Chapter Sixteen

Stimulus-Response PBD: Demonstrating "When" as Well as "What"

David W. Wolber and Brad A. Myers
University of San Francisco and Carnegie Mellon University

Abtract

Many programming by demonstration (PBD) systems elaborate on the idea of macro recording, and they allow users to extend existing applications. Few, however, allow new interfaces to be created from scratch because they do not provide a means of demonstrating when a recorded macro should be invoked. This paper discusses stimulus-response systems that allow both the when (stimulus/event) and the what (response macro) to be demonstrated.

16.1 Introduction

When a client hires a programmer, he generally will meet with her to show how he wants the proposed application to behave. Using a graphics editor, scratch paper, or even a dinner napkin, the client sketches the proposed interface and then walks the programmer through its behaviors, first playing the role of the end user and showing what actions he can take, and then playing the role of the system and showing how it will respond to a particular stimulus.

The goal of our research has been to automate this process, to build software tools that do the job of the human programmer in such a scenario. We envision a system that watches a client demonstrate behaviors and automatically creates the desired application.

16.1.1 PBD: An Elaboration of Macro Recording

Our research is based on the programming-by-demonstration (PBD) systems described in Cypher (1993), including Eager (Cypher 1991), SmallStar (Halbert 1984), Chimera (Kurland and Feiner 1991), Mondrian (Leiberman 1993), and Peridot (Myers 1990). These systems elaborate on a concept familiar to many computer users—macro recording. With macro recording, the user tells the system to watch him execute operations so they can be recorded and played back later. The goal is to give a name to a sequence of operations so the user won't have to repeat those operations again and again.

PBD elaborates on the concept of macros by generalizing the sequence of operations demonstrated. For instance, with Lieberman's (1993)

Mondrian system, a user can demonstrate the drawing of three rectangles around a square. Mondrian records the three DrawRectangle operations but generalizes them with parameters so that the recorded macro can be used to draw an arch around any square of any size.

16.1.2 PBD Macro Invocation

Once a macro is created, there must be a way to invoke it. Mondrian creates a new item in the palette of operations—the user just clicks the item to invoke the macro. Other systems just allow the user to execute a previously recorded macro by choosing it from a menu or list.

A standard method of invoking previously recorded macros is sufficient to help a user eliminate repetitive tasks in existing applications. But, as outlined in an earlier article (Kosbie and Myers 1993), sometimes a user would like to create a macro and have it invoked when a particular event occurs. For instance, a user might want to extend her desktop so that the next day's calendar is printed out (the macro) when she logs out (the macro invocation event).

Eager(Cypher 1991) and its successors (see Chapters 14 and 15) focus on macro invocation by combining PBD with a predictive interface. They watch a user work and learn to invoke macros after the user has explicitly executed its first few operations (i.e., they finish the job for the user).

But whereas the goal of these systems is to extend existing applications, our goal is to allow users to build complete interfaces starting with a blank canvas. Thus, we must provide mechanisms so that a designer can *explicitly* specify both macros and the events that invoke them.

To facilitate this goal, we have built a number of systems based on a two-phase process called *programming by stimulus-response demonstration* (McDaniel and Myers 1999; Myers, McDaniel, and Kosbie 1993; Wolber and Fisher 1991; Wolber 1997). With this process, the user first demonstrates the stimulus (invocation event), then demonstrates the response (macro procedure) that it invokes. The system then infers a behavior so that, at run time, when the stimulus is executed, it triggers the corresponding response.

The inclusion of macro invocation complicates PBD for both the designer and the underlying system. Macro recording is already difficult for many users, as it is easy to forget whether the *Record* button is on. Adding another mode button for "Record stimulus" complicates matters even more.

Furthermore, the events that can trigger activity are more often more complicated than the activity itself. Actions can be triggered by many different type of events, including low-level actions (e.g., mouse and keyboard actions), high-level operations (e.g., the rotation of an acceleration gauge can speed up a car), or the interface being in a particular context (a bullet intersecting a target).

Thus, it is a challenge to provide a simple "syntax" for demonstrating behaviors. It is also a challenge to provide a powerful enough generalization mechanism to infer the response operations and their parameters. Generalization is complicated because a response can be related to the stimulus and its parameters. For instance, a car responds (speeds up) an amount proportional to how much the acceleration gauge is modified.

16.1.3 Augmenting the Capabilities of Traditional Interface Builders

Stimulus-response demonstration can augment the capabilities of interface builders such as Visual Basic (Microsoft).[1] These popular tools significantly decrease the time and expertise necessary to define the *layout* of an interface, but the designer must still code its dynamic *behavior.* For instance, a Visual Basic designer can easily change the static location of an object in the interface by dragging it, but specifying that the object should be moved at *run time* in response to some stimulus (e.g., a button click) requires coding.

Our strategy is to provide the designer with a complete set of widget, text, and graphics operations, along with the capability of using them for both drawing the interface and demonstrating its dynamic behavior. With such a capability, the designer is not restricted to wiring together standard widgets such as menus, list boxes, and buttons but can instead draw arbitrary graphics and demonstrate how they respond to stimuli (we call these *application-specific behaviors*). Such capabilities allow nonprogrammers to create animated and graphical applications and not be limited to standard business-type applications.

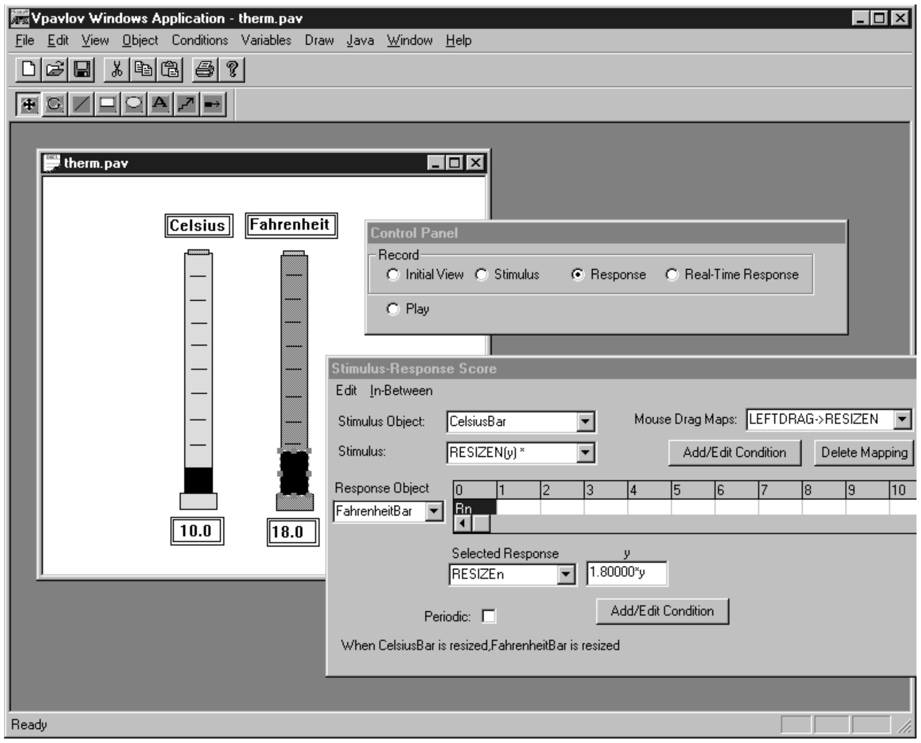
16.1.4 A Quick Example

Consider how stimulus-response demonstration is used to create a Celsius-Fahrenheit converter in the Pavlov system (Wolber 1997, 1998). The designer first draws two thermometers, as shown in Figure 16.1. Note that the thermometers in Figure 16.1 consist of raw graphics—they are not widgets with predefined behavior. After drawing the thermometers, the designer

S
R
L

1. msdn.microsoft.com/vbasic.

FIGURE **16.1**



Record: Initial View Stimulus [✓ Response] Real-Time Response

Stimulus Object: CelsiusBar

Stimulus: RESIZEN(y) *

Mouse Drag Maps: LEFTDRAG->RESIZEN

Response Object: FahrenheitBar

Selected Response: RESIZEn

y: 1.80000*y

Periodic: ☐

When CelsiusBar is resized,FahrenheitBar is resized

Pavlov development of a Celsius-Fahrenheit converter, with the control panel (upper right) as a video-recorder-like mode palette that allows the designer to draw, record stimuli, record responses, or "play" interfaces. The Stimulus-Response Score (lower right) displays a high-level representation of the behaviors that have already been demonstrated.

uses the same drawing operations to demonstrate the interface's behavior. He first tells the system he is demonstrating a stimulus—he selects "Stimulus" mode in the control panel—then drags (resizes) the Celsius indicator ten pixels up. Then, in "Response" mode, he demonstrates how the system should respond to the stimulus: he stretches the Fahrenheit indicator eighteen pixels along the *y*-axis (he might also modify the text box below the Celsius thermometer if he hasn't already demonstrated this behavior previously).

From the demonstrations, the system infers a generalized behavior. In this case it infers that any time the Celsius bar is moved, the Fahrenheit bar should be moved 9/5 (1.8) as much. When the interface is executed and the end user moves the Celsius bar, the Fahrenheit bar is moved the proportional amount.

16.1.5 Wait a Second!

An example like the Celsius-Fahrenheit converter inspires more questions than it answers. How does the system know to infer the 9/5 proportion? Can the end user move the Celsius indicator outside its enclosing box? How would such a restriction be demonstrated? If the designer doesn't like what the system infers, can she change it? Can animated behaviors and timing be specified? Can more complex stimuli be demonstrated, such as the behaviors of a PacMan character?

Trying to answer such questions is the challenge of our research. This chapter introduces what we have learned over the last decade in building our early systems (DEMO [Wolber and Fisher 1991] and Marquise [Myers et al. 1993]) and their successors (Pavlov [Wolber 1997] and Gamut [McDaniel and Myers 1999], respectively). Our goal is a system that allows the behavior of any interface to be defined without coding. There has been some significant progress: stimulus-response demonstration can be used to create such complex interfaces as diagram editors, driving simulators, shooting arcades, board games, and even PacMan. However, few of these ideas have appeared in commercial interface builders, and much research is still needed to reach the ultimate goal.

In this chapter, we'll view stimulus-response as a kind of programming language, in which the language consists of behavior demonstrations instead of code. Thus, we'll provide an overview of alternatives for both the syntax and the semantics of stimulus-response languages. We'll also discuss various alternatives for providing feedback to designers so that they can view and edit a high-level representation of the behaviors that have already been specified.

16.2 The Syntax of Stimulus-Response

The syntax of a stimulus-response system is the mechanism by which a user presents the "example" behaviors to the system. Providing a usable syntax is a challenge because the system is providing the end user with much more power than most software—it is allowing him to create software, not just use it. The key difficulty is that a user must perform the same actions both to use the system and to demonstrate the behavior of the target software being created.

Perhaps the most straightforward syntax is to provide a video-recorder-like mode palette as shown in Figure 16.1 and used in DEMO, Pavlov, and Marquise. The designer uses "Initial View" mode to draw the *initial* interface. The system keeps a record of the initial state of each graphic, so that it can snap back to it after the designer demonstrates stimuli and responses. Pavlov reverts to this initial state whenever Initial View or "Play" mode is selected.

In *Stimulus* mode, the designer plays the role of end user and demonstrates an event. This event can be as simple as a button click or key press, or it can be one of the drawing operations (e.g., "Move," "Rotate"). It can also be contextualized; that is, the designer can demonstrate a graphical condition that must be true before a response is triggered.

Often the stimulus is something the end user initiates directly at run time, but not always. For instance, one stimulus might trigger the execution of a series of response operations, and one of those operations in turn might trigger some other response. Thus, the designer would demonstrate two behaviors, the first with an "end-user" stimulus, the second with a "system" stimulus. The different types of stimuli are discussed in detail later in this section.

After a stimulus is recorded, the system automatically changes the mode to Response mode. In this mode, the designer plays the role of the system by executing the operations that should occur in response to the stimulus. To end the response sequence, the designer changes to any other mode. Play mode allows the interface to be executed. In some systems, the development menus and palettes disappear in Play mode, leaving only the target interface.

16.2.1 Eliminating Modes

Conventional interface builders such as Visual Basic have two modes: "build" and "test." Environments like the one described earlier add at least two more to the mix: "record stimulus" and "record response." We have found that users have significant difficulty keeping track of the current mode when these additional modes are added.

Some systems provide alternative syntax that eliminate one or more of the modes. Chimera (Kurlander and Feiner 1991) effectively eliminates response mode by recording all actions in a history list and requiring the designer, after the fact, to signify which actions in the list should be part of a macro. Such a scheme could be applied to a stimulus-response system by requiring the designer also to signify an operation in the history list as the stimulus.

Peridot (Myers 1990) provides an onscreen virtual mouse that eliminates some of the need for a stimulus mode. Instead of choosing stimulus mode and using the physical mouse to demonstrate what the end user might do, the Peridot designer drags the logical mouse to the object and manipulates it.

Gamut (McDaniel and Myers 1999) eliminates stimulus mode by replacing the four-mode palette with two buttons: “Do Something” and “Stop That” (see Fig. 16.2). When the user performs an action that is supposed to do something and it doesn’t, she hits the Do Something button. The event just prior to pushing the button is assumed to be the stimulus. The system is then in Response mode until the user hits “Done” (the Done button appears when the system is in Response mode). Stop That is used to give negative examples when the system does something wrong; it also puts the system in response mode, and it tells the system to eliminate the response(s) that were previously recorded. Besides eliminating stimulus mode, the Gamut scheme also eliminates the distinction between build and test mode by having the system always be in test mode (i.e., any previously defined behaviors are always active, even when the designer is editing).

16.2.2 Demonstrating Stimuli

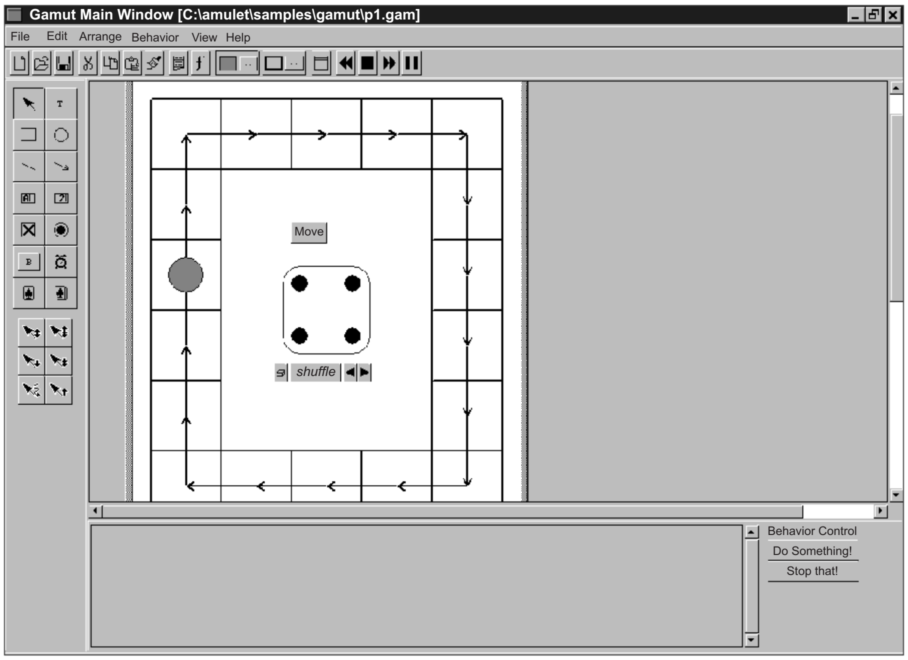
Various systems have allowed any of the following stimuli to trigger a response in an interface:

- an end user manipulating the mouse or keyboard (physical operations),
- the execution of some drawing editor operation, such as Move or Rotate (logical operations),
- objects in the interface exhibiting some state or context,
- the passage of time, or
- the execution of some application event.

The syntax for demonstrating each of these stimuli are discussed in the following sections.

S
R
L

FIGURE **16.2**


Gamut Main Window [C:\amulet\samples\gamut\p1.gam]
File Edit Arrange Behavior View Help
Move
shuffle
Behavior Control
Do Something!
Stop that!

A Gamut screen creating a board game. The graphics, text, and widget operations are accessed in the menus (top and left); the behavior control panel (lower right) provides "Do Something" and "Stop That" buttons that both initiate response demonstrations.

Physical and Logical Operations

One key aspect of PBD interface builders, as opposed to PBD macro systems that extend existing applications, is that the base system (e.g., the drawing palette) menu, and other mechanisms *do not appear at run time.* Thus, unless the designer explicitly demonstrates that end users can perform an operation in the target interface, they won't be able to.

For example, during development, the designer can rotate any object by choosing Rotate in the drawing mode palette. But if the designer wants the end user to be able to rotate that object, she must demonstrate a stimulus-response behavior that maps a mouse drag to a rotate operation.

Some systems require this physical (mouse drag) to logical (Rotate) mapping to be explicitly demonstrated. In DEMO (Wolber and Fisher 1991), when the designer clicks on an object in stimulus mode, she is prompted to choose between the mouse operations up, down, drag, enter, and exit. The designer selects one and then demonstrates the response that should be linked to the chosen stimulus.

Gamut provides special mouse operation icons that can be dragged onto an object to demonstrate the physical operation. (These are visible at the left of Figure 16.2, below the drawing palette). Since there is an icon for each type of operation, no choice dialogue is necessary.

If the physical → logical mapping is explicit, the designer needs two demonstrations for behaviors such as the Celsius-Fahrenheit program: one to specify the physical mapping:

LeftMouseDrag → CelsiusBar.Move

and one for the logical mapping:

CelsiusBar.Move → FahrenheitBar.Move.

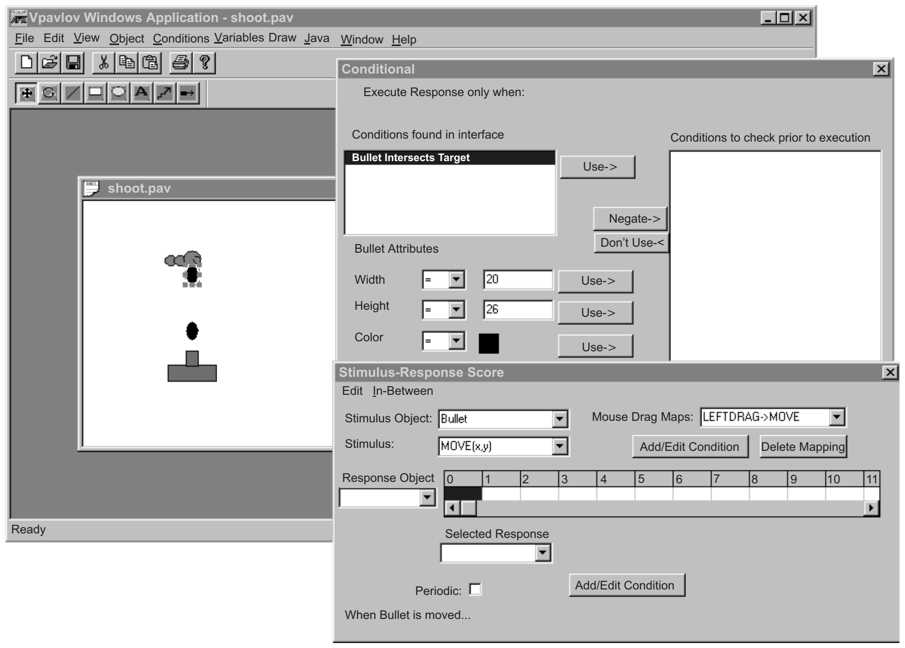
Pavlov eliminates the need for two demonstrations in such cases. When the designer moves an object in stimulus mode, the system implicitly infers the drag → Move mapping, and also records that the Move should be mapped to the upcoming response operation(s). The disadvantage of inferring the physical-logical mapping is that sometimes the designer only wants to map a logical stimulus to logical response(s) and doesn't want the physical-logical mapping. For instance, in a shooting arcade game, the designer demonstrates that moving a bullet (and hitting a target) should cause a response of deleting the target. In Pavlov, when the bullet.Move stimulus is demonstrated, the system infers a physical-logical mapping (i.e., that the end user can move the bullet). To restrict the end user from doing this, the designer must use the editor to delete the mapping.

Context-Triggered Behaviors

Sometimes the activity in an interface should be executed in response not to an operation but to the interface reaching a particular context or state. For instance, in the shooting arcade game, a target should be deleted only when a bullet intersects the target, as shown in Figure 16.3.

Such context-based stimuli are the focus of graphical rewrite systems such as Creator, which can be considered as a type of stimulus-response

FIGURE **16.3**



Execute Response only when:

Bullet Intersects Target

Width = 20
Height = 26
Color =

Stimulus Object: Bullet
Stimulus: MOVE(x,y)
Mouse Drag Maps: LEFTDRAG->MOVE
Periodic: ☐
When Bullet is moved...

Pavlov development of an arcade game. The condition dialogue (upper right) appears when the system identifies a condition during a stimulus demonstration.

system. Demonstrations in these systems consist of selecting a portion of the interface (a set of cells in the grid) that make up a "before state" and then manipulating the objects residing in the selected portion of the grid to show an "after state."

In terms of stimulus-response, the stimulus is the before picture instead of an operation. At run time, the system continually tests to see whether the demonstrated "before picture" occurs. When it identifies such a state, it executes the operations required to put the interface in the demonstrated "after state."

Like Creator, Pavlov allows context to be demonstrated, but it does not provide a special "demonstrate before-state stimulus" mode. Instead, it integrates context directly into its stimulus-response model. The rationale is that an interface does not magically enter a particular state but will only do so as a result of some operation. Thus, the designer always demonstrates a triggering stimulus operation, but during or prior to this demonstration, she puts the system in the state that must be true for the stimulus to trigger its response.

Consider again the shooting arcade game in Figure 16.3. In Pavlov, the designer demonstrates a move of a bullet as the stimulus but completes the move (releases the mouse) so that the bullet intersects a target (see Fig. 16.3). The system records the move stimulus and identifies conditions, such as “intersect,” relating the objects in the interface. Textual descriptions of these conditions are listed so the designer can choose which should be used as part of the stimulus. In this case, the designer selects *Bullet.Intersects(Target)* and then demonstrates the deletion of the target as the response. At run time, every time a bullet moves, the system checks the intersect condition and executes the response only if the condition is true.

Since all conditions are subordinate to a stimulus operation, there is a run-time efficiency gain compared to traditional rewrite schemes, because the system need only check a particular condition when the stimulus is executed, instead of after every time stamp. There is a syntactical disadvantage, however. If different stimuli can cause the same result (e.g., there are two objects, and movement of either might lead to a situation where they intersect), then the designer must demonstrate two separate behaviors.

Both Creator’s graphical rewrite rule scheme and Pavlov’s stimulus-response scheme have limitations in terms of the kinds of context that can be demonstrated. Creator restricts context specification (and object movement) to discrete grid coordinates (e.g.,the target game behavior might be described as “trigger the response when the bullet is *one square below the target*”). This grid restriction simplifies the syntax, but it also limits the conditions and behaviors that can be specified.

Though Pavlov’s scheme isn’t restricted by a grid basis, it is limited by the set of conditions or object relationships (e.g., intersect, encloses, etc.) for which it searches. It also requires the designer to specify and, or, and not relations in a dialogue when a condition is complex; in a system using graphical rewrite rules, the designer can specify such complex conditions with a single picture.

Gamut, like Pavlov, bases its context identification on object attributes and relationships and not on grid-based relationships. A general goal of Gamut is to eliminate the need for the designer to choose the correct generalization (e.g., condition), as is done in Pavlov. The reasoning is that the representation of context, whether textual or graphical, is inherently complex, so choosing can be difficult. Gamut’s philosophy, similar to that of InferenceBear (Frank and Foley 1995), is to allow the designer to perform multiple demonstrations of the same context. The system then uses artificial intelligence (AI) techniques to infer the intended context from the samples. Gamut also asks the designer to provide hints by selecting objects that are relevant to the context. The hints allow more complex conditions to be inferred than are possible in other systems (see Chapter 8).

It should be noted that the idea of demonstrating context is also important in Web information extraction systems. These systems allow designers to build new pages that are composites of information extracted from various other Web pages. Because Web pages change so rapidly, the difficulty lies in describing the information to be extracted so that even if the Web page changes, the description is still valid. Chapters 4 and 5 in this book explore techniques for automatically generating such descriptions from a user's demonstrations.

Time as a Stimulus

Sometimes activity should occur at a certain time, rather than in response to an external event. Gamut allows the designer to drag a special clock widget into the background window and then demonstrate a tick as a stimulus. Pavlov provides a special stimulus called "beginning of execution" that a designer can specify as the current stimulus. He can then select a time frame and demonstrate responses that, at run time, are executed at a particular time.

Structured Text as a Stimulus

Sometimes activity should be triggered when the user types in a piece of structured information. For instance, when a user enters a street address, an interface might automatically open a map with the address as the destination. The work described in chapter 2 allows a designer to demonstrate example instances of structured types and then automatically build a grammar that can recognize new instances.

System Events as Stimuli

An earlier article (Kosbie and Myers 1993) proposed a generalized event-based invocation scheme for stimulus specification. In this scheme, the system would register events for both low-level user actions (e.g., mouse and keyboard clicks) and the high-level interpretations (e.g., menu item *Delete* selected, or object A moved), and macros created by demonstration could be invoked when these events occur. This would allow such behaviors as creating a macro that copies files to a backup area that is invoked before a delete operation or printing out tomorrow's calendar before logging out.

16.2.3 Demonstrating Responses

A *response* is any action that creates, transforms, or deletes an object in an interface. Though most PBD research systems are based on fairly rudimentary graphics editors, the idea is that a commercial system would provide a full range of graphic and text-editing capabilities, and even access to a database and its operations.

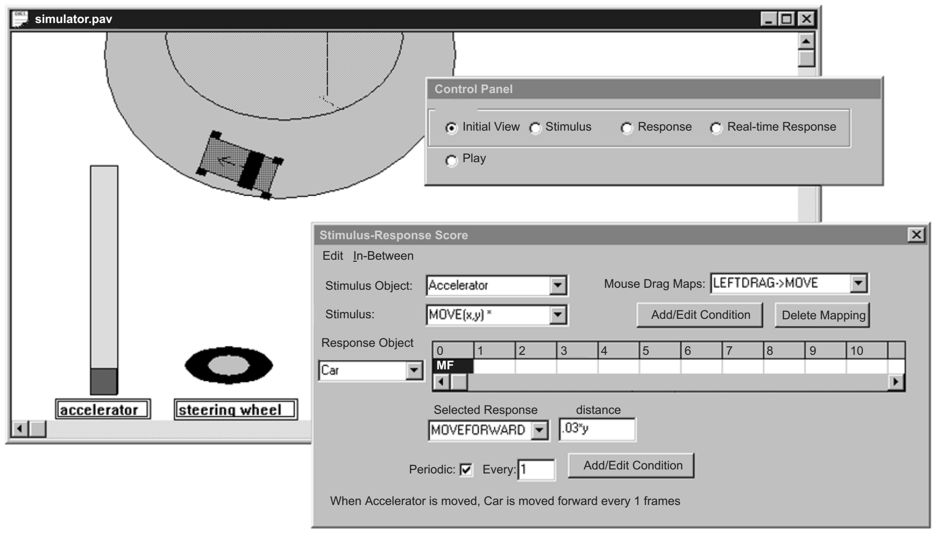
Pavlov combines PBD with some animation mechanisms that allow timing to be specified on behaviors. Using Pavlov's time line, the designer can specify a time stamp, relative to the stimulus, for a demonstrated response. Pavlov also provides a special mode, similar to one in Macromedia's Director (version 7.0) called *Real Time Response* mode. When this mode is chosen, the system records a sequence of time-stamped operations (an animation path) as the designer demonstrates an operation. The designer can also specify animation by stipulating that a response be executed *periodically* at run time.

Pavlov also allows a designer to specify the direction (nose) of an object using a guide object. Consider, for instance, the development of a driving simulator (Fig. 16.4). After drawing the car, accelerator slider bar, and steering wheel, the designer specifies the "nose" of the car by manipulating a special guide object representing its direction. She then demonstrates a stimulus of moving the accelerator slider bar, and a response of moving the car, and marks the response as periodic. Because the car has a notion of direction, its movement is restricted—the designer can only demonstrate that it moves forward or backward relative to the direction vector. In Pavlov, the response is recorded not as a Move(x,y) but as a (periodic) MoveForward operation. At run time, the car goes faster or slower as the accelerator is manipulated.

16.2.4 Demonstration Aids: Guide Objects and Ghost Marks

A guide object is one the designer uses during development but doesn't want to appear at run time (Fisher, Busse, and Wolber 1992). For instance, the designer might use a directed line to show how a game piece can move around the board. In Gamut, the designer simply sets an object property to specify that it is a guide object. It then appears in pastel colors during

FIGURE **16.4**



Control Panel: [✓ Initial View] Stimulus Response Real-time Response Play

Stimulus Object: Accelerator
Mouse Drag Maps: LEFTDRAG->MOVE
Stimulus: MOVE(x,y) *
Response Object: Car
Selected Response: MOVEFORWARD
distance: .03*y
Periodic: ✓ Every: 1

When Accelerator is moved, Car is moved forward every 1 frames

development and disappears at run time. In Figure 16.2, the yellow arrow lines in the center of the boxes are guide objects to help show where the pieces can move. When the designer drags a game piece along a yellow line, the system can infer that he is demonstrating a move from one end point of the line to the other (instead of a move by some offset). Thus, behaviors such as moving one (or more) squares in a game can be demonstrated.

Gamut also provides a backstage area in which all objects are guide objects. The objects in this area generally store "computational" information in the form of text boxes or other widgets. For example, a widget might store whose turn it is or the number of lives left. By replacing information traditionally stored in program variables, the need to connect an interface to application code is reduced or eliminated.

Marquise introduced the idea of leaving trails, or ghost marks, of the mouse cursor as it is moved, so the designer can hook responses to points on its path. Gamut extended the idea so that the previous state of an object also appears after it is transformed. Thus, a designer can specify an operation dependent on that previous state (e.g., that Object A is moved to the previous location of Object B after Object B is moved).

16.3 The Semantics of Stimulus-Response

The generalization or inference machine of a stimulus-response system interprets the meaning of the examples a designer provides during the demonstration. It takes the demonstrated stimulus-response examples as input and output behaviors that, if represented as code, would appear something like this:

```
On StimulusObject.Stimulus(<formal parameters>)
{
  if (checkcontext)
     Operation(<ResponseObject>,<other response parame-
          ters>)
     . . .
}
```

Some major challenges in interpreting the examples are as follows:

- Which object(s) is the designer signifying with her demonstrations?
- What should the parameters of the response operation(s) be?
- How should context modify the inferred behavior?

There are two main methods of generalization, based on whether the system allows more than one example of a behavior to be demonstrated. Pavlov generalizes from a single example, in the tradition of early PBD systems such as SmallStar (Halbert 19XX) and Chimera (Lieberman 1993). In AI terms, such single example systems are known as *explanation-based learning systems.*

The philosophy behind using single examples is to keep the inference machine simple so the designer can understand it and then provide a second-level editor that allows the designer to modify generalizations easily. Single-example systems use as much domain knowledge as they can so that the single "guess" is as accurate as possible.

In contrast, Gamut generalizes from multiple examples. In AI terms, multiple-example systems are known as *empirical learning systems*. Because more information is provided with multiple examples, these systems can infer more complex behaviors and need not rely as much on domain knowledge. However, it is more difficult for the designer to understand what the system is doing, and the syntax is also complicated because a

S
R
L

demonstration can conceivably be performed to edit, refine, or append (add another response) to a particular behavior.

Chapter 3 discusses generalization for PBD systems in general, focusing on the question of how much intelligence should be incorporated. This section explores generalization in the context of stimulus-response systems specifically.

16.3.1 Object Descriptor Problem

Perhaps the most important problem in generalizing from examples is the *object descriptor* problem (Halbert 1984)—determining the object or objects that the designer intends to signify when she demonstrates an action on an example object. In terms of the generated code template shown at the top of this section, the problem consists of how to describe the "StimulusObject" and "ResponseObject" of the code.

Sometimes no generalization need occur, as there is a one-to-one correspondence between the *demonstrated object* and the *intended run-time object.* For example, in the Celsius-Fahrenheit example, *FahrenheitBar* is the demonstrated response object, and it is the intended run-time object as well. *FahrenheitBar* is an example of a *constant object descriptor.*

In some cases, however, the designer intends the demonstration object to be representative of some set of objects or some single object that cannot be described with a constant. When the demonstration object is a dynamically allocated object, the object *can't* be described with a constant. Consider, for instance, the dynamically created bullets in an arcade game. A particular development-time bullet won't appear at run time, because no bullets appear until a stimulus occurs. When an action is demonstrated on a bullet (e.g., movement), a nonconstant descriptor *must* be inferred for it.

There are also cases when a nonconstant descriptor is appropriate even though the demonstration objects itself is statically created. For example, consider a board game that has a group of pieces that are statically created. When the designer demonstrates moving a piece, he may be signifying that the piece to move at run time is the one for the player whose turn it is, and he is using the particular piece as a representative of that concept.

Pavlov is a single-example system, so it only infers nonconstant descriptors for demonstrations on dynamic objects, since those must be generalized into nonconstants. Pavlov includes a number of heuristics to guide its generalization of dynamic objects. The most simple inference is that the dynamic demonstration object represents all instances, at run time, that have been created by the same stimulus. For example, suppose that the bullets in an arcade game are created by pressing the up-arrow key. If the designer demonstrates a stimulus of clicking a button, and a response of changing a particular bullet's color to blue, Pavlov infers that at run time, clicking the button changes the color of all bullets (created by an up arrow).

Pavlov does identify special cases that use more distinguishing descriptors. For instance, if a transformation is demonstrated in the same response sequence as a creation (e.g., a bullet is created then moved), then the object descriptor will be "the same instance as the one just created." If a response is demonstrated on the same object as the stimulus, the inferred descriptor is "the same instance as the stimulus." And, as mentioned in Section 16.2.2.2, Pavlov identifies existing conditions that exist when the example is demonstrated, so the designer can add if-then complexity to a descriptor if desired. In practice, Pavlov's special rules and condition identifier cover a wide range of behaviors. But no matter how many heuristics are used, the system will sometimes guess wrong using only a single example. In these cases, second-level editing or coding is necessary.

Because Gamut uses multiple examples, it can infer nonconstant descriptors for static and dynamic objects. After the first example, a constant descriptor is generally inferred, but as more examples are provided, the system analyzes the properties of the example objects to infer more complex descriptors.

Gamut also reduces the number of required examples by asking users to give *hints* about which of the many interface objects are important. With hints, the system can infer object descriptors that depend on objects that are not directly modified by the example action itself. Consider, for example, a two-player board game that has a red piece, a blue piece, and a toggle specifying whose turn it is. To specify the behavior, the designer first sets the toggle to true and demonstrates moving the red piece. He then sets the toggle to false and demonstrates moving the blue piece. At this point the system knows a nonconstant descriptor is needed to describe the piece that should be moved, but it doesn't know where to start. Thus, it asks the designer to select what auxiliary objects, besides the piece, the behavior depends on. When the designer selects the toggle, the system infers an object descriptor such as "the red piece if the toggle is true, the blue piece if the toggle is false."

16.3.2 Response Parameter Descriptors

In addition to the object parameter, PBD systems must infer the other parameters of an operation (see the "<other response parameters> in the code at the start of this section). For example, a move operation has both the object to be moved (as discussed earlier) and the position to which it moves. Again, the simplest inference is that the parameter is constant, but even this situation presents ambiguity: did the designer intend the destination demonstrated or the offset demonstrated (i.e., move to this location, or move by this amount)? Single-example systems must guess in this situation. Multiple-example systems can usually distinguish after two examples.

Inferring constant parameters is hardly the most challenging issue. For instance, the position to which a board game piece should be moved in Figure 16.2 might be described as "take the number on the dice and move that number of spaces clockwise around the board from where the piece used to be." Such a parameter can be inferred by Gamut but not by other systems. The following sections discuss various techniques that can be used for generalizing nonconstant parameters.

16.3.3 Linear Proportions

Pavlov takes advantage of the observation that many interface behaviors have response parameters that are proportional to the corresponding stimulus parameters. The Celsius-Fahrenheit converter is one example: when one gauge is transformed, the other one should be transformed 5/9 (9/5) as much. Another examples is a diagram editor in which moving a node should cause the ends of all lines connected to it to be moved by the same amount.

Pavlov uses the heuristic that when a stimulus and response are both transformations (e.g., Moves), each response parameter is inferred to be proportional to the corresponding stimulus parameter. As with object descriptors, Pavlov also watches for special cases. For instance, if the designer demonstrated a Move(20,0) as a stimulus, and a Move(0,40) as a response, it maps the nonzero second parameter of the response to the nonzero first parameter of the stimulus.

InferenceBear (Frank and Foley 1994) infers other linear combinations of parameter values from multiple examples of the desired behavior. This feature eliminates the need for hard-wired special cases but requires the user to choose the examples carefully. Furthermore, InferenceBear can take into account properties of the object that do not participate in the action (Pavlov only analyzes the demonstrated stimulus and response objects). For instance, in the shooting arcade game, Inference Bear can infer after multiple examples that the parameters of the bullet creation depends on the location of the gun, though the gun is not part of the stimulus or response.

16.3.4 Complex Parameters

Gamut adds even more inferencing power by using decision trees and other AI algorithms and by allowing the designer to provide hints to help the system generalize the parameters. For instance, after the first example of a bullet creation was provided, Gamut would ask the designer to point out objects that are important. The designer would select the gun, and Gamut would use this to infer the correct create response parameters (i.e., that the bullet should be created directly above the gun). In this way, hints can allow the system to "guess" the correct behavior more quickly than in systems that don't ask the designer for extra help.

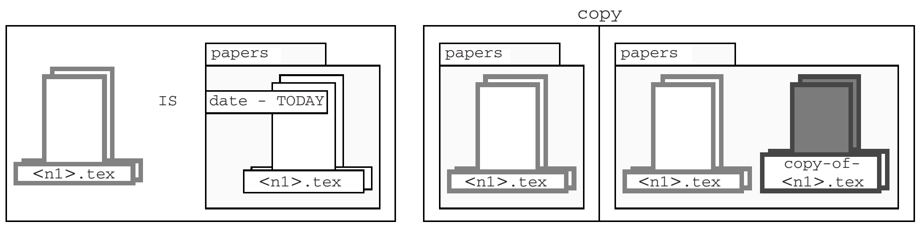
Gamut can also compute the parameters using values from many objects. Examples include setting the color of an object based on the color of a palette or computing the position of the board game piece using the dice as mentioned earlier. As a result, Gamut can be used to create complete applications such as a Turing machine, tic-tac-toe, or a PacMan game.

16.4 Feedback and Editing

When a PBD system infers a program from a designer's demonstration, it doesn't always infer the program that the designer intended. One solution is to allow the designer to demonstrate more examples until, hopefully, the system infers the intended program. But with such a scheme, the designer can never be sure what program the system has inferred and can feel lost not knowing what to do next. Another approach is to ask the designer questions that will hopefully disambiguate the examples. However, our experience with such systems suggest that users have great difficulty answering such questions and generally choose "yes" if given a choice, assuming the computer that is right, even when it isn't.

It is clear from our experience that the designer really needs to see some representation of the inferred program and be able to edit it. Here lies the dilemma, called the *PBD representation problem:* since the goal of PBD is to allow people that aren't programmers to create programs, the system probably shouldn't represent a program as conventional computer code (C++,

FIGURE **16.5**



A Pursuit storyboard showing a file manipulation program created by demonstration in Pursuit.

Java, etc.). Even special-purpose scripting languages, such as those generated by DEMO and InferenceBear, are difficult for end users.

16.4.1 Storyboards

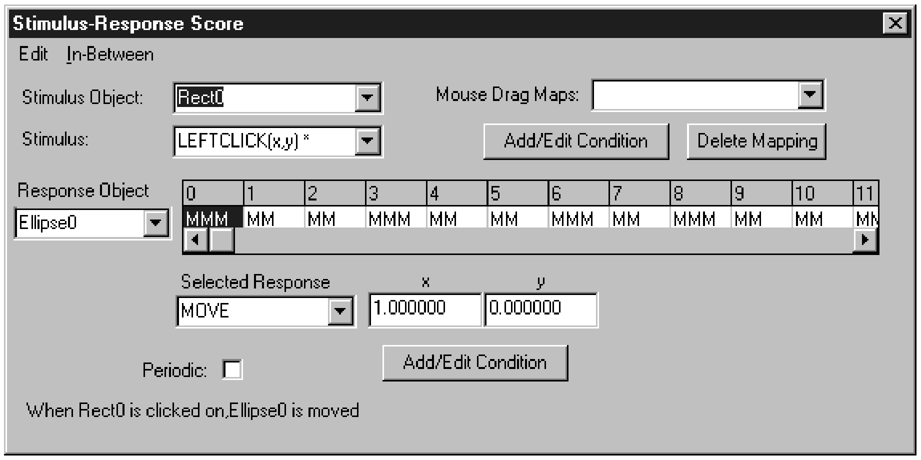
One solution is to present a graphical, storyboard representation of the inferred programs (Kurlander and Feiner 1991; Leiberman 1993; Modugno, Corbett, and Myers 1997). Figure 16.5 shows one from Pursuit (Modugno et al. 1997) that represents copying all the files edited today in the papers folder that end in ".tex".

Storyboards allow a designer to see what the system has recorded and what generalizations it has made. These "programs" can also be edited by moving tiles around, deleting them, or selecting them and redemonstrating the particular operation.

16.4.2 The Stimulus-Response Score

Unlike Pursuit, stimulus-response systems must provide a representation that includes the triggering stimulus. In Pavlov, a dialogue is provided that allows the designer to select an object and a current stimulus (Fig. 16.6). When the stimulus is changed, either with the dialogue or by a stimulus demonstration, the corresponding time line (middle of Fig. 16.6) shows only the responses corresponding to that stimulus. The designer can edit the parameters of each response (*x* and *y* in Figure 16.6) and can click "Add/Edit Condition" to view and edit a graphical condition on the stimulus.

FIGURE 16.6


Stimulus-Response Score

Edit In-Between

Stimulus Object: Rect0

Stimulus: LEFTCLICK(x,y) *

Mouse Drag Maps:

Add/Edit Condition | Delete Mapping

Response Object: Ellipse0

0	1	2	3	4	5	6	7	8	9	10	11
MMM	MM	MM	MMM	MM	MM	MMM	MM	MMM	MM	MM	MM

Selected Response: MOVE

x: 1.000000

y: 0.000000

Periodic: ☐

Add/Edit Condition

When Rect0 is clicked on,Ellipse0 is moved

The Pavlov Editor showing an animation path triggered by a button click. In this example, the current stimulus is a LEFTCLICK on Rect0. The displayed response is an animation path (a series of moves) involving Ellipse0.

Because there is a time line for each stimulus, and because operations, not object states, are recorded, the designer can specify interfaces in which objects behave asynchronously. Chapter 17 of this book provides more details.

It should be noted that Pavlov's stimulus-response score not only is used to view and edit behaviors after the fact but is integrated into the demonstration environment. For instance, the designer can choose a current stimulus in the score, instead of demonstrating it, if she knows its name and doesn't need to specify parameters for it graphically. As mentioned earlier, she can also use the time line in the score to specify time stamps prior to demonstrating responses.

16.5 Conclusion

Many people use software, but few can create it. One of the only areas of creation open to most people is to build static web pages with one type of stimulus: the button click on a link. Stimulus-response systems can open up creativity by allowing people to build applications that come alive, both for the Web and for the desktop.

Our research has introduced some of the alternatives of stimulus-response system design, in terms of syntax, semantics, and feedback. The next step, we believe, is detailed empirical testing between alternatives, either through research or the widespread use that would occur if stimulus-response technology were added to a popular interface builder.

References

Cypher, A. 1991. Eager: Programming repetitive tasks by example. In *Proceedings of CHI '91,* (New Orleans, May).

Cypher, A.,Halbert,D., Kurlander,D., Lieberman, H., Maulsby, D., Myers, B. Turransky, A., eds. *Watch what I do: Programming by demonstration.* Cambridge, Mass.: MIT Press.

Halbert, D. 1984. Programming by example. Ph.D. diss., University of California, Berkeley.

Frank, M. R., and J. D. Foley. 1994. A pure reasoning engine for programming by demonstration. In *Proceedings UIST'94: ACM SIGGRAPH symposium on user interface software and technology.* (Marina del Rey, Calif.).

Fisher, G. L, D. E. Busse, and D. Wolber. 1992. Adding rule-based reasoning to a demonstrational interface builder. In *Proceeding of UIST'92.*

Kosbie, D. S., and B. A. Myers. 1993. PBD invocation techniques: A review and proposal. In *Watch what I do: Programming by demonstration,* ed. A. Cypher. Cambridge, Mass.: MIT Press.

Kurlander, D., and S. Feiner. 1991. Inferring constraints from multiple snapshots. *ACM Transcations on Graphics* (May).

Lieberman, H. 1993. Mondrian: A teachable graphical editor." In *Watch what I do: Programming by demonstration,* ed. Allen Cypher. Cambridge, Mass.: MIT Press.

McDaniel, R., and B. Myers. 1999. Getting more out of programming-by-demonstration. In *Proceedings CHI'99: Human Factors in Computing Systems.* (Pittsburgh, Pa. May 15–20).

Modugno, F., A. T. Corbett, and B. 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.

Myers, B. 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.

Color profile: Generic CMYK printer profile
Composite Default screen

Myers, B., R. McDaniel, and D. Kosbie. 1993. Marquise: Creating complete user interfaces by demonstration. In *Proceedings of INTERCHI '93* (Amsterdam, April).

Smith, D. C., and A. Cypher. 1995. KidSim: End-user programming of simulations. In *Proceedings of CHI '95* (May).

Wolber, David. 1997. "An interface builder for designing animated interfaces. *Transactions on Computer-Human Interface* (TOCHI) (December).

———. 1998. A multiple timeline editor for designing multi-threaded applications. In *Proceedings of the User Interface and Software Technology (UIST) conference* (San Francisco).

Wolber, D., and Gene Fisher. 1991. A demonstrational technique for developing interfaces with dynamically created objects. In *Proceedings of UIST '91* (November).

S
R
L