In proceedings of IEEE/CS Symposium on Visual Languages. St. Louis, Mo, October, 1994.

A State-Based Visual Language for a Demonstrational Visual Shell

Francesmary Modugno and Brad A. Myers

Carnegie Mellon University
Pittsburgh, Pa 15213
{fmm,bam}@cs.cmu.edu

ABSTRACT

We present a new visual programming language and environment that serves as a form of feedback and representation in a Programming by Demonstration (PBD) system. The language differs from existing visual languages because it explicitly represents data objects and implicitly represents operations by changes in data objects. To improve feedback and provide a closer union between the PBD system and the program representation, the visual language is integrated into other parts of the system, such as the editor. Finally, in order to enable new operations to be added easily to the system, we introduce a declarative language for specifying how an operation's visual representation is generated.

INTRODUCTION

A visual shell (or desktop) is a direct manipulation interface to a file system. Examples include the Apple Macintosh desktop and the Xerox Star. Although such systems are easy to use, most do not support end-user programming. Pursuit[**?**] is a visual shell aimed at providing programming capabilities in a way that is consistent with the direct manipulation paradigm.

To enable users to construct programs, Pursuit contains a Programming by Demonstration (PBD) system[**?**]. With PBD, users execute actions on real data and the underlying system attempts to construct a program[**?**]. Such systems have limitations: they can infer incorrectly, most contain no static representation of the inferred program, feedback is often obscure or missing, and few provide editing facilities. This makes it difficult for users to know if the system has inferred correctly, to correct any errors, and to revise or change a program. Pursuit addresses these problems by presenting the evolving program in a *visual* language *while it is being constructed.*

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

This article describes the major components of Pursuit and explains how Pursuit attempts to provide a unified approach to programming in a visual domain using visual language techniques. It further outlines how Pursuit is implemented so that new operations can easily be added to the system.

SPECIFYING PROGRAMS VISUALLY

To construct a program in Pursuit, users begin by demonstrating its actions on example objects. As the actions are executed, Pursuit's PBD inference mechanism[1] records them and attempts to construct a general procedure. For each operation, Pursuit uses heuristics to determine why the user has chosen a particular set of objects as input. The heuristics combine knowledge of the domain with information about the current state of the system to identify the shared properties of the objects in the set. The heuristics were developed by studying a random selection of shell scripts written by members of the Carnegie Mellon School of Computer Science. We noted how the input to each script was derived and used this information to order a set of rules for Pursuit to use when examining the properties of the members of a set.

The inference mechanism also detects loops by scanning the history for a sequence of one or more repeated actions. In addition, Pursuit can detect a loop containing a branch on the exit conditions of an action by detecting two sequences of actions that begin with the same action(s) but then diverge when the outcome of an action in one sequence is different from the outcome of the corresponding action in the second sequence. An example of this type of loop is discussed in the third example below.

REPRESENTING PROGRAMS VISUALLY

While the user demonstrates a program's actions and the PBD system attempts to construct it, a representation of the evolving program in the Pursuit visual language appears in a special window on the desktop. The Pursuit visual language is based on the comic stripn metaphor[**?**]. Familiar icons are used to represent data objects, such as files and folders. Sets are represented by overlaying two icons of the same type (see Figure 1). Two panels are used to represent an operation. The *prologue* shows the data before the operation and the *epilogue* shows the data after. The operation is represented by the changes to data between the prologue and epilogue. A program ("visual script") is a series of operation panels concatenated together, along with representations for loops, conditionals, variables and parameters. These and other features of Pursuit are illustrated in the following three

[1]This research does not focus on inferencing and so we use a simple pattern matcher. Our research focuses on other limitations of PBD systems: feedback, representation and editing. The techniques described here are independent of the inference mechanism.

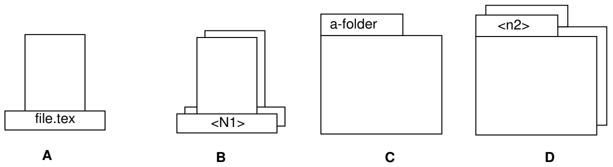

Figure 1: The main data types in Pursuit: a) a file; b) a set of files; c) a folder; d) a set of folders.

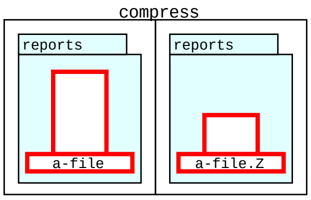

Figure 2: The representation the operation `compress a-file`.

examples. Unfortunately, it is difficult to fully portray the dynamic interaction of the system by a set of static pictures. Video[**?**] provides a greater understanding of the interaction between the user and Pursuit. Hopefully, though, these example will sufficiently convey the ideas within Pursuit.

Example 1

This simple example illustrates the visual representation that results from demonstrating how to compress a file and move it to the `backups` folder. Figures 2–3 show the developing program during the demonstration. Figure 2 shows the first two panels as they appear after the user opens the `reports` folder, selects `a-file` and compresses it. After the user moves the file to `backup` folder, the new panel in Figure 3 appears depicting the move.

Representing Operations by Changes to Data Figure 2 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 2 the icon representing the file `a-file` has the name `a-file` and is a "tall" icon. In the second panel the icon for the file has the name `a-file.Z` and is "short". The change in the file icon's name and height between the two panels imply the `compress` operation. These differences are similar to the change in appearance of the real file icon users see in the actual user interface when the compress operation is executed: the file's icon is replaced with a shorter one and ".Z" is added to its name.

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. Although an icon's appearance may change throughout the script, its color remains the same. For example, in the first panel of Figure 3 the icon representing the file has the name "a-file" and is in the `reports` folder. In the third panel, the same file has the name "a-file.Z" and is in the `backups` folder. Users can tell that the two icons represent the same file because they have the same color. Color serves the same purpose as a variable name in textual programming languages where a variable denotes a data object. Operations manipulate a variable, but affect only the data that the variable represents. The variable name (or its representation) is not changed. However, in Pursuit operations affect the visual representations of icons, which serve as "variables" representing data objects. Color insures that the link between an icon and the data it represents is not lost when the icon's appearance changes.

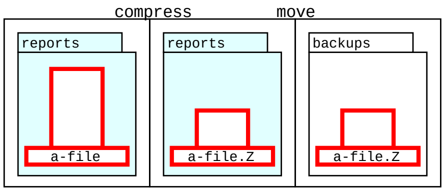

Figure 3: The visual representation of a program to compress the file `a-file` and then move it to the `backups` folder.

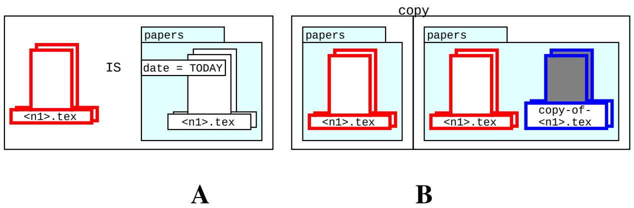

Figure 4: A). A visual declaration binding the set to be all the `.tex` files in the `papers` folder that were edited today. B). The `copy` operation.

Example 2

The next example illustrates how to write a program to backup all the `.tex` files in the `papers` folder that were edited today. Figures 4–6 show the developing visual representation during the demonstration. Figure 4A (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 4B), then moved to the `backups` folder (Figure 5), and finally compressed (Figure 6).

Depicting Abstract Data Objects The visual declaration (Figure 4A) appears after the user executes the `copy` operation. It defines the scope of the set variable. The icon on the right represents the set of all `.tex` files in the `papers` folder that were edited today. The icon on the left is the icon used in the script to represent this set. The string "date = TODAY" is an *attribute*. It constrains the set to those files edited today. Attributes allow for abstract sets of objects and indicate the PBD system's inferences. In addition, users can directly edit attributes to specify desired properties of data objects or to fix incorrect inferences. User defined attributes can also be constructed. Currently, the attributes Pursuit supports include

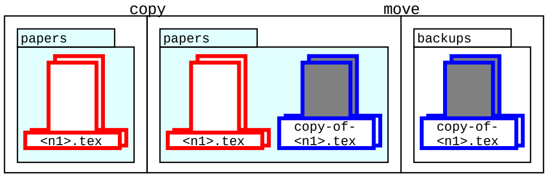

Figure 5: After the user drags (moves) the copy to the `backups` folder, the third panel appears. Notice that in the visual script icon for the copy has moved from the `example1` folder to the `backups` folder, reflecting the changes the user has seen in the actual interface when the real copy was moved.

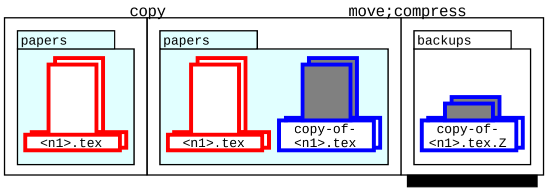

Figure 6: The completed script. The shadow beneath the third panel indicates that it represents multiple operations. Clicking on it reveals the individual operation panels for the `move` and `compress`.

an object's name, date, location, size, owner and contents. Attribute strings can be simple arithmetic expressions defining a single value or a range or values (e.g. "256 $<$ size $<$ 1024") and can contain variables and system constants such as "TODAY" or "USER."

Attributes and sets minimize the need for loops, conditionals and variables. For example, to define the above set in a traditional programming language, one would have to write code to loop through all the files in the `papers` folder and test to see which ones had names ending in `.tex` and were modified today.

Automatically Combining Operation Panels Because two panels per operation result in long, space inefficient scripts, the Pursuit visual language generator contains heuristics to make programs more concise. These heuristics combine knowledge of the domain with information about operations (further described below). For example, Pursuit determines when it can combine the prologue of one operation with the epilogue of the previous operation. In Figure 5 only one panel is added because Pursuit notices that the epilogue of the `copy` contains the prologue of the `move` operation. Pursuit also uses these heuristics to determine 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 for the `move` and `compress` operations.

Example 3

The final example illustrates how Pursuit automatically creates conditionals and loops. Assume a user wishes to write a program to copy all the `.tex` files in the `papers` folder. This could be done by demonstrating the program as in the previous example. However, suppose the folder contains a file named `copy-of-biblio.tex`. If `copy` is executed on any set containing the `biblio` file, it will fail because the `copy` operation must produce an output file called `copy-of-biblio.tex`. Therefore, the user must demonstrate this program on *two* examples – one in which the `copy` operation succeeds, and one in which it fails. In the latter case, the user must demonstrate what the program should do: delete the error causing file and re-execute the copy operation. Figure 7 shows the visual script after the second example is demonstrated.

Branches on Exit 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 side of the third panel in Figure 7) and a *branch connector* with an annotation (or predicate) stating the condition for that branch. In this example, the `copy` operation failed because a file with the required output name already existed, so the annotation is "exists" plus a named file icon. To see the remaining exit conditions of the `copy` operation, users can click on its conditional marker. This helps users interactively consider all possible paths a program could take so that they can demonstrate what the program should do in each case, if desired.

Dialog Boxes: Communicating with Users This example also illustrates how Pursuit handles dialog boxes. Applications use dialog boxes to relay messages or obtain information, such as the name of an output file. 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 is purely informational. In Pursuit, applications use a special Pursuit mechanism (described below) to display dialog boxes. If a dialog box appears when recording a program, Pursuit tries to determine if it can compute the user's response automatically, so that when the program is executed, no user intervention would be required. Pursuit then pops up a "meta" dialog box asking the user to confirm the inferences (if there are any) and to state whether or not the dialog box should appear whenever the program is executed (see Figure 8). To allow users to view or edit a dialog box response, an icon representing the dialog box is included in the script. An example is shown on the branch of the `copy` operation in Figure 7. By clicking on the icon users can see the Pursuit meta dialog box containing both an abstraction of the dialog box that appears when the `copy` operation fails.

The previous example illustrated how Pursuit handles dialog boxes that relay messages to users. Pursuit uses these same abstraction and inference mechanisms for dialog boxes that obtain information from users. When recording a program, if such a dialog box appears, Pursuit pops up a "meta" dialog box asking users how the information for each field of the

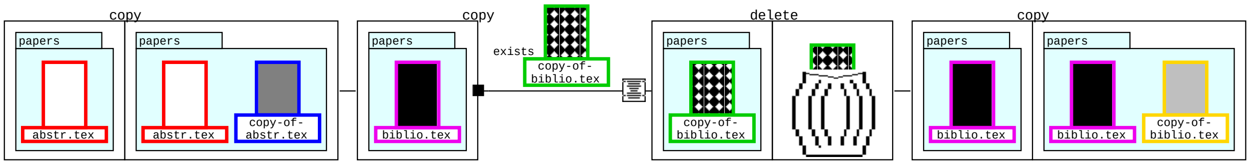

Figure 7: The two examples used to demonstrate a conditional program. The first two panels show a successful execution of the `copy` operation. The remainder of the script shows the unsuccessful execution and corrective actions the user wants the program to take in that case. The black square on the prologue of the second `copy` operation (panel 3) indicates that the operation failed. The predicate following explains why – the existence of a file with the name `copy-of-biblio.tex`. The dialog box icon indicates that the operation popped up a dialog box to the user. Clicking on the icon pops up the dialog box displayed when the operation failed. To correct the operation, the user deletes the error causing file (panels 4 and 5) and re-executes the copy operation (panels 6 and 7).

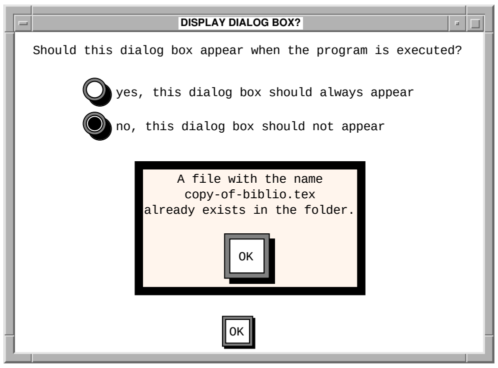

Figure 8: The Pursuit "meta" dialog box displayed when the user clicks on the dialog box icon in Figure 7. The user has indicated that the inner dialog box should not be displayed if the `copy` operation fails during program execution.

dialog box is to be obtained when the program is executed: by being computed by the program, entered by the user, or remaining constant. An icon representing this Pursuit dialog box appears in the script and users can change their responses by clicking on the icon and editing the displayed meta dialog box directly.

Loop Constructs Once users have demonstrated the two branches of the loop and begin to demonstrate the program on a third example, Pursuit recognizes a loop. It asks the user to verify the two loop iterations by highlighting the panels in Figure 7. It then finishes executing the loop and updates the visual script. The updated script (shown in Figure 9) is an example of an explicit loop containing an explicit conditional. The declaration in the upper left corner states 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 the loop operations are contained within the large outer loop rectangle. The black square on the right edge of the prologue of the first `copy` operation indicates that the program branches at this point. The first branch (containing the string "no errors") is taken when `copy` executes successfully. The lower branch is taken when `copy` fails because a file with the output filename already exists. As with the conditional in Figure 7, this branch contains a predicate with a dialog box icon. However, the dialog box represented by the icon is not the same one that users had seen when demonstrating the operation (*i.e.* the one in Figure 8). Rather it is an abstraction of that dialog box and represents the inference Pursuit has made for the file set name. Thus, the string displayed is "A file with the name copy-of-<n1>.tex already exists in the folder."

Saving and Parameterizing Programs After demonstrating a program, it can be saved. Users indicate parameters by clicking on those objects in the script that represent the actual parameters. For example, clicking on the `papers` folder in Figure 9 indicates that the folder over which the program executes is a parameter. Once saved, a program can be executed by indicating its arguments and selecting the program name from the menu of user defined programs. Programs can also be edited and re-saved, or deleted.

THE PURSUIT VISUAL EDITOR

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

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

Users can also select a point in the script and add operations either by copying them from another point in the script or by demonstrating them. Operations can not be drawn from scratch. User defined branches (similar to the Lisp `cond`

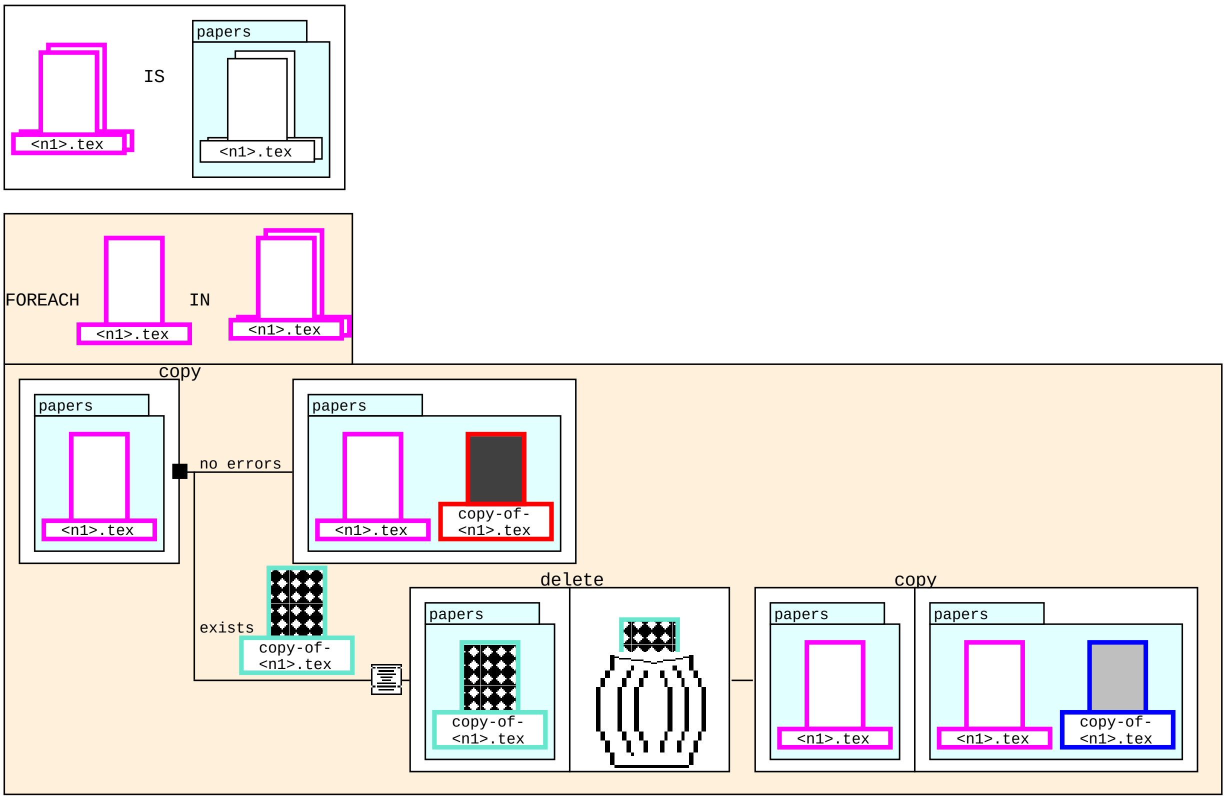

Figure 9: The Pursuit script that copies each `*.tex` file in the `papers` folder. If the `copy` operation fails because of the existence of a file with the output file name, the program deletes that old output file, and re-executes the `copy` operation. Users can see the other possible outcomes of the `copy` operation by clicking on the conditional marker.

construct) can also be added to the script. Currently, users can add predicates to test for the existence of an object, to compare the properties of two objects, such as their names or date of modification, or to compare the property of an object to a system constant, such as USER or TODAY.

INTEGRATING THE PBD SYSTEM AND VISUAL LANGUAGE

The main purpose of Pursuit's visual language is to represent demonstrated programs. In addition, Pursuit integrates the visual language into other parts of the PBD system. This helps improve feedback and forms a closer union between the PBD system and the program representation.

Inferring Iterations

Pursuit uses the visual script to indicate that it has inferred a loop. In the third example above, once Pursuit noticed that the user was demonstrating the loop it highlighted the operation panels in Figure 7. Users determine if Pursuit correctly inferred the loop by looking at the highlighted operations.

Inferring Sets and Subsets

When Pursuit identifies a set to loop over, it displays a dialog box containing the set's graphical representation, if such a representation exists. For example, suppose the user wishes to extend the program of Figure 6 so that the "copy-of-" prefix is removed from the compressed copies' names. To do so, the user removes the prefix from one of the copies and then another. Once Pursuit detects a loop and identifies the set to loop over, it queries the user to verify its inference (Figure 10). Users can confirm whether or not Pursuit has chosen the correct set by identifying the set's icon. A similar mechanism is used when Pursuit identifies one set as a subset of another.

Editing Programs

Pursuit also uses a program's visual representation to assist in the editing process. For example, suppose the user has forgotten to demonstrate a program path and the program is executed in a state that causes it to go down that path. Rather than aborting the program execution, Pursuit pops up the visual representation of the program and highlights the path that the program has taken, indicating where the program is incomplete. The user can then continue demonstrating the program, and Pursuit automatically updates the script.

Data icons are also used by the editor. For example, to add a user defined branch, users construct a predicate by choosing from a menu of data objects. The objects in the menu are represented using their visual icons (see Figure 11). Other ways that data icons are used by the editor include indicating

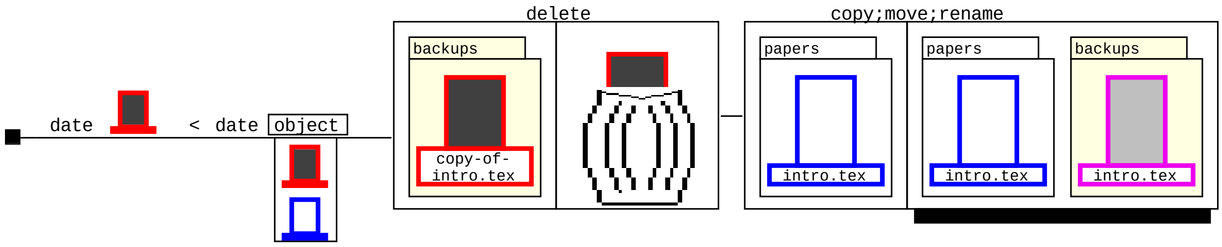

Figure 11: The user is adding a predicate to a branch before the demonstrated operations. The operations delete the file `copy-of-intro.tex` from the `backups` folder and make a new copy of `intro.tex` in the `papers` folder to place in the `backups` folder. The predicate states that the operations will be executed only if the old copy in the `backups` folder is older than the `intro.tex` file in the `papers` folder. The icon for the old copy is a miniature representation of the icon used in the first 2 panels of the script. The user is selecting the icon for `intro.tex` from the pop-up menu of data objects. The menu contains miniature representations of the data objects used in the script.

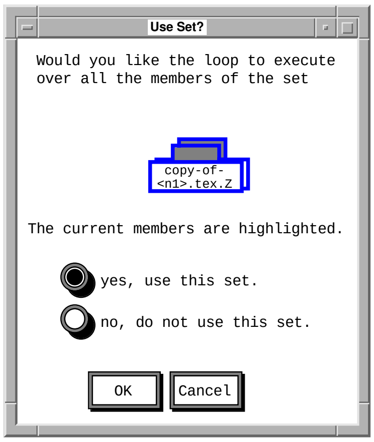

Figure 10: A Pursuit dialog box asking the user to confirm the set to loop over. The icon for the set of files is the same icon found in the third panel of Figure 6. The dialog box appears when the user executes the `rename` operation on two members of the set representing the copied files.

program parameters when saving a script, binding actual parameters to formal parameters when executing a script, and indicating the loop iteration set when wrapping a sequence of operations in an explicit loop.

IMPLEMENTATION DETAILS: SPECIFYING OPERATIONS DECLARATIVELY

In order to make it easy for programmers to add new operations, Pursuit contains a declarative language used to specify operations. An operation's specification defines its visual representation, its error conditions and dialog boxes, and how it affects the graphical appearance of data objects. During a demonstration, Pursuit uses this specification to generate a representation of the operation in the visual program as well as to infer over dialog boxes.

Figure 12 shows the specification for the `compress` operation. When an operation is executed, Pursuit looks up its specification in the table of operation specifications. Then it binds the actual inputs (outputs) of the operation to the inputs (outputs) in the specification. For a non-zero exit code, it may also bind the error causing file/folder to its representation in the specification. Then depending on the exit code returned by the operation, it determines what to represent visually in the program. In the remainder of this section we outline how Pursuit constructs its visual representations.

Creating the Prologue Before creating a prologue, Pursuit determines if 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 a new prologue is created. The panel for the prologue is added to the visual script 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 7).

Creating the Epilogue and Branch Conditions If an operation returns a zero exit code, then there was no error. Pursuit creates the epilogue by copying the prologue and executing the actions for the zero exit condition. If a non-zero exit code is returned, Pursuit constructs a branch condition using the information in the predicate slot for the returned exit code. When constructing a loop, Pursuit may have an operation that has two exit codes, such as the `copy` operation in Figure 9. In this case, predicates are constructed for all demonstrated exit conditions (including the no error condition) using the specification in the predicate slots of the exit codes.

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 uses the operation's specification along with the specifications of all the operations whose epilogues are contained in the prologue of the current operation to determine if the new operation's epilogue can be combined with the previous ones. 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

```
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 12: A pseudocode representation of the declarative specification of the `compress` operation.

all the involved operations is empty. This insures 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.

Dialog Boxes An operation's specification also contains information on the contents of the dialog boxes it displays. A dialog box specification provides a pointer to the actual displayed dialog box along with a specification for how the initial values of fields in the dialog box are derived. In addition, each dialog box contains a list of its fields and whether or not users can set them. For example, the `merge` operation, which appends the contents of one file to another and writes the output to a file, pops up a dialog box in order to obtain the name of the output file from the user. The output file name field of that dialog box is marked as user setable. On the other hand, most dialog boxes that are used to relay error messages have fields whose values are determined only by the operation.

In order to display a dialog box, all utilities must send the dialog box along with a list of initial values to the Pursuit dialog box manager. The dialog box manager then initializes the dialog box fields and displays the dialog box. When the user closes the dialog box, control is returned to the dialog box manager. If the user is recording a script, then using the list of (field, how set) pairs, the dialog box manager determines which fields (if any) are set by the user. If all fields are constant (such as with error messages), then Pursuit pops up the meta dialog box like the own shown in Figure 8. If some fields must be set by the user then Pursuit pops up a different meta dialog box. The dialog box manager then cycles through all the user set-able fields in the displayed dialog box asking the user to indicate how the field should be set when the program is executed: by being entered by the user at execution time, by being computed by the program at execution time, or by having the same value at execution time that the user entered this time. For fields that must be computed by the program, Pursuit attempts to determine how that field's value is derived. For example, the output file name could be based on the input file name. If the inference mechanism can not determine how that field is derived, it converts it to an ask the user field. If after Pursuit determines how the value of each field is determined the dialog box manager notices that all fields are either constant or can determined by the program, then it asks the user if the dialog box should be displayed during program execution.

RELATED WORK

There have been several previous attempts to use visual language techniques to add end-user programming to visual shells. SmallStar[**?**] contains a macro recorder that makes an "English-like" transcript of user actions. The representation contains icons for files and folders. Although effective in automating simple, repetitive tasks, SmallStar records exactly what users do – only the object that is pointed to can be a parameter and the transcript consists of a straight-line sequence of commands. The transcript can only be generalized via a menu of commands.

Some visual shells have invented a special graphical programming language (*e.g.*, [**?**, **?**]) to enable users to write programs. Most of these languages are based on the data flow model, in which icons represent utilities and lines connecting them represent data paths. Unfortunately, most contain no way to depict abstractions or control structures. The types of programs users can write are quite limited. In addition, these languages require users to learn a special programming language whose syntax differs significantly from what they see in the interface. Finally, constructing programs by wiring together objects is quite different from the way users ordinarily interact with the system.

Although Pursuit's representation language is similar to the editable histories of Chimera[**?**], there are many important differences. First, Pursuit's visual language contains *ab-*

stractions 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, since Pursuit objects can be identified by their icons. On the other hand, Chimera panels contain objects not involved in operations (such as the cursor) in order to provide contextual information to help identify objects and operations. Furthermore, Pursuit's inferences are displayed in the visual script and are always visible, whereas Chimera's inferences are contained in textual supplements and are not visible in scripts. In addition, Pursuit scripts are two dimensional. Information is conveyed from left to right and top to bottom (see Figure 9). This makes the language more powerful than Chimera, which use only a linear (left to right) display. Finally, Pursuit scripts visibly represent loops and conditionals, which are inferred automatically. Chimera macros must be edited to contain loops, which have no explicit representation, and Chimera contains no mechanism for inferring, adding or representing conditionals.

Mondrian[**?**], a demonstrational graphical editor, also uses a similar representation paradigm. Like Chimera, Mondrian programs are one dimensional focused snapshots of the screen augmented with other objects needed for context. Like Pursuit, Mondrian interactively obtains advice from the user to help with generalizing. Mondrian is unique in its use of speech as a form of feedback, the ability to create recursive procedures by including an iconic representation of the developing program in that program during the demonstration, and the ability to add textual annotations.

STATUS AND FUTURE WORK

Pursuit is implemented in Lisp using the Garnet Toolkit[**?**]. Currently, we are performing user studies to evaluate the visual language itself as well as the entire Pursuit system to determine how well it helps users automate tasks. In these studies, users will construct programs for some tasks using Pursuit. These results will be compared with users doing the same task in the Pursuit visual shell but whose program representation language is an "English-like" language similar to the one found in SmallStar[**?**]. This will help us evaluate whether or not the visual representation really does help simplify the programming process. We also plan to compare the recognizability of programs in both the Pursuit visual language and the "English-like" language. This study will help us determine if there are aspects of the visual language which are particularly useful (or not) for helping users to understand and recognize programs.

In the future, we hope to extend the PBD techniques so that users can demonstrate actions on all objects in the interface, not just files and folders. We will then address the problems of how to represent these objects (such as dates, owners, etc.) in the visual language and how to extend the declarative specification language to handle them.

CONCLUSION

We have presented a visual shell called Pursuit. Pursuit contains a new visual programming language and environment that enables users to construct programs containing variables, loops and conditionals with minimal programming knowledge. Visual shells are easy to use because of the constantly visible, concrete, familiar representations of data objects and the illusion of concrete manipulation of these data objects[**?**]. Unfortunately, this "conceptual simplicity" is often lost when programming is introduced: users interact with the system visually, but usually program it off-line in a textual programming language. Users must develop two very different bodies of knowledge: one for interacting with the system and one for programming it. Pursuit attempts to bridge this gap. By 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. Moreover, because a program's representation appears incrementally during the demonstration, users can interactively learn how operations and data (*i.e.* the language's syntax) are represented. Finally, by integrating these visual representations within three different parts of the PBD system – the program representation, the communication of inferences, and the editor, Pursuit forms a close union between the interface, the PBD system and the program representation, and helps bridge the gap between the user's mental model of the programming process and the actual programming task.

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