Figures 4–6 show the developing visual representation during the demonstration. Figure 4(A) (further explained in the following subsection) is a *visual declaration* and defines the set of files over which the program executes. The files are first copied [Figure 4(B)], then moved to the **backups** folder (Figure 5), and finally compressed (Figure 6).

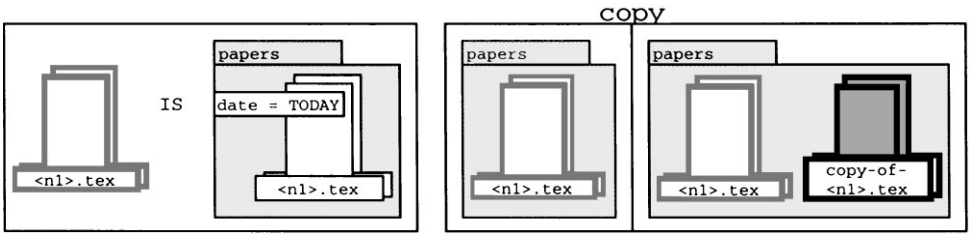

Figure 4. (A) A visual declaration binding the set to be all the .tex files in the papers folder that were edited today. (B) The copy operation

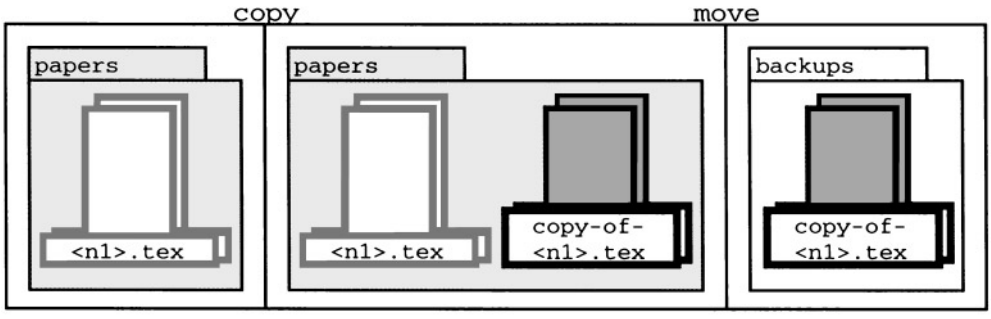

Figure 5. After the user moves the copies to the backups folder, the third panel appears

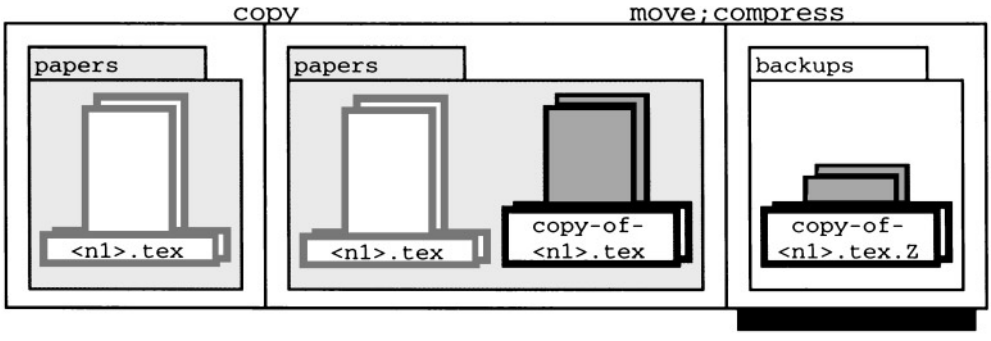

Figure 6. The completed program. The shadow beneath the third panel indicates that it represents multiple operations. As shown in Figure 7, clicking on the shadow reveals the individual operation panels for the move and compress operations

This example also illustrates how Pursuit supports *implicit* set iteration; i.e. iteration without the explicit use of a loop construct. Examples 3 and 4 illustrate explicit set iteration.

4.2.1. Sets and Attributes

Because most shell programs that users write tend to operate over sets of objects related in some specific way [2], the main model of computation of the Pursuit visual language

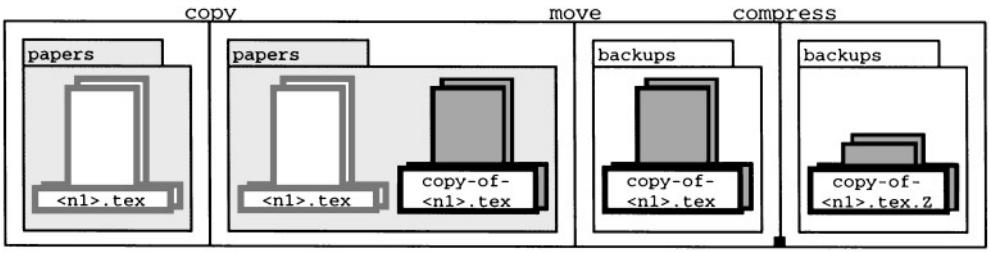

Figure 7. Clicking on the shadow beneath the third panel in Figure 6 reveals the individual operation panels. To recombine the panels, the user clicks on the little black square between them

is the manipulation of *sets* of objects. In Pursuit, sets are represented by overlapping and offsetting two icons of the same type. To define set membership, Pursuit constructs a visual declaration.

In this example, the visual declaration [Figure 4(A)] appears after the user executes the **copy** operation. The icon on the right represents the set of all **.tex** files in the **papers** folder that were edited today. The icon on the left is the icon used in the script to represent this set. The string 'date = TODAY' is an *attribute*. It constrains the set to those files edited today. Attributes allow for abstract sets of objects and indicate the PBD system's generalizations. In addition, users can directly edit attributes to specify desired properties of data objects or to fix incorrect generalizations. Currently, the attributes Pursuit supports include an object's name, date, location, size, owner and contents. Attribute strings can be simple arithmetic expressions defining a single value or a range of values (e.g. '256 < size < 1024') and can contain variables and system constants such as 'TODAY' or 'USER'. Section 7 details how Pursuit uses domain-specific knowledge to generalize attributes.

Attributes and sets reduce the need for loops, conditionals and variables in the language. For example, to define the above set in a traditional programming language, one would have to write code to loop through all the files in the **papers** folder and test to see which ones had names ending in **.tex** and were modified today. In Pursuit, such looping and testing is implicit in the set and attribute notation so that the user does not have to create explicit control constructs. This is important because novice programmers often have difficulty understanding and using loops and conditionals [6].

4.2.2. Space-Saving Heuristics

Because two panels per operation result in long programs that occupy a great deal of screen space, the Pursuit visual language generator contains heuristics to make programs more concise. As detailed in Section 9.1, these heuristics combine knowledge of the domain with information about operations. There are two ways that Pursuit makes programs more space efficient: combining epilogues with prologues and combining multiple operations within a single panel.

Combining Epilogues and Prologues. Pursuit determines when it can combine the prologue of one operation with the epilogue of the previous operation. This eliminates redundant

panels. In Figure 5 only one panel is added to the program because Pursuit notices that the epilogue of the **copy** contains the prologue of the **compress** operation. When the prologue of an operation cannot be combined with the previous epilogue, Pursuit leaves a space between the two panels (an example is shown between the epilogue of the **delete** operation and the prologue of the final **compress** in Figure 10.) Like the spacing between paragraphs in text, the space between two panels in Pursuit's visual language is a visual cue indicating a change in the program's focus. Section 9.2 details how Pursuit determines when to combine a prologue with the previous epilogue.

Combining Multiple Operation Panels. Pursuit also determines when several operations can be represented in a single panel. The shadow beneath the third panel of Figure 6 indicates that it contains both the **move** and **compress** operations. By clicking on it, users can see the individual panels of the operations (Figure 7). In this way, programs are more concise, while users still have full access to the complete representation. Section 9.3 details the mechanisms Pursuit uses to determine when to combine several operations into one panel.

4.3. Example 3—Inferring Loops and Conditionals

Although Example 2 is a valid program, it will work only the first time it is executed. After that, the **backups** folder will always have a compressed copy of all the **.tex** files in the **papers** folder that were edited today. The next time the user executes the program, the **compress** operation may encounter an error condition because **compress** cannot create another file with the same name as an existing file. This error condition will cause the program as written to fail. This section illustrates how Pursuit automatically creates a conditional and loop to handle the described failure.

To handle these types of conditionals, Pursuit uses *explicit* set iteration. Pursuit automatically infers an explicit loop whenever the state conditions, such as those illustrated here, preclude implicit iteration; i.e. whenever two or more members of the iteration set have different outcomes for the same operation thereby requiring two different epilogue panels and potentially two different subsequent program paths.

To construct a program that automatically handles an error condition, the user must first decide what the program should do whenever it encounters such a condition. Let us assume that whenever a compressed copy already exists, the program deletes the old backup copy and makes a new copy to compress. Thus, the program must handle two conditions in the state of the **backups** folder: one in which the **compress** operation executes normally and one in which it fails because of the existence of an old backup copy.

To create this program in Pursuit, the user must demonstrate actions for each of the two conditions that the program may encounter. Pursuit then automatically creates a program (Section 7 explains how Pursuit does this). The user begins by demonstrating the first (no error) condition that the program must handle on a *single* file. After entering record mode, the user (1) copies the **abstr.tex** file; (2) moves the new **copy-of-abstr.tex** file to the **backups** folder and (3) compresses the **copy-of-abstr.tex** file. Figure 8 illustrates the completed first iteration.

To demonstrate the second (error) condition, the user (4) copies the **biblio.tex** file; (5) moves the new **copy-of-biblio.tex** file to the **backups** folder; and (6) compresses the

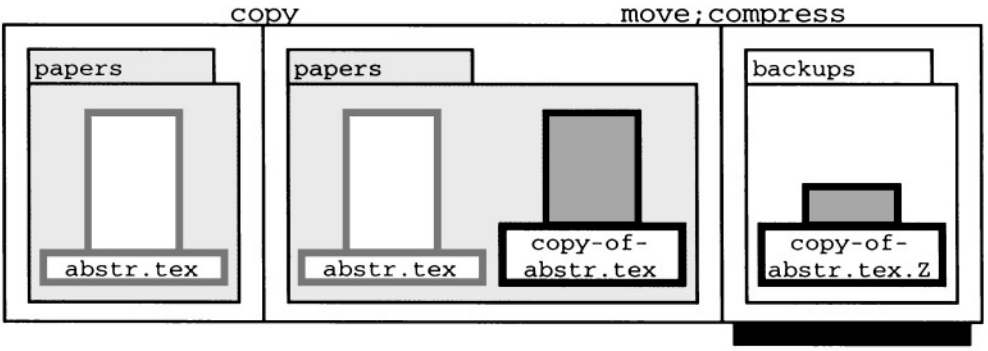

Figure 8. The first loop iteration of Example 3. The user begins by demonstrating a single iteration of the loop showing what the program should do when no error is encountered: copy the input file, then move and compress the output (copy) file

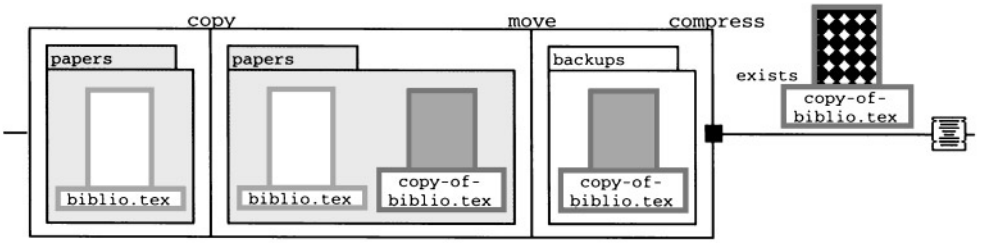

Figure 9. The beginning of the second iteration of Example 3 (due to lack of space, the first iteration, shown in Figure 8 is omitted). The compress operation has failed because of the existence of the file copy-of-biblio.tex.Z. This is indicated by the conditional marker (i.e. the small black square) on the right side of the last panel and the branch connector with the annotation **exists**. . . . The small icon at the end of the annotation represents the dialog box that was displayed when the compress operation failed.

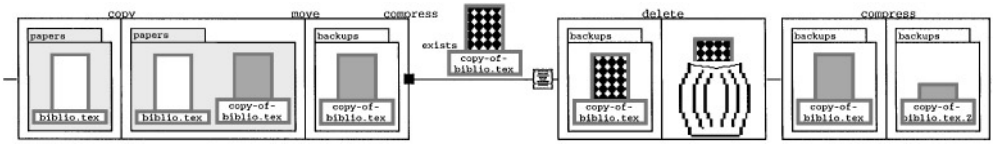

Figure 10. The completed second iteration. The user has demonstrated how the program should handle the encountered error condition: delete the error-causing file and re-execute the compress operation

copy-of-biblio.tex file. At this point the compress operation fails because a file with the name copy-of-biblio.tex.Z—the same name that the output of the compress operation would have—already exists in the backups folder. Figure 9 shows the program at this point.

To finish this case, the user (7) deletes the exiting copy-of-biblio.tex.Z file and (8) re-executes the compress operation. Figure 10 shows the updated program.

To complete the program, the user begins to demonstrate a third iteration on a particular file. Pursuit then detects the loop and queries the user to verify the two loop

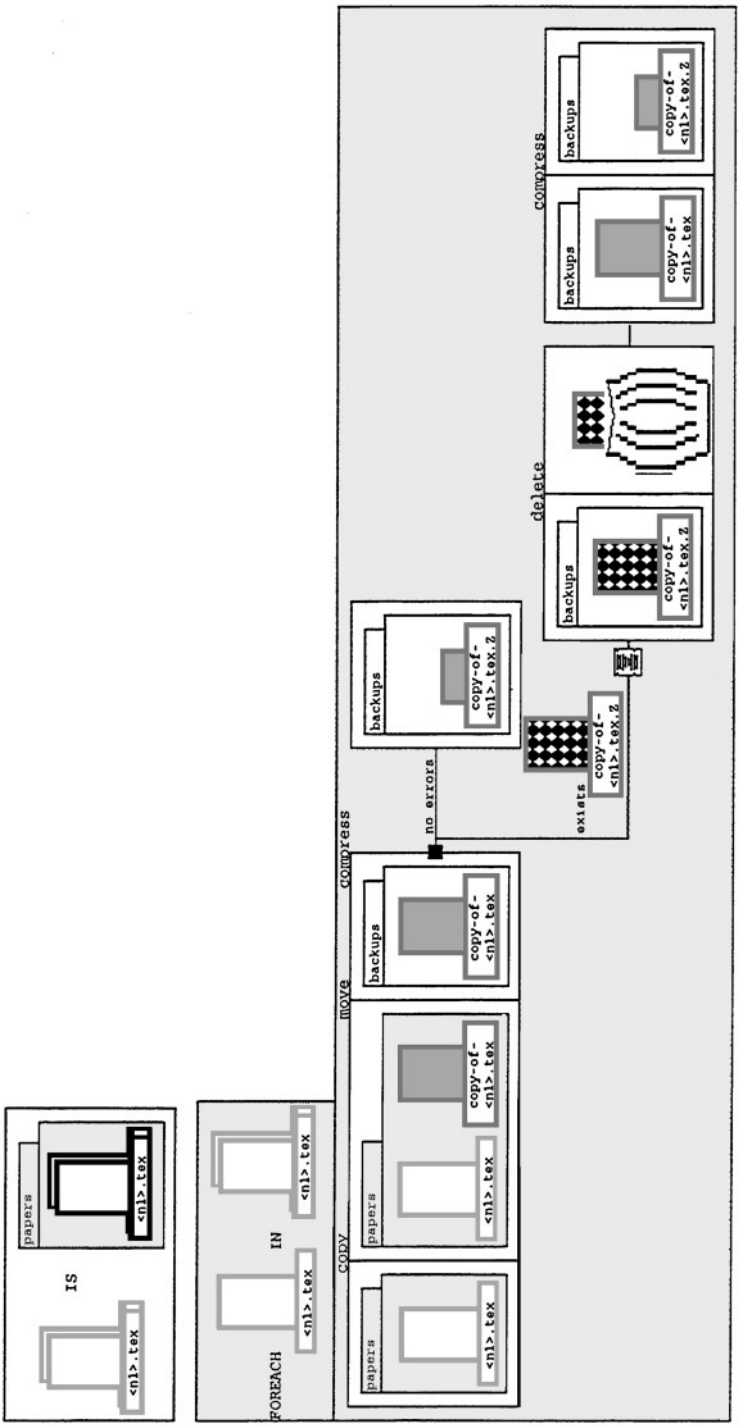

Figure 11. The Pursuit script that places a backup copy each *.**tex** file in the **papers** folder into the **backups** folder. If the **compress** operation fails because of the existence of a file with the output file name, the program deletes that old output file, and re-executes the **compress** operation. Users can see the other possible outcomes of the **compress** operation by clicking on the conditional marker (i.e. the little black square on the prologue of the **compress** operation)

iterations by highlighting the operations panels (i.e. the program panels shown in Figures 8 and 10). After the user verifies the loop iterations, Pursuit pops up a dialog box asking the user to select the members of the set over which to loop. The user selects the desired files by clicking on them in the same way that files are ordinarily selected for any operation. Pursuit then finishes executing the loop and updates the program representation. The final program is shown in Figure 11.

4.3.1. Annotations: Representing Error Conditions

When an operation fails, Pursuit cannot construct an epilogue panel. Instead, it creates a *conditional marker* (i.e. the black square on the right-hand side of the third panel in Figure 9) and an annotation (or predicate) stating the exit condition for that operation. In this example, the **compress** operation failed because a file with the required output name already existed, so the annotation is 'exists' plus a named file icon. To see the remaining exit conditions of the **compress** operation, the user can click on the conditional marker. This helps the user to interactively consider all possible paths a program could take so that, if desired, she can demonstrate what the program should do in each case.

4.3.2. Dialog Boxes: Representing User Interactions

This example also illustrates how Pursuit handles dialog boxes. Applications use dialog boxes to relay messages to the user or to obtain input from the user, such as the name of an output file. When a program contains an operation that uses a dialog box, it is questionable whether or not the user would like this dialog box to appear when the program is subsequently executed, especially if the dialog box simply relays a message. Furthermore, if the information that the user enters into the dialog box can be determined automatically, the user may not wish that dialog box to appear. To address this issue, Pursuit contains a special dialog box manager, which handles the two types of dialog boxes in Pursuit: message-relaying dialog boxes and user-input dialog boxes.

Message-Relaying Dialog Boxes. The contents of dialog boxes that simply relay a message to the user are constant and are determined by the executing operation. Hence, the user need only specify whether or not the dialog box should appear if the program encounters the particular error-causing condition when executing. To enable the user to state this choice, Pursuit pops up a 'meta' dialog box containing the message-relaying dialog box. The meta dialog box requests that the user specify whether the operation's dialog box should appear when the program executes (Figure 12). This interaction occurs during program demonstration, immediately after the user acknowledges the dialog box displayed by the operation.

In the Pursuit visual language, message-relaying dialog boxes are represented by a dialog box icon next to the predicate describing the error-causing condition. An example is shown next to the predicate of Figure 9. Clicking on the dialog box icon displays the Pursuit meta dialog box shown in Figure 12. The user can change the meta dialog box response (and hence whether or not the operation's dialog box appears during program execution) by clicking on the desired button choices.

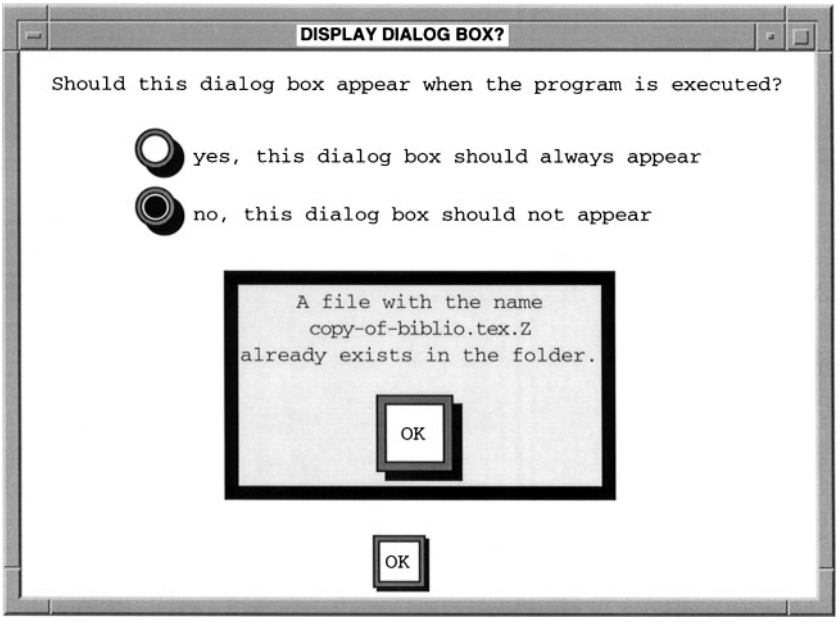

Figure 12. The Pursuit 'meta' dialog box asking the user whether the inner dialog box should be displayed if the compress operation fails when the program executes. This same 'meta' dialog box is displayed when the user clicks on the dialog box icon in Figure 9. The user has indicated that the inner dialog box should not be displayed during program execution

User-Input Dialog Boxes. User-input dialog boxes are more complex than message-relaying dialog boxes because their contents are supplied by the user during program execution and are not predetermined by operations. Because the contents of user-input dialog boxes may not be constant, Pursuit must determine how the input information will be obtained whenever the program executes.

There are three ways to obtain this information. First, the user can enter it during execution. In this case, the user-input dialog box must appear. Second, the information can remain constant for all program executions. Finally, Pursuit can attempt to compute the information during execution using the same heuristics for generalizing data objects from multiple examples (described in Section 7.1). If all the information is computable or constant, then the dialog box need not be displayed during program execution. Otherwise, the user must supply some of the information at runtime.

To determine how to obtain the user input during program execution, Pursuit displays a special input meta dialog box. An example is shown in Figure 13. Using this meta dialog box, Pursuit cycles through all the input fields in the operation's dialog box and asks the user to indicate how the information in that field is to be obtained during program execution.

Like message-relaying dialog boxes, user-input dialog boxes are represented in the Pursuit graphical language as a dialog box icon. However, because input dialog boxes obtain information that serves as input to the operations, the dialog box icon appears in the operation's prologue.

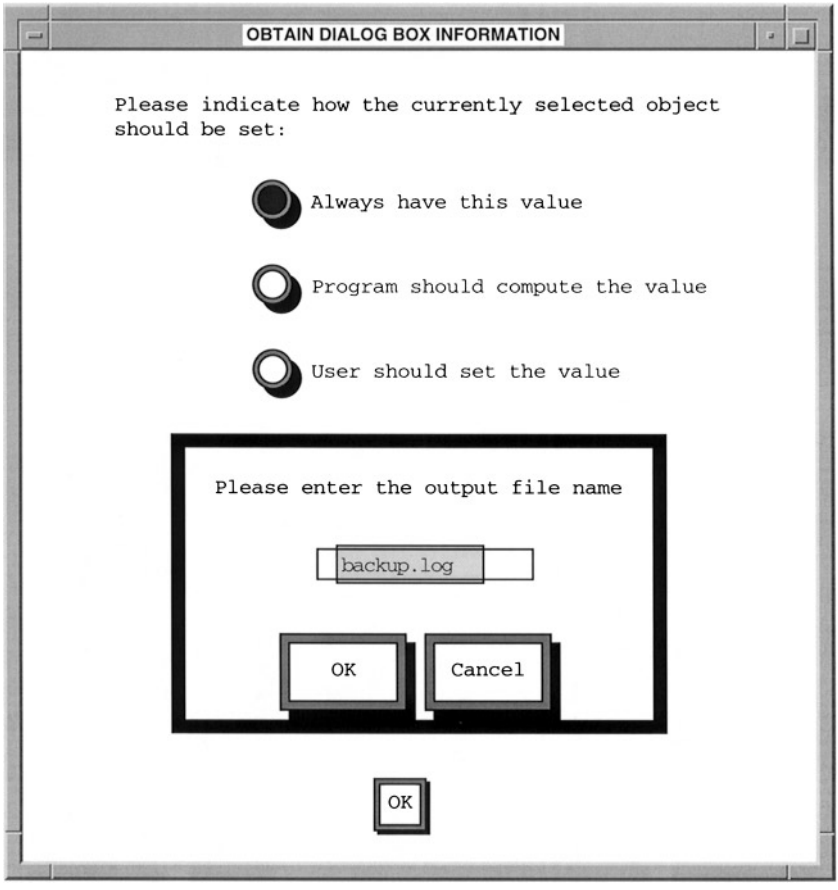

Figure 13. A Pursuit input meta dialog box. Using this dialog box, the user indicates how the input to the enclosed dialog box is obtained during program execution. Each input gadget's color is changed to match the color of the associated radio button at the top of the meta dialog box. This dialog box appears during the program demonstration and can be displayed afterwards by clicking on the dialog box icon contained in the prologue of the operation that displayed the inner dialog box

4.3.3. Loop Constructs: Iterating Over Sets

Figure 11 is an example of an *explicit* loop containing an explicit conditional. The loop is explicit because of the concrete syntactical representation of the loop construct—i.e. the large outer rectangle enclosing the program operations. The declaration in the upper left corner indicates that Pursuit has inferred that the loop iterates over the set of all .tex files in the papers folder. The loop parameter is defined to represent a member of the declaration set (**foreach** *loop parameter* **in** *declaration set*) and the loop operations within the outer loop rectangle are abstractions of the operations demonstrated by the user.

Note that the current version of Pursuit employs the explicit declaration to define the loop iteration set. However, this is unnecessary, and an improvement to the visual language design would be to define the loop iteration set directly in the **foreach** statement, thus removing a level of the indirection in the visual language. Additionally,

removing the indirection would decrease the vertical space used and therefore decrease the empty space in the program window.

4.3.4. Conditional Constructs: Branching Explicitly

Figure 11 also illustrates how Pursuit uses a combination of graphical constructs (boxes and lines) along with annotations and layout to explicitly represent conditionals. However, the dialog box in the annotation is not the same one that the user saw when demonstrating the operation (i.e. the one in Figure 12). Rather it is an abstraction of that dialog box and represents the generalization Pursuit has made for the file set name. Hence, the string displayed is 'A file with the name copy-of-< n1 >.tex.Z already exists in the folder'.

4.4. Example 4—Advanced Editing Features

Although the program in Example 3 adequately handles the error condition that the **compress** operation may encounter, there is another valid way to construct a program to deal with the possible existence of an old backup copy. The program can first delete the old backup copies and then make new compressed ones. However, the program must also handle the situation in which there may not be a backup copy. Thus, the program should execute the **delete** operation only when an old backup copy exists. Otherwise, it should make a compressed backup copy. This final example illustrates how to construct such a program using some advanced editing features of Pursuit. It also illustrates another way that users can construct explicit loops in Pursuit.

First, the user demonstrates one loop iteration (Figure 14): (1) delete **copy-of-biblio.tex.Z** from the **backups** folder; (2) copy **biblio.tex**; (3) move **copy-of-biblio.tex.Z** to **backups** and (4) compress the new **copy-of-biblio.tex**.

Next, the user exits **record** mode and begins to edit the program using the Pursuit editor. The editor is similar to a direct manipulation text editor. Data objects are selected by clicking on them and operations are selected by clicking and dragging the mouse across their panels. Once an object is chosen, appropriate editing commands appear in the **Edit** menu located in the program window.

4.4.1. Constructing User-Defined Branches

First, the user constructs a test for the existence of the file to be deleted. The test is constructed by selecting from a set of pop-up menus (Figures 15 and 16). Similarly, the

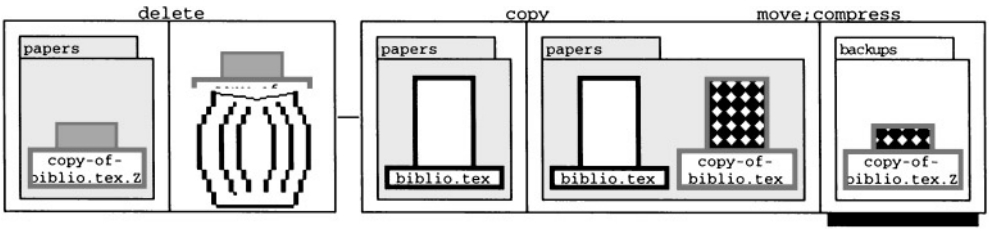

Figure 14. The user begins by demonstrating the main actions of the loop on a single data object

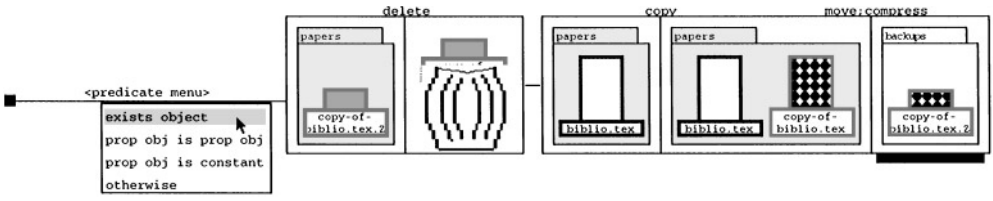

Figure 15. By clicking on the left side of the delete operation, a visual cursor appears (not shown) and the user can add a **predicate menu** (shown). To construct the predicate, the user selects the exists object menu item, which replaces the predicate menu with the predicate-specific menu to check for the existence of an object. (see Figure 16)

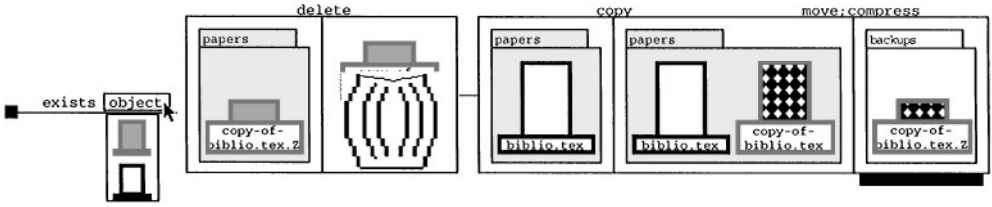

Figure 16. In this case, the predicate-specific menu consists of the word **exists** and the **object** menu. The **object** menu consists of a list of possible data objects. The list contains miniature icons of the data objects available to the user for constructing the predicate. Once the user selects the appropriate data object (the gray-filled icon), the **object** menu is replaced with the program icon (the icon is shown in Figure 17)

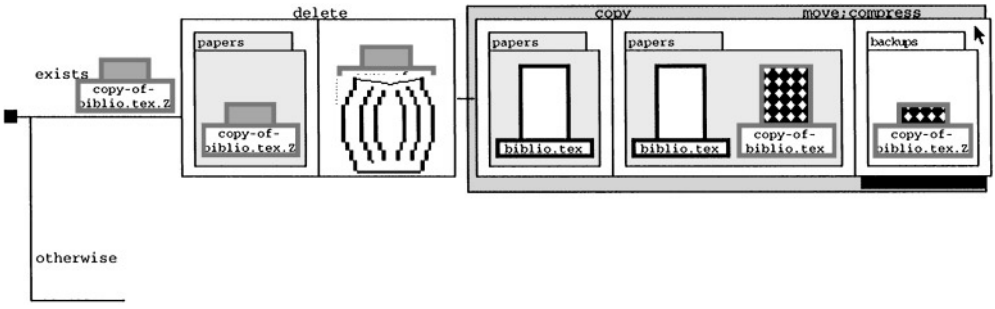

Figure 17. The user selects the operations to copy by dragging the cursor from the first panel of the copy operation to the last panel of the compress operation

user employs the menus to construct a test for the case in which no backup exists. By clicking on the black conditional marker, a new predicate menu appears. As shown in Figure 17, the user selects the Otherwise predicate, because this branch will execute whenever the predicate for the first branch is false (i.e. the backup copy does not already exist).

4.4.2. Cutting, Copying and Pasting Operations

To complete the second branch, the user selects the copy, move and compress operations (Figure 17) and copies them by selecting copy from the **Edit** menu in the program window. She then pastes the copied operations into the program after the otherwise predicate. Figure 18 shows the updated program.

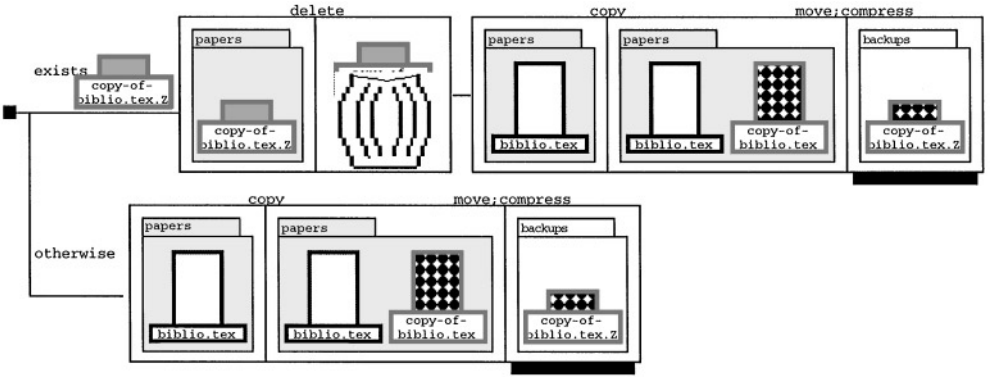

Figure 18. The result of pasting the copied operations after the otherwise predicate

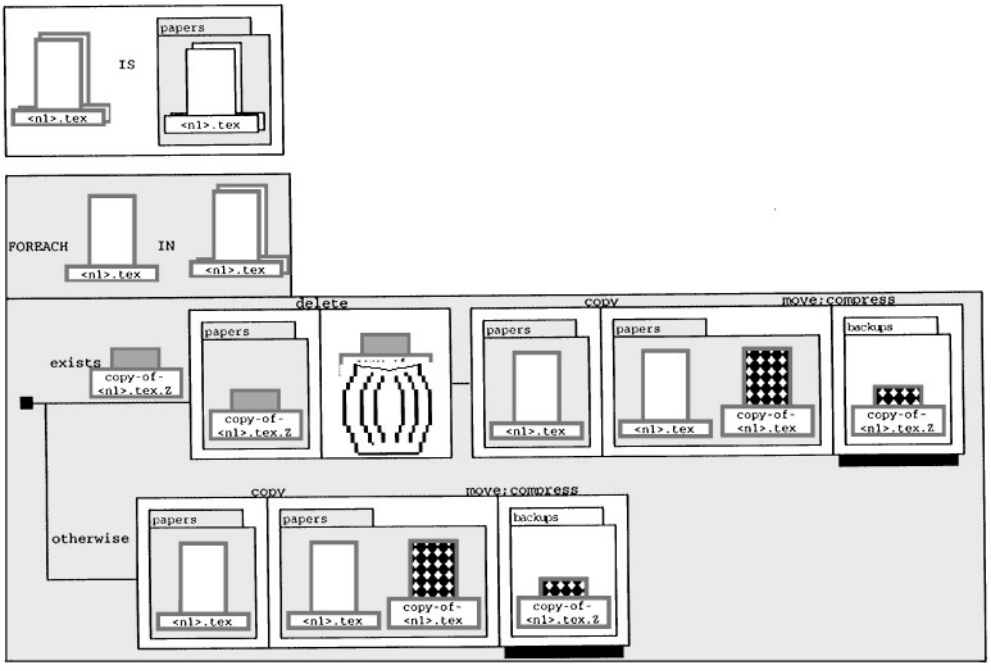

Figure 19. After the user highlights the program in Figure 18, selects Insert into Loop from the **Edit** menu and clicks on any white-filled biblio.tex icon to indicate the loop parameter, Pursuit automatically creates an explicit loop with a user-defined branch

4.4.3. Inserting Operations Into a Loop

Finally, the user wraps the program in a loop. She highlights all the operations, selects Insert into Loop from the **Edit** menu and indicates the loop parameter by clicking on the icon representing the parameter anywhere in the program (in this case, any biblio.tex icon). She then indicates the set members by selecting them in the ordinary way, and Pursuit generalizes the set's attributes. The final program is shown in Figure 19.

4.4.4. Additional Editing Features

Although not mentioned in this example, Pursuit's editor contains additional features. For example, a user can change, add or delete a set's attributes either by directly editing them using Pursuit's text editing commands or by editing the set's property sheet. To help maintain consistency, edits are immediately propagated throughout the program. For example, to make the program in Example 4 work for all .mss files, the user simple changes .tex to .mss in the declaration and Pursuit automatically updates the names of all the other files in the program. This update mechanism is limited to substitutions in text strings, and the updates are propagated to all graphical objects that represent the data item as well as all data objects related to the edited object. Thus, changing a text string does not change a data object.

There are several other features, and we refer the interested reader to the complete description of Pursuit [18].

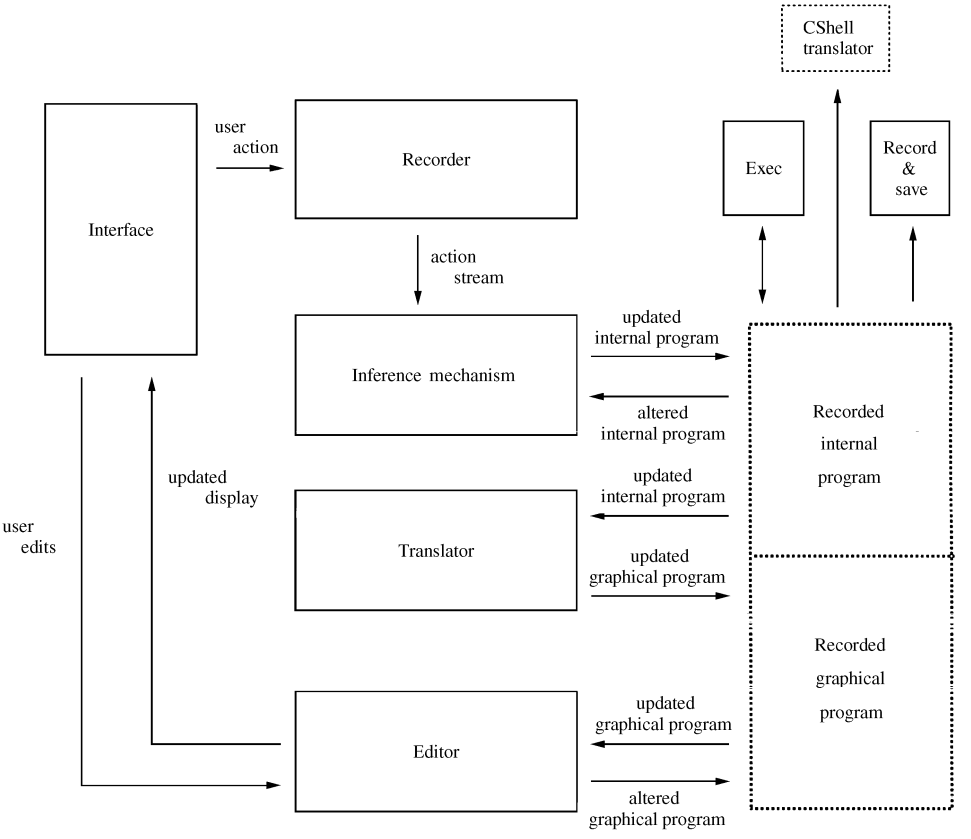

Figure 20. The Pursuit architecture. User actions are recorded by the **Recorder** and generalized by the **Inference Mechanism**. The generalized program is stored in an internal format which is used by the **Translator** to generate the graphical representation. The user edits the graphical representation directly, and the **Editor** automatically updates the graphical program representation, and these changes are propagated to the internal representation

5. Implementing Pursuit—Pursuit's Architecture

Pursuit is implemented in Lisp using the Garnet toolkit [24] and runs on any Sun SPARC workstation or HP workstation containing a Common Lisp environment and running X/11 windows. In this section, we give an overview of Pursuit's architecture and its basic implementation. For specific details, see the complete description of Pursuit [18].

Figure 20 shows the general architecture of Pursuit. As the user demonstrates an action, Pursuit's **Recorder** makes a transcript of the action. Using this transcript, the **Inference Mechanism** attempts to generalize data objects as well as detect sets of repeated program operations. The latter could indicate that the user is demonstrating a loop. The **Inference Mechanism** creates a record of the action in an intermediate representation language. Using this representation, the **Translator** generates a graphical representation of the action, which is displayed to the user. During editing, the user directly edits the graphical representation, and the **Editor** automatically updates both the graphical and internal program representations.

6. Creating an Intermediate Program Representation

When in record mode, Pursuit makes a record of each operation the user executes, and appends it to the growing list of operation records. Internally, Pursuit represents the program as a doubly linked list of operation records. This list represents a state machine that defines the program.

6.1. The Internal Representations of Operations

Figure 21 is an example of an internal representation of an operation record describing a single compress operation. The first field of the record contains the operation name. The second and third fields contain program variable records for the parameters and outputs, respectively, of the operation. Because compress produces no output, the outputs field is empty.

The dialog-box field contains an array of pointers to dialog boxes, which correspond to the dialog boxes for each exit condition of the operation. Pursuit uses information in a dialog-box-info array entry during program execution to determine the contents of the dialog box as well as to determine whether or not to display the dialog box.

The fifth field of the operation record contains a list of (exit-code, operation-record) pairs. This list determines the operation that is executed after compress. In Pursuit, all operations must return an exit code. A zero exit code indicates that the operation

```
operation-record-1
   name:              ' do-compress
   parameters:        program-variable-1
   outputs:
   dialog-box:        dialog-box-info-1
   next:              ((0 operation-record-2))
   previous:          ((operation-record-0 0))
```

Figure 21. A pseudo-code description of the internal representation of a compress operation

```
program-variable-1
  name attribute:                < n1 > .tex
  date attribute:                Jan 21 < date
  size attribute:                5000 < size < 10,000
  owner attribute:
  other attribute:
  initial folder:                papers
  current values:                aa.tex ab.tex ac.tex
  history:
```

Figure 22. The program variable record for all the .tex files in the papers folder that were modified after January 21 and that have a size between 5000 and 10 000 bytes

executed successfully (without errors). A non-zero exit code indicates an error. When a program executes, the next operation to execute is the operation that is paired with the exit code returned by the current operation. In this example, if compress returns successfully, then the operation in operation-record-2 will execute next. Because the next field can contain a list, an operation record implicitly can contain a branch. We refer to a branch based on the exit code of an operation as an *exit branch*. Example 3 in Section 4 illustrated this type of branch. Hence a single compress operation record can cover multiple instances of a compress operation that have been observed in a trace of the user's actions and that represent those compress operations that have been coalesced when forming a loop.

Finally, the previous field of an operation record is a list of (operation-record, exit-code) pairs containing all the operations that immediately precede the compress operation, along with the exit code they return in order for compress to execute next. The previous field can contain a list because Pursuit contains exit branches. When two or more branches merge, the operation at the merge point has multiple predecessors.

6.2. The Internal Representations of Variables

Figure 22 illustrates the internal representation of a program variable. The name, date, size, owner and other attribute fields contain the attributes that Pursuit generalized for the set. Section 7.1 discusses how Pursuit generalizes the attributes of a set. The initial folder slot contains the folder from which the set is initially chosen. The current value field lists the data objects currently in the set.

The history field is used by single object variables (i.e. a file or a folder). It contains a list of (*operation-record, object state vector)* pairs. Every time the user executes an operation that affects the state of the object, Pursuit records the operation's record and the state of the object immediately prior to the operation. As discussed in Section 7.2, Pursuit may use this information when constructing a loop.

7. A Simple Inference Mechanism

To create a general procedure from the user's actions, Pursuit must generalize over both the data the user inputs to operations and the sequence of the operations that the user executes. The former generalization defines why a set of objects is chosen by the user. The latter generalization identifies loops and exit branches in the operation sequence.

To do both types of generalizations, Pursuit contains a very simple inference mechanism. There are three reasons why we chose to use a simple inference mechanism. First, the focus of this research is on representing the demonstrated program and providing editing capabilities to users. We were not interested in ways to improve inferencing. Second, by using a simple inference mechanism we were able to explore how well users would be able to understand the representation and fix inference errors. A system with greater inferencing power might make fewer errors but would not have allowed us to explore the editing issue as deeply. Finally, we designed the system to be independent of the inference mechanism so that advances in that area of research could be incorporated into Pursuit in the future.

7.1. Inferring Data Objects

When the user selects a set of objects to input to an operation, Pursuit attempts to determine why those particular items were chosen. To generalize data sets, Pursuit applies a set of simple heuristic rules. The rules can be divided into two classes: domain heuristics and context heuristics.

7.1.1. Domain Heuristics

Domain heuristics incorporate knowledge of how users typically operate in the visual shell (the domain), as well as knowledge about the behavior of operations. For example, it is common practice to name a file with a prefix followed by an extension indicating the file's type, such as '.c' for C-code files, '.tex' for Latex files, etc. Similarly, the **copy** operation always produces a file whose name begins with 'copy-of-'.

We developed domain heuristics by informally examining a random sampling of shell scripts written by members of the Carnegie Mellon School of Computer Science. We noted how these scripts selected the data to manipulate and used this information to order a set of rules for examining the properties of the members of a set of objects. An interesting observation was that most often users write programs to manipulate a set of files based on a prefix or suffix string in the files' names, such as all **.PS** files. Moreover, rarely did more than one property determine the members of a set.

7.1.2. Context Heuristics

The second class of rules contains context heuristics. These heuristics use information about the particular configuration of the system at the time of the demonstration to identify the set's common properties. Context heuristics suggest an ordering on the domain heuristic rules. For example, if the user has ordered the files in a folder by their date of modification, then Pursuit will first examine the date property of a set to determine whether it adequately defines the set's membership.

7.1.3. Data-Inferring Algorithms

For simplicity, Pursuit generalizes only four properties: name, date of modification, size and owner. For each property, Pursuit first determines whether there is a suitable way for that property alone to describe exactly the members of the data set.

Every set is a subset of all objects within a folder, and hence once the user chooses a set, Pursuit has both positive (the selected files) and negative (the unselected files) examples of set members. For a given data set and property, Pursuit determines how that property describes all members of the set. Once Pursuit determines the property's value, it checks to see if that property is true for any of the negative examples. If so, then the property alone cannot sufficiently define set membership.

When no single property defines the set, Pursuit checks to see whether the conjunction of any pair of properties can define the set. When no pair of properties sufficiently defines the set, then Pursuit checks the conjunction of three-tuples of properties to find a group of properties that exactly defines set membership. Likewise, when no three-tuple of properties defines the set, all four properties together are considered. If this fails, Pursuit uses the name property as a default, and adds an **other** property to the set definition. The **other** property produces a graphical representation of a blank **other** attribute and indicates to the user that she needs to define a way to determine set membership. Currently, Pursuit enables the user to define set membership only by creating conjunctions of the four properties mentioned above.

A limitation of this approach is that it assumes that the user has selected *all* the positive examples and that the remaining examples are only negative. If the user accidently missed a positive example, then Pursuit will not generalize the set correctly. The editing features of Pursuit were incorporated to enable the user to handle these types of situations. An extension of the inferencing system might allow the user to introduce additional positive examples later on so that the system can automatically correct itself.

7.2. Inferring Loops and Conditionals

After recording an operation and inferring over any data objects, Pursuit attempts to identify any loops in the sequence of recorded operations. There are two types of explicit loops that Pursuit can detect: without exit branches and with exit branches.

7.2.1. Explicit Loops Without Exit Branches

An explicit loop without exit branches is a straight-line sequence of operations contained in a loop construct. When Pursuit detects two consecutive, identical subsequences each containing at least two operations, it infers an explicit loop (note that a subsequence of two identical operations with the same outcome is generalized as a single operation over a set). Because each of the two subsequences represents a single loop iteration, Pursuit replaces the subsequences with an explicit loop whose operations are an abstraction of the operations in the repeated sequences.

7.2.2. Explicit Loops With Exit Branches

The second type of explicit loop that Pursuit can detect is a loop containing a branch on the exit conditions of an operation. Example 3 (Figure 11) is an example of such a loop. The techniques Pursuit uses for this type of inferencing are similar to those found in the Tinker system [14].

To detect an explicit loop with exit branches, Pursuit notices a sequence of operations containing two subsequences that begin with the same operation(s) but then diverge

when the outcome of an operation in one subsequence is different from the outcome of the corresponding operation in the second subsequence. To ensure that the two subsequences are two loop iterations, Pursuit first verifies that the parameters of paired operations are the same type (file or folder). In this case, Pursuit infers that the user may be demonstrating a loop containing a branch based on the exit conditions of the operation. An example of such a sequence is shown in Figures 8–10.

To identify the loop parameter, Pursuit selects the input parameter in each iteration that appears most often and is not the output of any operation in the iteration. For instance, in Example 3, Section 4.3, Pursuit infers that abstr.tex and biblio.tex represent the loop parameter because they are both the only input parameters found in both branches. (In the second iteration, copy-of-biblio.tex is also an input parameter, but there is no corresponding input parameter in the first iteration. Hence, Pursuit eliminates it from the set of input parameters that could represent the loop parameters.)

Next, Pursuit determines how the remaining input parameters are chosen: either they are constant, they are the output of an operation in the loop or they are derived from the loop parameter via a simple string-transformation function (e.g. by appending a prefix to the loop parameter's name). Details of the algorithms to determine how parameters are chosen and of the Pursuit string transformation functions are in the complete description of Pursuit [18].

After defining all parameters, Pursuit must generalize over dialog boxes. To accomplish this, Pursuit requires all operations to use the Pursuit dialog box manager. When displaying a dialog box, an operation passes the dialog box and a list of initial values for all its fields to the dialog box manager. This enables Pursuit to record information about the dialog box's contents in the dialog-box field of the operation record (shown in Figure 21). Pursuit abstracts over this recorded information when identifying a loop.

Finally, Pursuit must identify the actual set over which the loop will iterate. There are two possible ways to define the loop set. First, the two representative loop parameters can already be members of an identified set. In this case, Pursuit pops up a dialog box asking the user to verify that this is indeed the set of objects over which to loop.

If there is no set containing the two instantiated loop parameters or if the user does not want to use an existing set, Pursuit highlights the two instantiated loop parameters and asks the user to identify the remaining members of the set. Using its data-inferring algorithm (Section 7.1.3), Pursuit abstracts over the set's members to define the set's properties. However, because a demonstrated iteration might change the relevant properties of the members of a set (e.g. by changing a file's name) in order to make the proper inference, Pursuit examines the state of the set members at the point in time immediately prior to the first demonstrated iteration, which it recorded in the history field of the program variable (discussed in Section 6.2).

Once Pursuit completes all its inferences, it finishes executing the loop and updates both the internal and graphical program representations.

8. The Declarative Specification Language

To generate the graphical representation of operations and to make it easy for application programmers to add new operations to the system, Pursuit contains a declarative language for specifying operations. An operation's specification defines its

```
compress-spec
   inputs:                 A:file
   outputs:                nil
   required inputs:        A
   required properties:    name (A), icon (A)
   changed properties:     name (A), icon (A)
                           contents (A)
exit codes:
   0: predicate:    "no errors"
      actions:      name (A) = name (A) @".z"
                    icon (A) = ^compressed-icon
   1: predicate:    "exists" B where
                    name (B) = name (A) @".z"
                    location (B) = location (A)
      dialog box:   ^compress1-gadget where
                      file-name = name (B)
   2: predicate:    "is-compressed" A
      dialog box:   ^compress2-gadget where
                      file-name = name (A)
   3: predicate:    "is-folder" A
      dialog box:   ^compress3-gadget where
                      file-name = name (A)
```

Figure 23. A pseudo-code representation of the declarative specification of the compress operation.

visual representation, its error conditions and dialog boxes and how it affects the graphical appearance of data objects. During a demonstration, Pursuit uses this specification to generate automatically a representation of the operation in the visual program as well as to generalize over dialog boxes. Pursuit also uses the specification to produce the graphical annotations for operations that encounter an error condition.

Figure 23 shows the specification for the compress operation. An operation's specification has several fields. The inputs and outputs fields contain a list of variables along with their types, (e.g. A: file). These variables represent the inputs and outputs of the operation. The required inputs field lists those inputs that must be present in the representation of an operation. For example, the file that is compressed must be visibly depicted in order for Pursuit to fully represent the compress operation. Inputs that are not required may or may not be represented in the graphical representation of the operation. If the preceding operation contained those inputs in its graphical depiction, then Pursuit will depict those inputs in the current operation's representation. Otherwise, the graphical representation of non-required inputs is omitted.

The required properties field lists those properties of the required inputs that must be present in the graphical depiction of the input. A property of an object is described by its name applied to the variable representing the object. For example, name(A) represents the name of the object represented by variable A. Currently, the properties of an object include its name, location, size, contents and type.

The changed properties field lists the graphical properties of the inputs that are affected by the operation. For example, the changed properties field of the compress operation lists the name, icon and contents of the input file.

The exit codes field lists the possible exit codes that the operation can return. Each exit code lists its associated predicate, dialog box and actions. By default, an operation always returns a zero exit code whenever it executes without problems. The predicate for the zero exit code is always 'no errors,' and the only dialog box that *may* be associated with it is a user-input dialog box.

Non-zero exit codes signal that an operation encountered an error condition. The **predicate** field for error conditions describes how the annotation (predicate) for the error condition is graphically represented. The **dialog box** field points (denoted by the caret) to the message-relaying dialog box displayed by the operation. It also contains a list of the dialog box's fields and how the values of those fields are constructed. For example, the **file-name** field of the **compress1-gadget** states that its value is the same as the name of the error-causing file (whose name is the name of the input file with '.Z' appended). These are the fields that Pursuit uses when determining how to obtain the contents of a dialog box via an input meta dialog box and when abstracting over a dialog box's contents.

The **actions** field of an exit code describes how the operation affects the graphical representation of the input and output icons. For example, the **actions** of the **compress** operation append a '.Z' to the file's name and replace the file's icon with a compressed icon.

Of course, we could not describe every effect of every operation in such a simple way; nor could we graphically depict the effects of some operations with the simple graphics of the visual language. For example, it would be very difficult to graphically display the effects of compiling a file by trying to represent the changes between the source code and object code. Instead, we designed the declarative specification language so that operations could specify some surface characteristics that would provide enough context to graphically depict them.

We close this section by noting that not all operations can be represented easily in this declarative representation language. For example, **delete** requires a special icon for the trash; therefore, when specifying **delete** we had to incorporate a special pointer in its specification. Thus, some operations may require additional programming beyond the declarative specification. Nonetheless, we think that the declarative specification language does capture most of the main operations represented in the desktop metaphor along with the main file manipulation operations found in Unix C-shell and therefore will provide sufficient coverage.

9. Generating Graphical Representations

Pursuit uses the declarative specification of operations to create the visual representation that it displays for the user. This includes creating an operation's prologue and epilogue and possibly an exit branch, as well as combining multiple operations into a single panel in order to save space. Pursuit is the first system to define this automatically from the specification. In previous systems like Chimera [13], the programmer had to determine how to display and combine panels.

9.1. Creating A Prologue

Before creating a prologue, Pursuit determines whether the epilogue of the previous operation can serve as the prologue of the new operation by verifying that all the required inputs and properties of the new operation are found in the epilogue of the previous operation. If the previous epilogue cannot be used as the prologue for the new operation, then Pursuit creates a new prologue.

To generate a new prologue, Pursuit constructs the graphical representation for all the required inputs and their required properties. Pursuit uses a similar procedure to create the location objects (i.e. the folders) of the input variables.

Once Pursuit constructs all the graphical representations for the inputs and their locations, the file and file set icons are added to their location folder icons, and all the folder icons are added to a panel (the outer rectangle) representing the prologue. The panel is then added to the visual program and a gap is added to indicate that the prologue of the new operation is not contained in the epilogue of the previous operation (e.g. panel 3 in Figure 9).

9.2. Creating An Epilogue

If an operation returns zero as its exit code, then there was no error and Pursuit creates the operation's epilogue.

To construct the epilogue, Pursuit copies the operation's prologue and then creates graphical representations for all the outputs of the operation. Since the outputs are newly created by the operation, Pursuit must construct an icon for each output. Once Pursuit creates all the data objects, it updates them by executing all the **actions** for the zero exit condition as specified in the operation's declarative specification.

Like the prologue, Pursuit adds the folder icons representing the locations of the file icons to a new panel. Pursuit then adds the new panel to the graphical program representation, aligning the left side of the panel with the right side of the previous panel so that the epilogue is attached to the prologue, as shown in Figure 1.

9.3. Combining Panels

Pursuit also uses operation specifications to determine when more than one operation can be depicted in the same panel. After Pursuit constructs an operation's epilogue, it determines whether the new operation's epilogue can be combined with the previous operations' epilogues. Pursuit first checks that the prologue panel of the previous operations can be used as the prologue of the new operation (in the same way it checked to see that the epilogue of the previous operation could be used as the prologue of the new operation). If so, it then checks to make sure that the intersection of the changed input properties of all the involved operations is empty. This ensures that no two operations contained in the same panel change the same graphical properties of an object. Thus, each operation's effects are identifiable.

For example, in order for the **compress** operation in Figure 6 to be combined with the **move** operation, the second panel, which is the prologue of the **move** operation, must contain the prologue of **compress** (the file to be compressed) and the set of changed input properties of **move** (the file's location) must be independent of that of **compress** (the file's contents, size and name). Since these conditions are met, the two operations are combined into a single panel. The individual epilogues of the **move** and **compress** operations are stored so that when the user clicks on the shadow beneath the combined panel, the panel is replaced by the two original epilogues. As a second example, the **compress** operation would never be combined with the **rename** operation because both operations change the name of the input file.

9.4. Creating An Exit Branch

When an operation returns a non-zero exit code, Pursuit cannot construct an epilogue. Instead, it constructs a branch with a predicate depicting the error condition encountered by the operation. Usually a predicate consists of a keyword followed by an icon for a file or folder. The word and icon together describe some condition of the system that caused the operation to fail. For example, in Figure 9 the predicate consists of the word **exists** followed by the icon for the **copy-of-biblio.tex.Z** file. The predicate indicates that the existence of the **copy-of-biblio.tex.Z** file caused the **compress** operation to fail.

The information describing the state condition that caused the operation to fail (i.e. the keyword and any description of an object) is contained in the **predicate** slot for the returned exit code. For example, the **predicate** slot for exit code 1 of the **compress** operation specification (Figure 23) contains the keyword 'exists' and the variable **B**. The variable **B** represents the object that is located in the same folder as the input object (A) and whose name is the same as the input object's name with a '.Z' appended.

To construct an exit branch, Pursuit simply adds a conditional marker (i.e. a little black square) to the right edge of the last panel in the program (which is the prologue of the current operation). Then Pursuit constructs the predicate (annotation) for the branch using the information in the **predicate** slot for the exit code found in the operation's specification. The keyword is added to the program after the conditional marker. Finally, Pursuit adds the icon for the error-causing data object, which it obtains from the operation record.

10. Evaluations of Pursuit

This paper has presented the design and implementation details of Pursuit. Other papers detail the evaluations of Pursuit. Briefly, we performed a pencil and paper evaluation of the system [20] using Cognitive Dimensions [7] as well as a user study [19] comparing a version of Pursuit containing the language presented here with a version of the system containing a less graphical language that is similar to the SmallStar [9] language which was tested and shown to be usable by non-programmers. Although conceptually different, the languages in the two Pursuit versions are functionally equivalent. There is a 1-to-1 mapping between their commands and constructs. Moreover, actions to construct programs are identical in both languages, so that the concepts users need to learn do not vary between the languages.

Sixteen non-programmers were randomly assigned to use either the version of Pursuit presented here or the equivalent less graphical version. The user study had two parts: program generation and program comprehension.

In the generation part, users were given four task descriptions and asked to construct programs. The tasks were similar in complexity to the examples shown in this paper. With less than 2 hours of training, users in both groups were able to accurately construct programs more than 70% of the time. However, a two-way ANOVA showed that the group using the more graphical version of Pursuit (presented in this paper) was twice as accurate in generating programs, $F(1, 28) = 13.00$, $p < 0.002$.

In the comprehension part, users were shown a program and a task description and asked if the program implemented the task. Users in both groups correctly identified

a program more than 70% of the time, with the users of the version of Pursuit presented here performing better on more complex programs [$t(14) = 1.84, p < 0.04$].

The user study, as well as the cognitive dimension analysis, confirmed that Pursuit did provide programming capabilities to non-programmers. However, we were surprised to see the great effect that the particular representation language had on users' ability to generate programs, since the user actions in constructing programs are identical across languages. One reason could be the representation of control constructs in the more graphical language. The fact that users of the less graphical version rarely constructed *complex* programs correctly and did much worse for *complex* programs in the other studies suggests that possibly the more graphical representation of loops and conditionals enhanced users' understanding of these concepts. Responses in a post-study questionnaire support this hypothesis. The group using the more graphical version cited branches, loops, and operations as the most intuitive features in the language, whereas the other group cited only files and folders. Further studies are needed to assess this hypothesis.

We wish to state that although the user studies were encouraging, they were not definitive and only suggest that the more graphical approach is promising. The study was small and only covered a limited subset of Pursuit's capabilities. For example, the users did not need to specify parameters for programs once they finished the demonstration. Thus, the study could not provide feedback on the success of Pursuit's approach for parameter specification. Nonetheless, the study does suggest that further research along this path is warranted.

11. Conclusions

Visual shells are easy to use because of the constantly visible, concrete, familiar representations of data objects and the illusion of concretely manipulating these data objects [25]. Unfortunately, this 'conceptual simplicity' is often lost when programming is introduced: users interact with the system visually, but usually program it off-line in a textual programming language. Users must develop two very different bodies of knowledge: one for interacting with the system and one for programming it.

The Pursuit visual shell attempts to bridge this gap. By specifying programs by demonstration and by representing programs in a visual programming language that reflects the desktop, users can apply knowledge of the interface and its objects to the visual language and its objects when constructing, viewing and editing a program.

The purpose of the research behind Pursuit was to investigate a way to enable non-programmers to create file manipulation programs without developing programming expertise. Moreover, we were interested in exploring the relationship of visual languages to Programming by Demonstration systems. As the user studies showed, Pursuit met its goal. In addition, the studies suggested that the more visual language presented here, when compared to a conceptually equivalent but less graphical language that used text for command names and control constructs and visual objects only for data objects, enabled users to construct and comprehend programs more readily. This suggests that more graphical languages may be more successful for PBD systems, especially in the visual shell domain. Hence, this work is an important stepping stone for

using Program by Demonstration and graphical programming language techniques to address the end-user programming problem.

However, the problem is not completely solved. For example, Pursuit's solution can only address PBD systems in which programs are explicitly invoked by the user. There is no support for specifying alternative invocation methods nor for allowing users to specify conditions in which programs will automatically execute. Incorporating Hierarchical Event Histories [12] into Pursuit's record of user actions could address part of this problem.

Additionally, while Pursuit enables users to create programs to manipulate files, it does not enable users to construct programs that manipulate other objects, such as dates. This limits Pursuit's utility. We have recognized this problem and have suggested ways to address it [17]. One such solution is to make all interface objects manipulable and introduce generic visual objects to represent the interface objects within the visual programs. Only additional research can determine whether our solutions are feasible. If so, our work will bring us one step closer to applying the technologies of Programming by Demonstration and graphical programming languages to commercial applications.

Acknowledgments

This research was funded by NSF Grants IRI-9020089 and IRI-9319969, and by grants from the Hertz Foundation and AAUW. This research was performed while the first author was a graduate student at Carnegie Mellon University. We thank the anonymous reviewers for their many very helpful comments that greatly improved the presentation and ideas behind this work.

References

1. K. Borg (1989) Visual programming and UNIX. In: *IEEE Workshop on Visual Languages*, Rome, Italy, pp. 74–79.
2. K. Botzum (1995) An empirical study of shell programs. Technical Report in progress, Bell Communications Research.
3. CE Software, Inc., P.O. Box 65580, West Des Moines, IA 50265. *QuicKeys*[2].
4. A. Cypher (1991) EAGER: programming repetitive tasks by example. In: *Proceedings of CHI '91*, New Orleans, LA, pp. 33–40.
5. A. Cypher (1993) *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA.
6. S. Doane, J. Pellegrino & R. Klatzky (1990) Expertise in a computer operation system: conceptualization and performance. *Human–Computer Interaction* **5,** 267–304.
7. T. R. G. Green (1989) Cognitive dimensions of notations. In: *People and Computers V* (A. Sutchiff & L. Macaulay, eds). Cambridge University Press, Cambridge, pp. 443–460.
8. P. E. Haeberli (1988) ConMan: a visual programming language for interactive graphics. In: *ACM SIGGRAPH 1988,* Atlanta, GA, pp. 103–111.
9. D. Halbert (1984) Programming by example. Ph.D. Thesis, Computer Science Division, University of California, Berkeley, CA.
10. T. Henry & S. Hudson (1988) Squish: a graphical shell for UNIX. In: *Graphics Interface '88*, Edmonton, Albertaca Ca, pp. 43–49.
11. B. Jovanovic & J. D. Foley (1986) A simple graphics interface to UNIX. Technical Report GWU-IIST-86-23, The George Washington University, Institute for Information Science and Technology, Washington, DC 20052.

12. D. Kosbie & B. A. Myers (1994) Extending programming by demonstration with hierarchical event histories. In: LNCS 876. *Proceedings of the East-West International Conference on Human–Computer Interaction*, Springer, New York.
13. D. Kurlander & S. Feiner (1988) Editable graphical histories. In *IEEE Workshop on Visual Languages*, Pittsburgh, PA 15213, pp. 127–134.
14. H. Lieberman (1986) An example based environment for beginning programmers. *Instructional Science* **14,** 277–292.
15. H. Lieberman (1992) Dominoes and storyboards: beyond "icons on strings". In: *IEEE Workshop on Visual Languages.*
16. D. Maulsby & I. Witten (1989) Inducing programs in a direct-manipulation environment. In: *Proceedings of CHI '89,* Austin, Tx, pp. 57–62.
17. F. Modugno (1994) Interface issues in visual shell programing. In: *Visual Object Oriented Programming* (M. Burnett, A. Goldberg & T. Lewis, eds). Prentice-Hall, pp. 95–112. Englewood Cliffs, NJ.
18. F. Modugno (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.
19. F. Modugno, A.T. Corbett & B.A. Myers (1996) Evaluating program representation in a demonstrational visual shell. In: *Empirical Stuides of Programmers 6th Workshop*, Arlington, Va, pp. 131–146.
20. F. Modugno, T.R.G. Green & B.A. Myers (1994) Visual programming in a visual domain: a case study of cognitive dimensions. In: *Proceedings of Human–Computer Interaction '94*, Glasgow, Scotland, pp. 91–108.
21. F. Modugno & B. A. Myers (1993) Graphical representation and feedback in a PBD system. In: Watch What I Do: Programming By Demonstration (A. Cypher, ed.), MIT Press, Cambridge, MA, Chap. 20, pp. 416–422.
22. B. A. Myers (1988) Creating User Interfaces by Demonstration. Academic Press, Boston, MA.
23. B. A. Myers (1992) Demonstrational interfaces: a step beyond direct manipulation. *IEEE Computer* **25,** 61–73.
24. B. A. Myers *et al.* (1990) Garnet: comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer* **23,** 71–85.
25. B. Shneiderman (1983) Direct manipulation: a step beyond programming languages. *Computer* **16**, 57–69.
26. D. C. Smith, C. Irby, R. Kimball & E. Harslem (1982) Designing the star user interface. *Byte* **7**, 242–287.
27. D. C. Smith, A. Cypher & J. Spohrer (1994) Kidsim: programming agents without a programming language. *Communications of the ACM* **37**, 54–67.