Color profile: Generic CMYK printer profile
Composite Default screen

Chapter Three

Demonstrational Interfaces:

Sometimes You Need a Little Intelligence, Sometimes You Need a Lot

BRAD A. MYERS

Human Computer Interaction Institute
Carnegie Mellon University

RICHARD MCDANIEL

Siemens Technology to Business Center

S
R
L

V:\002564\002564.VP
Wednesday, December 13, 2000 10:12:04 AM
TNT Job Number: [**002564**] • *Author:* [**Lieberman**] • *Page:* **45**

Color profile: Generic CMYK printer profile
Composite Default screen

Abstract

Over the last fifteen years, we have built over a dozen different applications in many domains where the user can define behaviors by demonstration. In some of these, the system uses sophisticated artificial intelligence (AI) algorithms so that complex behavior can be inferred from a few examples. In other systems, the user must provide the full specification, and the examples are primarily used to help the user understand the situation. This chapter discusses our findings about which situations require increased intelligence in the system, what AI algorithms have proven useful for demonstrational interfaces, and how we cope with the well-known usability issues of intelligent interfaces such as knowing what the system can do and what it is doing.

3.1 Introduction

Demonstrational interfaces allow the user to perform actions on concrete example objects (often by direct manipulation), but the examples represent a more general class of objects. This can allow the user to create parameterized procedures and objects without requiring the user to learn a programming language. The term *demonstrational* is used because the user is *demonstrating* the desired result using example values. Significant differences exist among demonstrational systems along many dimensions. Some demonstrational interfaces use *inferencing,* in which the system guesses the generalization from the examples using heuristics. Other systems do not try to infer the generalizations and therefore require the user to tell the system explicitly what properties of the examples should be generalized. One way to distinguish whether a system uses inferencing is that if it does, it can perform an incorrect action even when the user makes no mistakes. A noninferencing system will always do the correct action if the user is correct (assuming there are no bugs in the software, of course), but the user is required to make all the decisions.

The use of inferencing comes under the general category of *intelligent interfaces.* This category entails any user interface that has some “intelligent” or artificial intelligence (AI) component, including demonstrational interfaces with inferencing but also other interfaces such as those using natural language. Systems with inferencing are often said to be “guessing”

S
R
L

V:\002564\002564.VP
Wednesday, December 13, 2000 10:12:05 AM
TNT Job Number: **[002564]** • *Author:* **[Lieberman]** • *Page:* **46**

Color profile: Generic CMYK printer profile
Composite Default screen

what the user wants. Another term is *heuristics,* which refers to rules that the system uses to try to determine what the user means. The rules in most demonstrational systems are defined when the software is created and are often quite specialized. (For another discussion of the strategies used by demonstrational systems and other intelligent interfaces, see chap. 3 of Maulsby 1994.)

Over the past fifteen years, the Demonstrational Interfaces group at Carnegie Mellon University has created over a dozen demonstrational systems that use varying amounts of inferencing. Some systems have none and just create an exact transcript of what the user has performed so it can be replayed identically. Most provide a small number of heuristics that try to help users perform their tasks. Our most recent system, Gamut, employs sophisticated AI algorithms such as plan recognition and decision tree learning to infer sophisticated behaviors from a small number of examples. (Gamut is described in more detail in chap. 8 of this book.)

This chapter discusses our experience with using various levels of "intelligence" in the interfaces. Some have criticized the whole area of intelligent interfaces because users may lose control. We strive to overcome this problem by keeping users "in the loop" with adequate feedback so they know what is happening and opportunities to review what the system is doing and to make corrections. However, this is still an unsolved area of research and reviewing the different trade-offs in different systems can be instructive.

First, we provide an overview of the demonstrational systems we have created, followed by a discussion of systems that have no inferencing, simple rule-based inferencing, and sophisticated AI algorithms. Finally, we discuss the issue of feedback, which is important to keep the user informed about what the system is doing.

3.2 Our Demonstrational Systems

The Demonstrational Interfaces group at Carnegie Mellon University has created a wide variety of demonstrational systems over the last fifteen years. This section summarizes our systems.

- *Peridot* (Myers and Buxton 1986; 1990): One of the first demonstrational systems to use inferencing, it allows users to create widgets such as menus and scroll bars entirely by demonstration. It uses extensive

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

heuristic, rule-based inferencing from single examples and, for each rule that matched, asks the user to confirm the inference.

- *Lapidary* (Myers, Vander Zanden, and Dannenberg 1989; Vander Zanden and Myers 1995): It allows application-specific graphical objects, such as the prototypes for node and arc diagrams, to be created interactively. Lapidary uses simple heuristics from a single example.
- *Jade* (Vander Zanden 1990): It automatically creates dialogue boxes from a specification. Some of the parameters of the layout could be specified by demonstration.
- *Gilt* (Myers 1991b): This is an interface builder where the parameters to the callbacks attached to the widgets can be demonstrated by example. A callback that set some widget's value based on others can be eliminated by demonstrating its behavior. Gilt uses simple heuristics from single examples.
- *Tourmaline* (Myers 1991c; Werth 1993): It uses rule-based inferencing of text structures and styles from one example. When an example contains more than one kind of formatting, the system tries to infer the role of each part and its formatting. For example, if a conference heading is selected, the system tries to infer the separate formatting of the author, title, affiliation, and so forth.
- *C32* (Myers 1991a): It provides a spreadsheet-like display for defining and debugging constraints. The constraints can be generalized by giving an example.
- *Pursuit* (Modugno, Corbett, and Myers 1997; Modugno and Myers 1997b): It creates scripts for automating "visual shell" (desktop) tasks on files. Focuses on providing a visible record of the inferred program in a graphical "comic book"–style visual language. Generalizes the objects on which to operate based on the properties of a single example.
- *Gold* (Myers, Goldstein, and Goldberg 1994): It creates business charts and graphs given an example of one or two elements of the chart and the desired data. It generalizes from the example marks to match the format of the data, and the results of the inferences are immediately shown in the chart.
- *Marquise* (Myers, McDaniel, and Kosbie 1993): It creates drawing editors by showing examples of what the end user would do, and then what the system would do in response. Built-in heuristics and widgets help with

S
R
L

V:\002564\002564.VP
Wednesday, December 13, 2000 10:12:05 AM

Color profile: Generic CMYK printer profile
Composite Default screen

palettes that control modes and parameters. Feedback to the user is provided using an English-like presentation of the inferred code, where choices in the code are pop-up menus.

- *Katie* (Kosbie and Myers 1993, 1994): It records high-level events (e.g., "delete object-A") as well as low-level events (e.g., mouse-move) and allows macros to use any level. Katie proposes that macros could also be invoked when low- or high-level events occur.
- *Turquoise* (Miller and Myers 1997): It creates composite Web pages from examples of what they should contain. The demonstration is performed by copying and pasting content from the source Web pages.
- *Topaz* (Myers 1998): It creates scripts in a graphical editor, in a way analogous to macros in textual editors such as Emacs. Generalizations of parameters to operations are available in dialogue boxes. The generated script is shown in an editing window.
- *Gamut* (McDaniel and Myers 1998, 1999; see also chap. 8): Sophisticated inferencing mechanisms allowed the creation of complete applications from multiple examples, both positive and negative. Novel interaction techniques make specifying the behaviors easier.

3.3 Level of Intelligence

Many of our systems, and most of the PBD systems by others, have used fairly simple inferencing techniques. The hope has been that since these systems are interactive, the user will be "in the loop," helping the system when it cannot infer the correct behavior. Furthermore, many of the PBD system developers have not been AI researchers, so they have been reticent to apply unproven AI algorithms. Finally, a number human-computer interface (HCI) researchers (e.g., Shneiderman 1995) have argued against "intelligent" user interfaces, claiming that this will mislead the user and remove the necessary control. On the other hand, experience with the many PBD systems shows that users continually expect the system to make more and more sophisticated generalizations from the examples, and the PBD researchers want their systems to be able to create more complex behaviors. Therefore, there is pressure to make the systems ever more "intelligent." In our own systems, we have explored systems at both ends of spectrum and many points in between.

S
R
L

V:\002564\002564.VP
Wednesday, December 13, 2000 10:12:06 AM
TNT Job Number: [**002564**] • *Author:* [**Lieberman**] • *Page:* **49**

Color profile: Generic CMYK printer profile
Composite Default screen

3.3.1 No Inferencing

Our Topaz system for creating macros or scripts of actions in a video editor requires the user to explicitly generalize the parameters of the operations using dialogue boxes (Myers 1998). For example, Figure 3.1(a) shows a script window (in the background) with a pop-up dialogue box that can be used to generalize the references to objects in the recorded script. The script was created using specific example objects, but the user could generalize these objects in various ways. Among the choices in the dialogue box are that the run-time object should be whatever object is selected at run time, the objects that result of a previous command, and so forth. The only heuristic built into Topaz is that when a script creates objects and then operates on the created objects, the default generalization for the latter operations is that they operate on the dynamically created objects each time the script executes. This heuristic is almost never wrong, so there is little chance of the system guessing incorrectly in this case.

Since there is no inferencing, the system never makes mistakes, and the user has complete control. However, the user must figure out how to get the desired actions to occur, which often requires clever use of special Topaz features, such as searching for objects by their properties and the different ways to generalize parameters.

Many other PBD systems take the no-inferencing approach, including the seminal Pygmalion (Smith 1977) and SmallStar (Halbert 1993) systems.

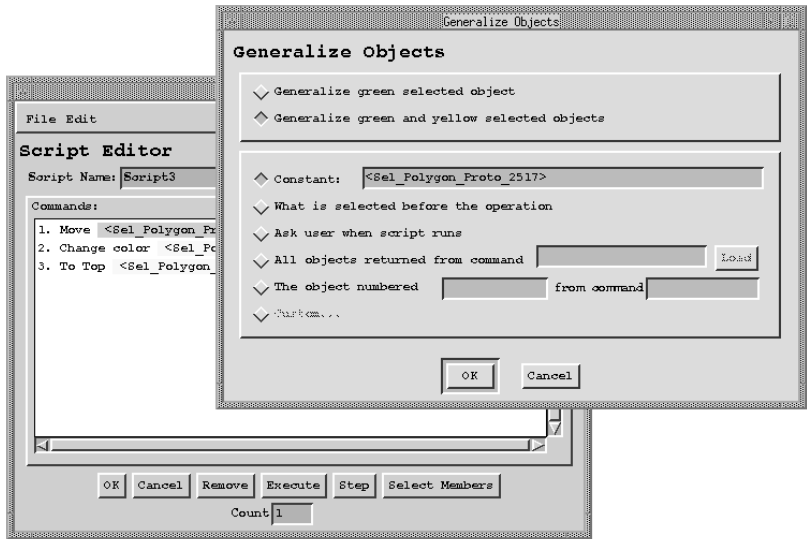
3.3.2 Simple Rule-Based Inferencing

Most of our systems have used simple rule-based heuristic inferencing for their generalizations. For example, the early Peridot system (Myers 1990) used about fifty hand-coded rules to infer the graphical layout of the objects from the examples. Each rule had a test part, a feedback part, and an action part. The test part checked the graphical objects to see whether they matched the rule. For example, the test part for a rule that aligned the centers of two rectangles would check whether the centers of the example rectangles were approximately equal. Because the rules allowed some sloppiness in the drawing, and because multiple rules might apply, the feedback part of the rule was used to ask the user if the rule should be applied. For example, Peridot would ask something like, “Do you want the selected rectangle to be centered inside the other selected rectangle?” If the user answered yes, then the action part of the rule generated the code to maintain the constraint.

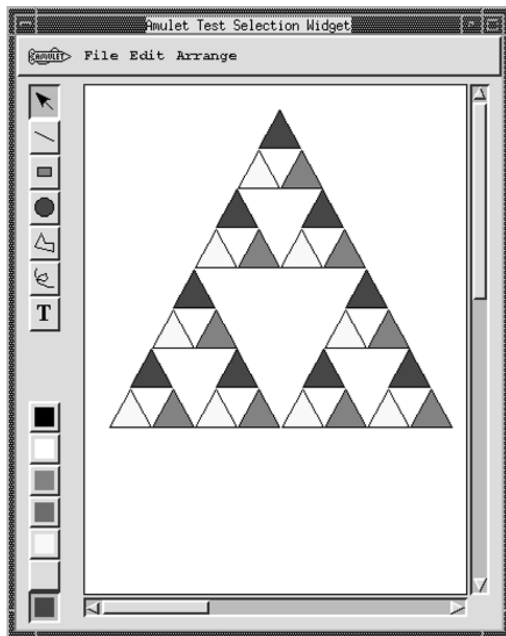
S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

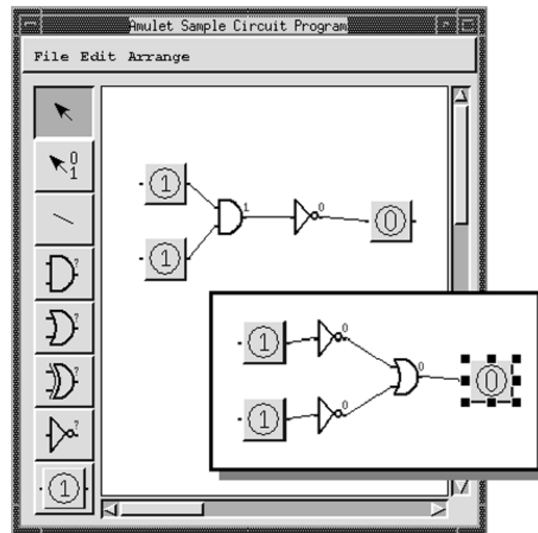
FIGURE 3.1



(a)



(b)



(c)

(a) The dialogue box for generalizing objects in Topaz, with the script window in the background. The user must select which generalizations to apply. Using such dialogue boxes, sophisticated scripts can be created, such as (b) a script to create a "Sierpinski Gasket" and (c) a script that applies DeMorgan's law by changing an And gate and a Not gate into two Not gates and an Or gate and reconnects the wires.

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

Subsequent systems have used similar mechanisms, although often without the explicit list of rules used by Peridot. For example, Tourmaline contained rules that tried to determine the role of different parts of a header, such as the section number, title, author, and affiliation, and the formatting associated with each part (Werth and Myers 1993). The results were displayed in a dialogue box for the user to inspect and correct. Many other PBD systems use the rule-based approach from single examples, including Mondrian (Lieberman 1992) and Pavlov (Wolber 1997).

An important advantage of using such rule-based heuristics that generalize from a single example is that it is much easier to implement. No sophisticated AI algorithms are required, and the developer can hand-code the rules and adjust their parameters. It is also easier for the user to understand what the system is doing, and eventually users may internalize the full set of rules. The disadvantages are that only a limited form of behavior can be generalized, since the system can only base its guess on a single example. More complex behaviors either are not available or must be created by editing the code generated by the PBD system. Since predetermined behaviors are all that are available, some argue that these behaviors might instead be made available in a direct-manipulation way, such as a menu, to avoid the problems of inferencing.

Another interesting issue with rule-based systems is whether the end user can change the rules. Most existing PBD systems have used a fixed set of rules created by the designer. In many cases, the users might have a better idea of the rules that would be appropriate for the applications they would like to build, so it would be useful for the end user to be able to specify new rules. However, if users could write the code for the rules, then they would probably not need PBD support. No one has produced a system that allows new rules to be entered into the system using PBD, which is an interesting metaproblem.

3.3.3 Sophisticated AI Algorithms

Recognizing the limitations of single-example, rule-based approaches, and wanting to create more sophisticated behaviors, the Gamut system infers behaviors from multiple examples (McDaniel and Myers 1999).

Gamut begins building a new behavior in much the same way as the single-example systems. However, each time the developer refines the behavior by adding a new example, Gamut uses metrics to compare the new example with the current behavior and choose how the behavior should be changed. For example, if Gamut sees that parts of the code that were

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

previously executed are no longer being used in the new example, it will enclose that code in an if-then statement. If Gamut sees that an object is being set with a different value, it will use the new value along with heuristics to select changes to the code that produces that value. Gamut also uses a *decision tree* algorithm to generate code that could not be generated from rules alone. For example, Gamut uses decision trees to represent the predicates of if-then statements.

Gamut uses interaction techniques to support its algorithms. For example, the developer can create "guide" objects to represent the state of the application that is not part of its visible interface. These are visible at design time but disappear at run time. The developer can also highlight objects into order to give the system hints. The hints guide Gamut's heuristic algorithms when they select relationships to put into a behavior. Gamut can also request that the developer highlight objects when its algorithms reach an impasse and can no longer infer code without the developer's help.

Other PBD systems have worked from multiple examples. Some script-based systems, such as MetaMouse (Maulsby and Witten 1989) and Eager (Cypher 1991), compare multiple executions of commands to look for possible macros that the system might create automatically for the user. From the matches, these systems generalize the parameters to the operations, using simple heuristics such as the next item in a sequence, or offset by a constant factor. InferenceBear can generate more sophisticated behaviors by supporting linear equations to compute the parameters and letting the user provide negative examples to show when behaviors should not occur (Frank 1994).

When inferring from multiple examples, the system must be able to recognize both positive and negative examples. In the simplest case, a positive example demonstrates a condition when a behavior should occur; a negative example shows when the behavior should not occur. Without negative examples, a system cannot infer many behaviors, such as ones that use a Boolean-OR. In Gamut, the developer can use either the *Do Something* or *Stop That* button to create a new example. These choices roughly correspond to creating a positive or negative example. In some other systems, negative examples are recognized implicitly. For instance, when MetaMouse detects that a behavior is repeating, the actions that are not repeated in the current iteration act as a negative example. This will signal to the system that a conditional branch is required. InferenceBear required users to note negative examples explicitly, which proved difficult for users to understand.

The primary advantage of using more sophisticated techniques is that the system can infer more complex behaviors. The disadvantages include

S
R
L

V:\002564\002564.VP
Wednesday, December 13, 2000 10:12:19 AM

Color profile: Generic CMYK printer profile
Composite Default screen

that the systems are much more difficult to implement, and they still may not infer correctly. There is a "slippery slope" for all intelligent interfaces, including PBD systems: once the system shows a little intelligence, people may expect it to be able to infer as much as a human would, which is well beyond the state of the art. It is still an open problem to balance the sophistication of the system with the expectations of users.

3.4 Feedback

In user studies of the EAGER system, users were reticent to let the system automate their tasks because it provided no feedback about what the stopping criteria were (Cypher 1991). Another important issue is how can users edit the program if they change their mind about what it should do?

Clearly some form of representation of the inferred program is desirable. But since users of PBD are generally considered to be nonprogrammers, it seems inappropriate to require them to read and understand code. If they could understand the code, they might be able to write it in the first place, so no PBD would be needed. In some cases, however, the goal of the PBD is to help the user get started with the language. Since it is easier for people to recognize than to recall, seeing the code generated from the examples may help people learn the language and give them a start on their programs. This is an important motivation for adding the record mode for creating macros by example in spreadsheets and other programs. However, much evidence indicates that people have great difficulty understanding the generated code and modifying it when it isn't exactly correct. Therefore, an important challenge for PBD systems is how to let the user understand and edit the program without obviating the benefits that PBD provides.

Peridot (Myers 1990) used question-and-answer dialogues to confirm each inference, which proved to be problematic because people tended to simply answer yes to every question, apparently assuming that the computer knew what it was doing. Peridot also had no visible representation of the code after it was created, so there was no way to check or edit the resulting program.

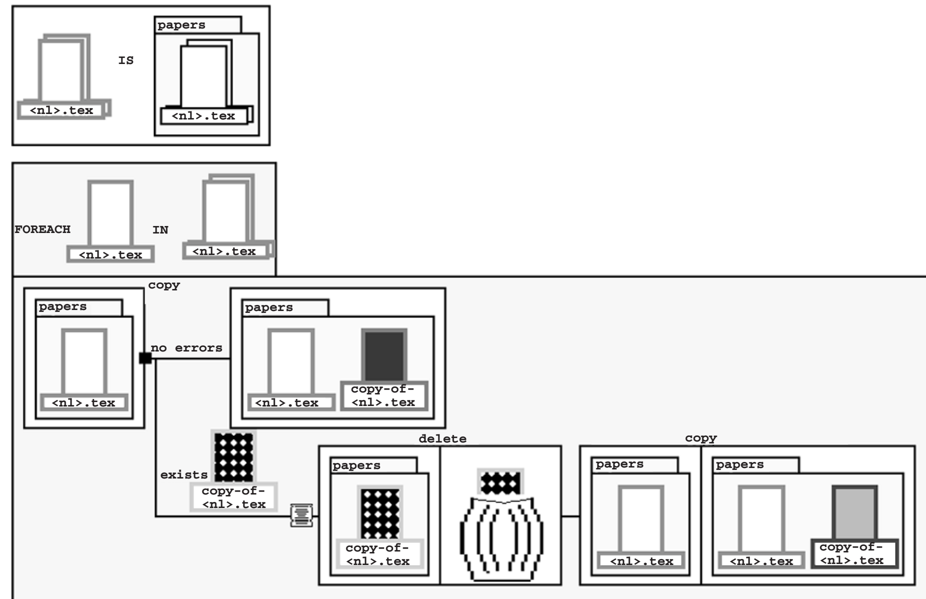
Our Pursuit system (Modugno and Myers 1997) focused on studying the issue of feedback. It used a novel graphical presentation that showed before and after states, based on a "comic strip" metaphor. The domain was the manipulation of files in a "visual shell" or desktop, and the visual presentation showed examples of files icons before and after each operation. Figure 3.2 shows an example of a relatively complex program in Pursuit. The same

S
R
L

V:\002564\002564.VP
Wednesday, December 13, 2000 10:12:19 AM
TNT Job Number: [002564] • Author: [Lieberman] • Page: 54

Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE **3.2**



A program represented in Pursuit that looks for all the files in the papers directory that match ".tex" and, for each of these, tries to copy the file. If an error occurs during copying because the resulting file already exists (lowest branch), then the program deletes the old file and does the copy again.*

visual language was used to allow users to confirm and repair inferences by the system (e.g., how to identify the set of files to operate on) and to enable editing of programs later, (e.g., if the user wanted a slightly different program for a new task. User studies with Pursuit showed that people could more easily create correct programs using the graphical language than with an equivalent textual representation and that they could make relatively complex programs containing conditionals and iterations.

Marquise (Myers et al. 1993) generated a textual representation of the program, as shown in Figure 3.3. The evolving program was represented as sentences, with each choice represented by a button embedded in the sentence. Clicking on a button would provide alternate options, and changing the option might also change the subsequent parts of the sentence.

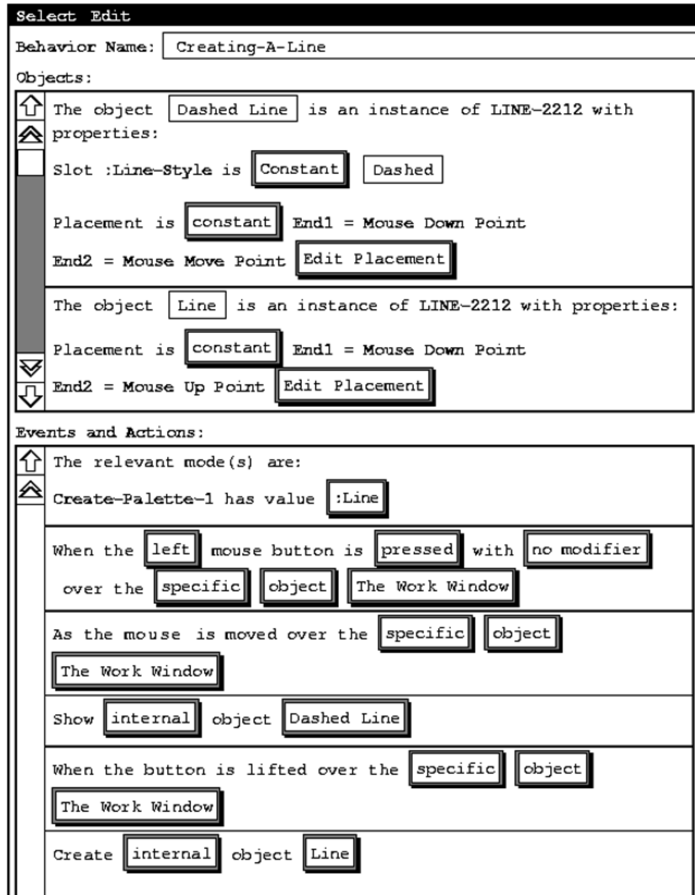
Experience with this form of feedback showed that it had a number of problems. The implementers found it very difficult to design the sentences

S
R
L

V:\002564\002564.VP
Wednesday, December 13, 2000 10:12:23 AM

Color profile: Generic CMYK printer profile
Composite Default screen

FIGURE 3.3



This behavior, shown in the feedback window of Marquise, controls the drawing of a line. When the create palette is in the line mode, a dotted line follows the mouse, and when the mouse button is lifted, a solid line is drawn. The embedded buttons represent options that can be used to edit the behavior.

so they were readable and still contain the correct options. When users wanted to change the behavior in significant ways, it was often difficult to figure out which button to use.

The Topaz system (Myers 1998) used a relatively straightforward representation of the program (see Figure 3.1[a]). The operation name shown in

S
R
L

V:\002564\002564.VP
Wednesday, December 13, 2000 10:12:24 AM
TNT Job Number: [002564] • Author: [Lieberman] • Page: 56

Color profile: Generic CMYK printer profile
Composite Default screen

the script was often the same as the menu item that the user selected (e.g., *To Top*), and the parameters started off constant and could be changed using dialogue boxes. All the usual editing operations are supported for the script, such as Cut, Copy, and Paste. Clicking on a parameter would bring up the dialogue box used to generalize that type of parameter. The result is that this representation seems easy to understand and edit.

Like EAGER, Gamut has no visible presentation of the inferred program, and user studies with Gamut's report that one would be desirable (McDaniel and Myers 1999). Gamut gets around the editing problem by allowing users to supply new examples at any time that will modify the existing behaviors, so it is never necessary to edit code directly.

Many other systems have researched different presentations of the generated program. Most use custom languages designed specifically to match the PBD system. SmallStar (Halbert 1993) provided a textual representation of the program created from the examples, and the user had to explicitly edit the program to generalize parameters and add control structures. InferenceBear (Frank and Foley 1994) used a novel form of event language as its representation called *Elements, Events & Transitions* (EET) in which the actions and their parameters were represented as event handlers. This relatively complex language is probably not accessible to nonprogrammers. Mondrian (Lieberman 1992) provided a comic strip–style visual program, similar to Pursuit. Pavlov (Wolber 1997) tries to make the program accessible to nonprogrammers by using a time line–based view, in the style of MacroMedia's Director but based on an event-based model. This makes the language more appropriate for describing graphical applications since they are primarily event based (using the keyboard and mouse generates events that are handled by the application).

3.5 Conclusion

Designing the heuristics for a PBD system is a very difficult task. The more sophisticated the inferencing mechanism, the more complex the behaviors that can be handled, and, hopefully, the more likely it is that the system will infer correctly. On the other hand, sophisticated inferencing mechanisms often bring with them more elaborate user interfaces to control the inferencing, and these systems are much more difficult to implement. In any kind of PBD system, it is important that there be a well-designed feedback mechanism so that users can understand and control what the system is doing and change the program later. Much more research is needed on

S
R
L

Color profile: Generic CMYK printer profile
Composite Default screen

the appropriate level of intelligence and the forms of feedback in PBD systems.

Acknowledgments

The systems reported here were developed under many research grants, with support from the National Science Foundation under grants IRI-9020089 and IRI-9319969, DARPA under Contract No. N66001–94-C-6037, Arpa Order No. B326, and others. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

References

Cypher, Allen. “Eager: Programming Repetitive Tasks by Example.” Proceedings of CHI ’91, ACM, New Orleans, 1991, pp. 33–39.

Frank, Martin R., and James D. Foley. 1994. A pure reasoning engine for programming by demonstration. In *ACM SIGGRAPH symposium on user interface software and technology* (Marina del Rey, Calif., November).

Halbert, Daniel C. 1993. SmallStar: Programming by demonstration in the desktop metaphor. In *Watch what I do: Programming by demonstration,* ed. Allen Cypher. Cambridge, Mass.: MIT Press.

Kosbie, David S., and Brad A. Myers. 1993. A system-wide macro facility based on aggregate events: A proposal. In *Watch what I do: Programming by demonstration,* ed. Allen Cypher. Cambridge, Mass.: MIT Press.

———. 1994. Extending programming by demonstration with hierarchical event histories. In *Human-computer interaction: 4th International Conference EWHCI’94. Lecture Notes in Computer Science, Vol. 876.* Berlin: Springer.

Lieberman, Henry. 1992. Dominos and storyboards: Beyond icons on strings. In *1992 IEEE Workshop on visual languages* (Seattle, Wash., September).

Maulsby, David. 1994. *Instructible agents.* Ph.D. diss. University of Calgary, Calgary, Alberta, Canada.

Maulsby, David L., and Ian H. Witten. 1989. Inducing procedures in a direct-manipulation environment. In *Human factors in computing systems* (Austin, Tex., April). ACM CHI ’89 Proceedings

S
R
L

V:\002564\002564.VP
Wednesday, December 13, 2000 10:12:24 AM
TNT Job Number: **[002564]** • *Author:* **[Lieberman]** • *Page:* **58**

Color profile: Generic CMYK printer profile
Composite Default screen

McDaniel, Richard G., and Brad A. Myers. 1998. Building applications using only demonstration. In *1998 International Conference on Intelligent User Interfaces* (San Francisco, January). ACM

———. 1999. Getting more out of programming-by-demonstration. In *Human factors in computing systems* (Pittsburgh, Pa., May 15–20). ACM CHI '99 Proceedings

Miller, Robert C., and Brad A. Myers. 1997. Creating dynamic World Wide Web pages by demonstration. Carnegie Mellon University School of Computer Science, CMU-CS-97–131 and CMU-HCII-97–101.

Modugno, Francesmary, Albert T. Corbett, and Brad A. Myers. 1997. Graphical representation of programs in a demonstrational visual shell—An empirical evaluation. *ACM Transactions on Computer-Human Interaction.* 4, no. 3: 276–308.

Modugno, Francesmary, and Brad A. Myers. 1997. Visual programming in a visual shell—A unified approach. *Journal of Visual Languages and Computing* 8, nos. 5/6: 276–308.

Myers, Brad A. 1990. Creating user interfaces using programming-by-example, visual programming, and constraints. *ACM Transactions on Programming Languages and Systems.* 12, no. 2: 143–177.

———. 1991a. "Graphical techniques in a spreadsheet for specifying user interfaces." Proceedings of CHI '91, ACM, New Orleans, 1991, pp. 243–249.

———. 1991b. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *ACM SIGGRAPH symposium on user interface software and technology* (Hilton Head, S.C., November).

———. 1991c. Text formatting by demonstration. In *Human factors in computing systems.*

———. 1998. Scripting graphical applications by demonstration. In *Human factors in computing systems* (Los Angeles, April).

Myers, Brad A., and William Buxton. 1986. Creating highly interactive and graphical user interfaces by demonstration. In *Computer graphics* (Dallas, Tex., August).

Myers, Brad A., Jade Goldstein, and Matthew A. Goldberg. 1994. Creating charts by demonstration. In *Human factors in computing systems* (Boston, April).

Myers, Brad A., Richard G. McDaniel, and David S. Kosbie. 1993. Marquise: Creating complete user interfaces by demonstration. In *Human factors in computing systems* (Amsterdam, April).

Myers, Brad A., Brad Vander Zanden, and Roger B. Dannenberg. 1989. Creating graphical interactive application objects by demonstration. In *ACM SIGGRAPH Symposium on user interface software and technology* (Williamsburg, Va., November).

Shneiderman, Ben. 1995. Looking for the bright side of user interface agents. *ACM Interactions* 2, no. 1: 13–15.

S
R
L

V:\002564\002564.VP
Wednesday, December 13, 2000 10:12:25 AM

Color profile: Generic CMYK printer profile
Composite Default screen

Smith, David Canfield. *Pygmalion: A computer program to model and stimulate creative thought.* Basel: Birkhauser.

Vander Zanden, Brad, and Brad A. Myers. 1990. Automatic, look-and-feel independent dialog creation for graphical user interfaces. In *Human factors in computing systems* (Seattle, Wash., April).

———. 1995. Demonstrational and constraint-based techniques for pictorially specifying application objects and behaviors. *ACM Transactions on Computer-Human Interaction* 2, no. 4: 308–356.

Werth, Andrew J., and Brad A. Myers. 1993. Tourmaline: Macrostyles by example. In *Human factors in computing systems* (Amsterdam, April).

Wolber, David. 1997. An interface builder for designing animated interfaces. *ACM Transactions on Computer-Human Interaction* 4, no. 4: 347–386.

S
R
L

V:\002564\002564.VP
Wednesday, December 13, 2000 10:12:25 AM