A system needs the right level of intelligence to infer the correct generalizations from examples while providing enough feedback to keep the user in control.

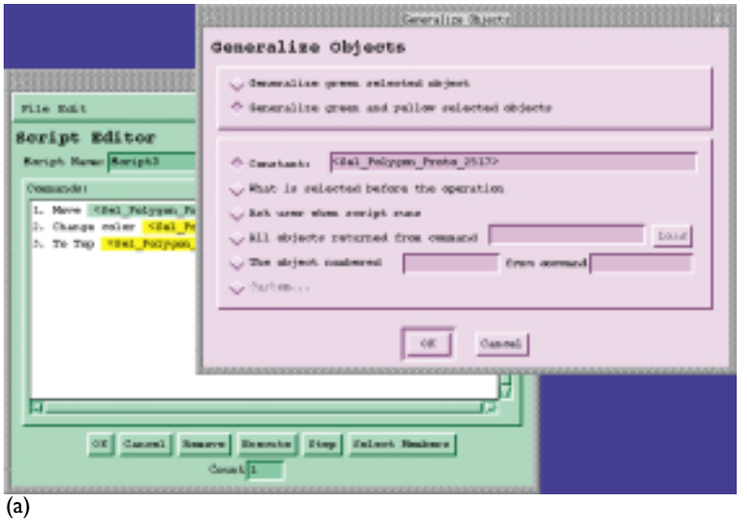
Intelligence in Demonstrational Interfaces

For the past 15 years, we have built about a dozen different applications in many domains in which the user defines behaviors by demonstration. In some of them, the system uses sophisticated AI algorithms, so complex behavior can be inferred from a few examples. In others, the user has to provide the full specification, and the examples are used primarily to help the user understand the programming situation. Here, we discuss what we've learned about which situations require increased intelligence in the system, which AI algorithms have proven useful for demonstrational interfaces, and how we cope with such well-known intelligent-interface-usability issues as knowing what the system can do and what it is doing at any given moment.

Demonstrational interfaces allow the user to perform actions on concrete example objects (often using direct manipulation), though the examples represent a more general class of objects. These actions allow the user to create "parameterized" procedures and objects without being required to learn a programming language. We use the term "demonstrational," because the user demonstrates the desired result through example values. Demonstrational systems vary greatly along many dimensions; some of these systems use inferencing, whereby the system employs heuristics to guess the generalization from the examples; others do not try to infer the generalizations and instead require the user to describe explicitly which of the examples' properties should be generalized. One way to determine whether a system uses inferencing is that it can perform an incorrect action, even when the user makes no mistakes. A system that doesn't employ inferencing always performs the correct action if the user is

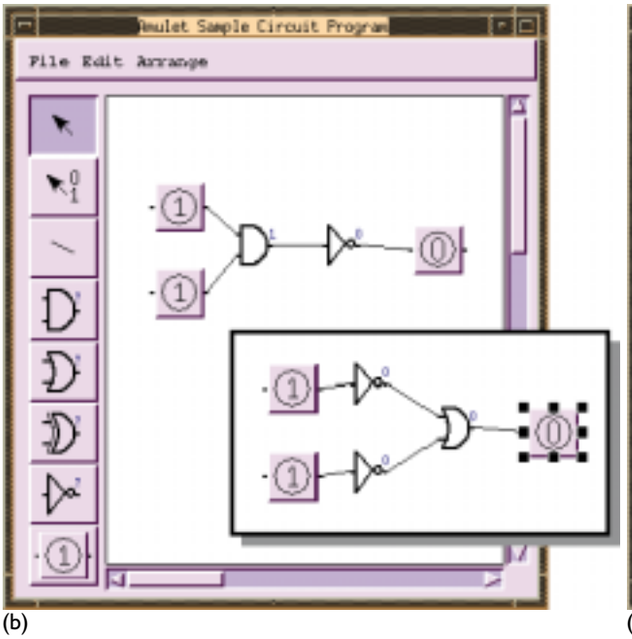
Brad A. Myers, Richard McDaniel, and David Wolber

Figure 1. Generalizing references to objects in a recorded script.

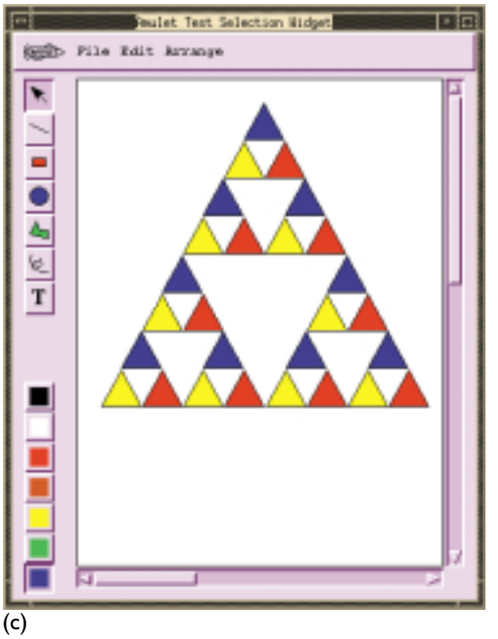


(a)

(a) The dialogue box for generalizing objects in Topaz, with the script window in the background. Users select the generalizations they want to apply; using dialog boxes, they create such sophisticated scripts as (b) for creating a "Sierpinski Gasket" and (c) for applying DeMorgan's Law by changing an And gate and a Not gate into two Not gates and an Or gate, then reconnecting the wires.



(b)



(c)

correct (assuming there are no bugs in the software), but the user has to make all the decisions.

The use of inferencing falls into the general category of "intelligent interfaces," a term referring to any user interface with some intelligent or AI component, including demonstrational interfaces with inferencing, as well as other interfaces, including those using natural language. Systems with inferencing are often said to be guessing what the user wants. Many demonstrational systems use heuristics, or the rules they use to try to determine what the user actually means.

The demonstrational systems we've been developing include varying amounts of inferencing. Some don't, creating only an exact transcript of the reactions the user has performed, so these reactions can be replayed identically. Most of our systems provide a limited number of heuristics that try to help users perform their tasks. Our most recent one, called Gamut, employs sophisticated AI algorithms, such as plan recognition and decision-tree learning, to infer sophisticated

behaviors from a limited number of examples.

We've been experimenting with various levels of intelligence in our interfaces. Meanwhile, some human-computer-interaction researchers have criticized the whole notion of intelligent interfaces, arguing they misleed users and remove necessary user control [10]. We've sought to overcome this problem by keeping users in the loop through continuous feedback so they always know what is happening; we've also provided opportunities to review what the system is doing and make corrections. However, the idea of the usability of intelligent agents is still an unsolved area of research, so reviewing the trade-offs in different systems is instructive.

Many of our systems, and most of the programming by demonstration (PBD) systems developed by others, have used fairly simple inferencing techniques. The PBD developer community's expectation has been that, since these systems are interactive, the user is helping the system whenever the system cannot infer the correct behavior. Moreover, as few PBD system developers have been AI researchers, they are reluctant to apply unproven AI algorithms. On the other hand, experience with many PBD systems has shown that users expect the system to make increasingly sophisticated generalizations from the examples the user supplies and that PBD researchers want their systems to be able to create increasingly complex behaviors. Therefore, a trend has developed toward making these systems increasingly intelligent.

Balancing a system's sophistication while managing the user's expectations remains an open problem.

No Inferencing

The Topaz system we developed in 1998 for creating macros, or scripts of actions, in a drawing editor requires the user to explicitly generalize the parameters of the operations using dialogue boxes [8]. For example, Figure 1 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 was able to generalize these objects in various ways. The choices in the dialogue box include: "The run-time object should be whatever object the user selects" and "The objects are the result of a previous command." The only heuristic we built into Topaz is that when a script creates objects and then operates on these objects, the default generalization uses the dynamically created objects whenever the script executes. This heuristic is almost never wrong, so there is little chance the system would guess incorrectly.

Many other PBD systems take a no-inferencing approach, including the seminal Pygmalion, created by the author Smith in 1975 as one of the first systems to exploit example-based programming, and Dan Halbert's SmallStar, created in 1984 to apply example-based programming to the office applications of the Xerox Star.

Since there is no inferencing in these systems, they never make mistakes, and the user has complete control. However, the user has to figure out how to get the desired actions to occur, often requiring the clever use of special features, such as searching for objects by way of their properties and the various ways to generalize parameters.

Simple Rule-Based Inferencing

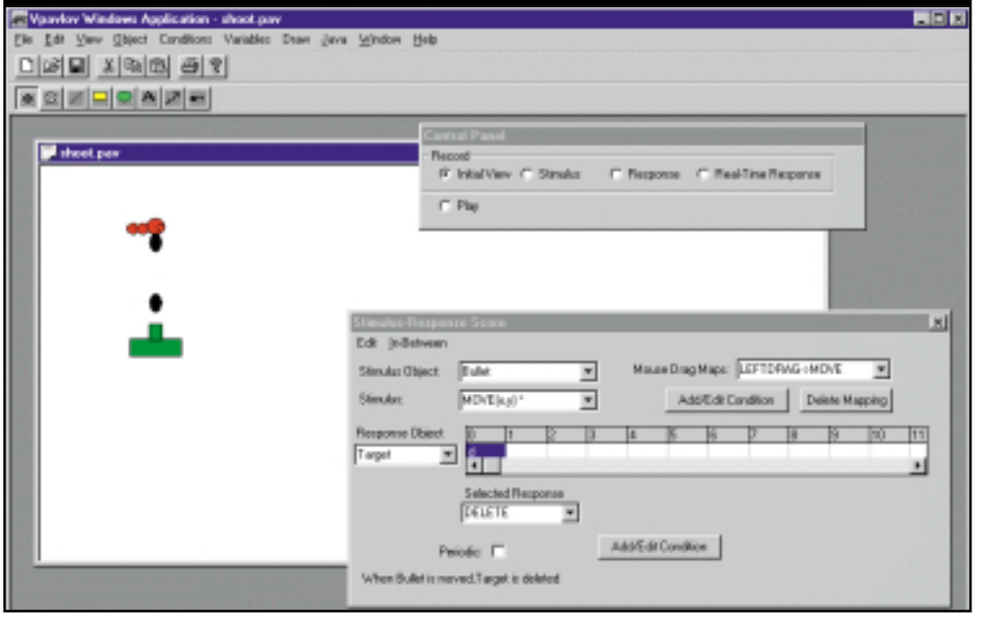
Most of our systems have used simple rule-based inferencing for their generalizations. For example, the early Peridot system we developed in 1987 to create widgets by example uses about 50 hand-coded rules to infer the graphical layout of the objects from the examples [7]. Each rule has three parts, one for testing, one for feedback, and one for the action. The test part checks the graphical objects to determine whether they match the rule. For example, the test part of a rule that aligns the centers of two rectangles checks whether the centers of the example rectangles are approximately centered. Because these rules allow some sloppiness in the drawing, and because multiple rules might apply, the feedback part of the rule asks the user whether the rule should be applied. For example, Peridot might ask something like: "Do you want the selected rectangle to be centered inside the other selected rectangle?" If the user answers yes, then the action part of the rule generates the code to maintain the constraint.

Our subsequent systems have used similar mechanisms, though often without the explicit list of rules we used in Peridot. For example, Tourmaline, which formats documents from example, contains rules that try to determine the role of different parts of a header

in a text document, such as section number, title, author, and affiliation, as well as the formatting associated with each part. The results are displayed in a dialogue box for the user to inspect and correct.

DEMO in 1991 [11] and Pavlov in 1997 [12] are stimulus-response systems that use rule-based inferencing to create interactive programs from examples. The user provides a single example of both a triggering event (the stimulus) and the sequence of operations that should be executed (the response). Rules are then used to infer stimulus-response behaviors. For example, if the user demonstrates a button-click as a stimulus and a transformation as a response, constant parameters are inferred for the response; the transformation is executed exactly as demonstrated. If the demonstrated stimulus and response operations are all transformations (such as "moves" and "rotates"), the system infers a proportion relating the parameters of the response to those of the stimulus.

Figure 2. The Pavlov development environment showing a target shooting game. The user's demonstrations are performed on the drawing canvas (containing the green gun, black bullets, and red targets). The user manipulates the control panel (top right) to inform the system when the initial view is being drawn. A stimulus or response is being demonstrated, or the interface is being tested (played). The dialogue (bottom right) is the stimulus-response score, allowing inferred behaviors to be edited. The Add/Edit Condition button brings up a dialogue showing the conditions that must be true for the stimulus to trigger the response.



Consider the development of a Celsius-Fahrenheit converter, and suppose the user demonstrates the stimulus of moving one slider five pixels and a system response of moving a different slider nine pixels. Since stimulus and response are both transformations, the system infers a proportional stimulus-response behavior. At run-time, when the user drags the first slider any amount, the second slider is modified by 9/5 as much.

Pavlov also includes rules for inferring conditional behaviors. If a stimulus operation is demonstrated (so that some graphical condition is displayed), the system infers that the response should occur only if that condition exists when the stimulus occurs at run time. Suppose the user demonstrates moving a bullet as a stimulus and completes the move (by releasing the mouse) with the bullet intersecting a target (see Figure 2). After a stimulus demonstration, Pavlov compares the stimulus object with other objects, looking for graphical relationships. It then displays a list of the relationships it finds in a dialogue box, and the user gets to choose which of them the system should now use to control the bullet's behavior. In this case, the user chooses `Bullet.Intersects(Target)`. The user then demonstrates the response (such as deleting the target and incrementing a score). At run time, the response is executed only when the bullet moves and intersects the target.

Pavlov also contains rules, so the objects referred to in behaviors need not be constant. If the user demonstrates an operation on a dynamically created object, that is, one created as a response to a stimulus, then Pavlov infers a complex object descriptor. In the bullet-target example, bullets and targets are created in response to some stimulus, say, pressing the up-arrow, thus causing a bullet to be created and "shot" at the target. When the user demonstrates the move-bullet stimulus, the system infers that moving any bullet (at run time) should cause the response.

When the target is deleted in the response demonstration, the system identifies the target as the one intersecting the bullet. The response object is thus described by Pavlov as "all targets intersecting bullet."

Many PBD systems from other developers use the rule-based approach from single examples, including Mondrian [3], which creates procedures in a drawing editor.

An important advantage of such rule-based heuristics for generalizing 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 bases its guess on only a single example. More complex behaviors are either not available or must be created by editing the code generated by the PBD system. Since predetermined behaviors are the only behaviors available, these behaviors in some cases might instead be made available to the user through a direct-manipulation interface, such as a menu, to avoid the problems of inferencing.

An important challenge for PBD system designers is how to let users understand and edit their programs without obviating the benefits of PBD.

Another challenging issue for PBD system developers and researchers working with rule-based systems is whether the user is allowed to change the rules. Most existing PBD systems use a fixed set of rules created by the designer. In many cases, the users themselves 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 them to be able to specify new rules. However, if users could write the code for the rules, they would probably not need PBD support. Because no one has produced a system allowing new rules to be entered into a PBD system, users' rule control represents an interesting meta-problem.

Sophisticated AI Algorithms

Recognizing the limitations of single-example, rule-based approaches, and wanting to create more sophisticated behaviors, our Gamut system infers behaviors from multiple examples to create games and other interactive applications [5].

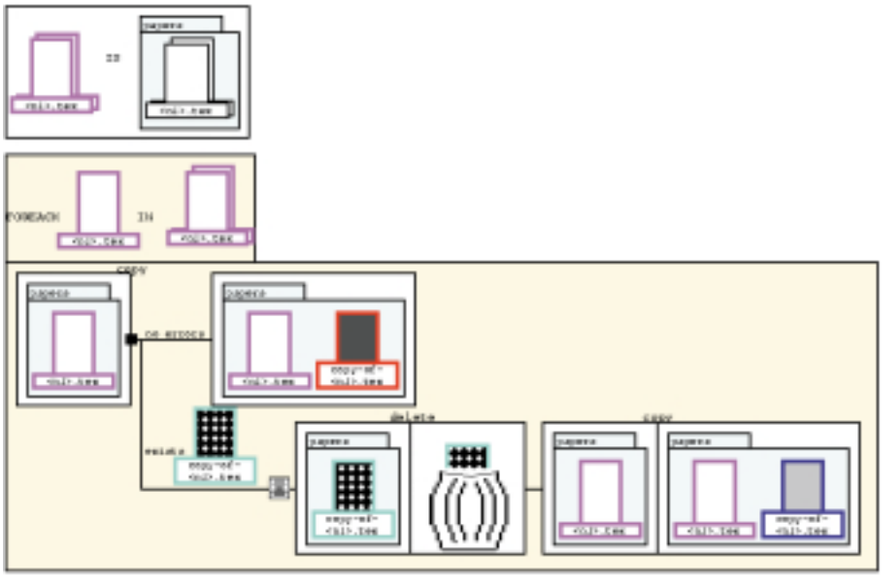
Gamut begins building a new behavior in much the same way the single-example systems build a new behavior. However, each time the developer refines a behavior by adding a new example, Gamut uses metrics to compare the new example with the current behavior and decides how the behavior should be changed. Therefore, if Gamut sees that parts of the code that were previously executed are no longer being used in the new example, it encloses that code in an if-then statement. If Gamut sees that an object is being set with a different value, it uses the new value (along with heuristics) to change the code to produce 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. Guide objects are visible at design time but disappear at run time. The developer can also highlight objects to give the system hints that then guide the heuristic algorithms when Gamut selects 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.

PBD systems developed by others have also worked from multiple examples. Some script-based systems, such as David Maulsby's MetaMouse [4] and Allen Cypher's Eager [1] compare multiple executions of commands to identify 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 numbers offset by a constant factor. Martin Frank's InferenceBear [2] can generate more sophisticated behaviors by supporting linear equations to compute the parameters, letting the user provide negative examples to show the system when behaviors should not be performed.

When inferring from multiple examples, systems should be able to recognize both positive and negative examples. In the simplest case, a positive example demonstrates a condition in which a behavior

Figure 3. A program in Pursuit that looks for all the files in the directory that match "*.tex"; it tries copying the files for each of them. If an error occurs during copying, because, say, the resulting file already exists (lowest branch), then the old file is deleted and the copying is done again.



should occur; a negative example shows when the behavior should not occur. Without negative examples, a system cannot infer many behaviors, including those using a Boolean-OR. In Gamut, the developer can use either the "Do Something" button or "Stop That" button to create a new example. These buttons correspond roughly to creating positive and negative examples.

In some other systems, negative examples are recognized implicitly. For instance, when MetaMouse detects a behavior is repeating, the system uses the actions that are not repeated in the current iteration as a *negative* example. This negative example then signals the system that a conditional branch is required. InferenceBear requires users to explicitly note negative examples, a requirement that has proven difficult for users to understand.

The primary advantage of using more sophisticated techniques is the system can infer more complex behaviors. The disadvantages include the system is much more difficult to implement and may still not infer correctly. There is a slippery slope for all intelligent interfaces, including PBD systems. Once the system shows a little intelligence, users who don't know how the system was designed may expect it to be able to infer as much as a human would, though such leaps are well beyond the state of the art. Balancing the system's sophistication while managing the user's expectations remains an open problem.

Feedback for Control

User studies of the Eager system have found that users are reluctant to let the system automate their tasks, because it provides no feedback about what the inferred program would do and what the program's stopping criteria are [1]. Another important issue is how users are allowed to edit the program when changing their minds about what they want it to do.

It is clear that some form of representation of the inferred program is desirable. But since users of PBD systems are generally viewed as nonprogrammers, it seems inappropriate to require them to read and understand code. If they understood the code, they might be able to write it in the first place—and wouldn't need PBD. In some cases, however, the goal of PBD is to help users get started with the language. Since it is easier for people to recognize than to recall things, seeing the code generated from the examples may help them learn the language, giving them a start on their programs. The "record" mode for creating macros by example in spreadsheets and other programs helps users get started learning the macro language.

Nonprogrammers have difficulty understanding the code generated by a PBD system and modifying it when it isn't exactly correct. Therefore, an important challenge for PBD system designers is how to let users understand and edit their programs without obviating the benefits of PBD.

Peridot's use of question-and-answer dialogs to confirm each inference proved to be problematic, because people tended to answer yes to every question, assuming the computer knew what it was doing. Peridot also had no visible representation of its code after it was created, so there was no way to check or edit the resulting program.

Our Pursuit system, developed in 1993, creates macros of file-manipulation actions for demonstrations. It focuses on the issue of feedback, using a novel graphical presentation that showed before-and-after states based on a "comic-strip" metaphor. Its domain is manipulation of files in a "visual shell," or desktop, and the visual presentation shows examples of file icons before and after each operation (see Figure 3 for a relatively complex program in Pursuit). The same visual language allows users to confirm and repair the system's inferences (such as how to identify the set of files to operate on) and facilitates the editing of programs later if, say, the user wants a slightly different program for a new task.

User studies of Pursuit have found that nonprogrammer users more easily create correct programs using Pursuit's built-in graphical language than with an equivalent textual representation; they also make relatively complex programs containing conditionals and iterations.

Our Marquise system [9], also developed in 1993, generates a textual representation of the program, while the user creates graphical editors from examples (see Figure 4). The evolving program is represented as sentences, with each available choice represented by a button embedded in the sentence. Clicking on a button provides alternate options; changing the option might also change the subsequent parts of the sentence.

Experience with this form of feedback revealed a number of problems. Implementers found it difficult to design the sentences so they were readable and still contained the correct options. When users wanted to change the behavior in significant ways, they often found it difficult to figure out which button to click on.

The Topaz system [8] uses a relatively straightforward representation of the program, as shown in Figure 1. The operation name displayed in the script is often the same as the menu item the user selected (such as "To Top"). For the script, Topaz supports all the usual editing operations, including cut, copy, and paste. Clicking on a parameter brings up the dialogue box used to generalize that type of parameter. The result is that this representation seems easy to understand and edit.

Pavlov focuses on interactive animation, providing a timeline-based view, in the style of MacroMedia's Director, but based on an event-based model [12], as shown in Figure 2. Because it provides a separate timeline showing the (animated) response to each stimulus, more complex animated interfaces can be described than in the traditional single-timeline approach.

Like Eager, Gamut offers no visible presentation of the inferred program, though users have pointed out that one would be desirable [5]. Gamut gets around the editing problem by allowing users to supply (whenever they want to) new examples that modify existing behaviors, so they never have to edit code directly.

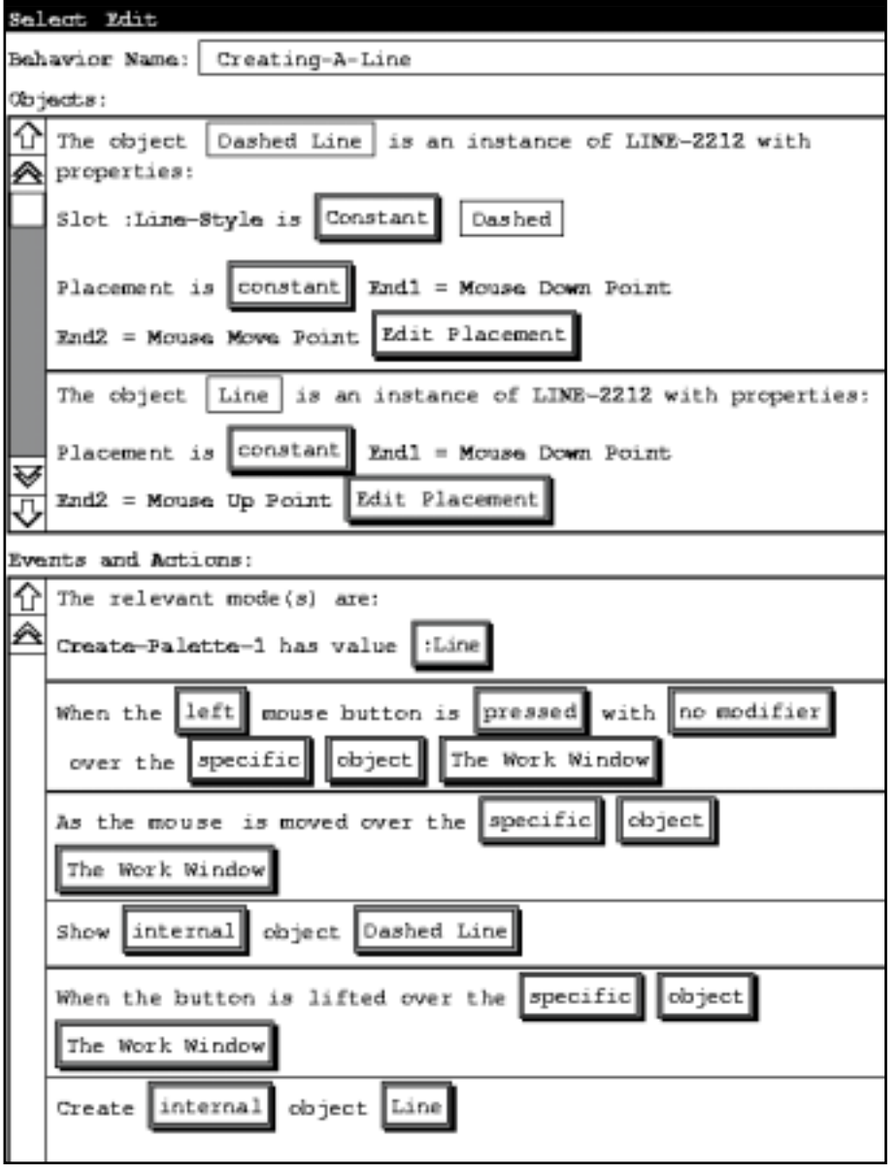
Developers of many other systems have researched different presentations of the generated program. Most use custom languages designed specifically to match the PBD system. For example, SmallStar provides a textual representation of the program created from the examples, and the user has to explicitly edit the program to generalize parameters and add control structures. For its representation, InferenceBear uses a novel form of event language called "Elements, Events & Transitions" whereby the actions and their parameters are represented as event handlers [2]. (This relatively complex language is probably not accessible to nonprogrammers.) Mondrian provides a comic-strip-style visual program [3], similar to the one in Pursuit.

Gamut gets around the editing problem by allowing users to supply new examples that modify existing behaviors, so they never have to edit code directly.

Conclusion

Designing the heuristics for a PBD system is difficult. 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 require more elaborate user interfaces to control the inferencing. These systems are much more difficult to implement. In any kind of PBD system, a well-designed feedback mechanism should be included, so users can understand and control what the system is doing and change the program as needed. Much more research is needed on the appropriate level of intelligence and how feedback is delivered to users in PBD systems. ■

Figure 4. 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; lifting the Mouse button draws a solid line. The embedded buttons represent options that can be used to edit a behavior.



References

1. Cypher, A. *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, Mass., 1993.
2. Frank, M. and Foley, J. A pure reasoning engine for programming by demonstration. In *Proceedings of UIST'94: ACM SIGGRAPH Symposium on User Interface Software and Technology* (Marina del Rey, Calif., Nov. 2–4). ACM Press, New York, 1994. pp. 95–101.
3. Lieberman, H. Dominos and storyboards: Beyond icons on strings. In *Proceedings of the 1992 IEEE Workshop on Visual Languages* (Seattle, Wa. Sept. 15–18). IEEE Computer Society Press, Los Alamitos, Calif., 1992, pp. 65–71.
4. Maulsby, D. and Witten, I. Inducing procedures in a direct-manipulation environment. In *Proceedings of SIGCHI'89: Human Factors in Computing Systems* (Austin, Tx., Apr. 30–May 4). ACM Press, New York, 1989, pp. 57–62.
5. McDaniel, R. and Myers, B. Getting more out of programming by demonstration. In *Proceedings of CHI'99: Human Factors in Computing Systems* (Pittsburgh, Pa., May 15–20). ACM Press, New York, 1999, pp. 442–449.
6. Modugno, F. and Myers, B. Visual programming in a visual shell: A unified approach. *J. Vis. Lang. Comput. 8,* 5/6 (Oct./Dec. 1997), 276–308.
7. Myers, B. Creating user interfaces using programming by example, visual programming, and constraints. *ACM Transact. Program. Lang. Syst. 12,* 2 (Apr. 1990), 143–177.
8. Myers, B. Scripting graphical applications by demonstration. In *Proceedings of SIGCHI'98: Human Factors in Computing Systems* (Los Angeles, Apr. 18–23). ACM Press, New York, 1998, pp. 534–541.
9. Myers, B., McDaniel, R., and Kosbie, D. Marquise: Creating complete user interfaces by demonstration. In *Proceedings of INTERCHI'93: Human Factors in Computing Systems* (Amsterdam, The Netherlands, Apr. 24–29). ACM Press, New York, 1993, pp. 293–300.
10. Shneiderman, B. Looking for the bright side of user interface agents. *ACM Interact. 2,* 1 (Jan. 1995), 13–15.
11. Wolber, D. and Fisher, G. A demonstrational technique for developing interfaces with dynamically created objects. In *Proceedings of UIST'91* (Hilton Head, S.C., Nov. 11–13). ACM Press, New York, 1991, pp. 221–230.
12. Wolber, D. An interface builder for designing animated interfaces. *ACM Transact. Comput.-Hum. Interact. 4,* 4 (Dec. 1997), 347–386.

These systems were developed under many research grants, including those 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 described here are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

Brad A. Myers (bam@cs.cmu.edu) is a senior research scientist in the Human Computer Interaction Institute in the School of Computer Science at Carnegie Mellon University, Pittsburgh, Pa. **Richard McDaniel** (richm@ttb.siemens.com) is an innovator in the Siemens Technology to Business Center in Berkeley, Calif. **David Wolber** (wolber@usfca.edu) is an associate professor in the Department of Computer Science at the University of San Francisco.