Creating User Interfaces Using Programming by Example, Visual Programming, and Constraints

BRAD A. MYERS
Carnegie Mellon University

Peridot is an experimental tool that allows designers to create user interface components without conventional programming. The designer draws pictures of what the interface should look like and then uses the mouse and other input devices to demonstrate how the interface should operate. Peridot generalizes from these example pictures and actions to create parameterized procedures, such as those found in conventional user interface libraries such as the Macintosh Toolbox. Peridot uses visual programming, programming by example, constraints, and plausible inferencing to allow nonprogrammers to create menus, buttons, scroll bars, and many other interaction techniques easily and quickly. Peridot created its own interface and can create almost all of the interaction techniques in the Macintosh Toolbox. Therefore, Peridot demonstrates that it is possible to provide sophisticated programming capabilities to nonprogrammers in an easy-to-use manner and still have sufficient power to generate interesting and useful programs.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming; D.2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*; H.1.2 [**Models and Principles**]: User/Machine Systems—*human factors*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*; I.3.6 [**Computer Graphics**]: Methodology and Techniques

General Term: Human Factors

Additional Key Words and Phrases: Constraints, direct manipulation, plausible inference, programming by example, user interface design, user interface management systems, visual programming

1. INTRODUCTION

Peridot is a new, experimental tool for creating graphic, highly interactive user interfaces. One of the primary goals of Peridot is to allow these interfaces to be created by nonprogrammers. To achieve this, Peridot uses the techniques of visual programming, programming by example, constraints, and plausible inferencing. Peridot was developed as part of the author's Ph.D. dissertation. Previous papers have presented an overview of Peridot [31] and a detailed discussion of the handling of the mouse [26]. A full report on Peridot is also available [28].

This work was performed while the author was at the University of Toronto. This research was partially funded by the National Science and Engineering Research Council (NSERC) of Canada.
Author's address: Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213.

This paper focuses on the programming aspects of Peridot and how programming capabilities are provided to nonprogrammers.

Peridot stands for *P*rogramming by *E*xample for *R*eal-time *I*nterface *D*esign *O*bviating *T*yping and is a working prototype implemented in Interlisp-D on the Xerox 1109 DandeTiger workstation. Peridot can create many types of interaction techniques (sometimes called "widgets"), which are the low-level components of user interfaces. This includes most kinds of menus, property sheets, light buttons, radio buttons, scroll bars, two-dimensional scroll boxes, percent-done progress indicators, graphic potentiometers, sliders, iconic and title line controls for windows, and many others. Thus, Peridot can create almost all of the Apple Macintosh [53] interface, as well as many new interfaces, such as those that use multiple input devices concurrently. Peridot also has created its own user interface.

Two of the most important components of the design for highly interactive user interfaces are the graphic presentation and the way the graphics change due to the mouse and other input devices. Unfortunately, previous tools for creating user interfaces have not adequately addressed these components. In order to allow the graphic parts of the user interface to be easily specified, Peridot allows the user interface designer to draw a picture of what the user interface should look like using a special drawing package. In order to specify how the user interface should work with the mouse, Peridot allows the designer to move the mouse and toggle its buttons. While this is happening, Peridot creates code that allows the interface to be used with actual application programs. The code produced is not simply a transcript of the designer's actions, however, because this would not provide sufficient functionality. For example, a pop-up menu might be designed with a particular set of strings, but the same code should work for any list of strings. Therefore, Peridot creates *parameterized* procedures, where parts of the interface can depend on the values of the parameters. For example, the size of the menu might be adjusted to fit exactly around the strings. Peridot uses some simple artificial intelligence techniques to infer how the graphics and mouse should change based on the actual values for the parameters.

In order to allow interaction technique procedures to be created in a *direct manipulation* manner [43], Peridot has the designer provide example values for each parameter. For instance, when creating a menu, the designer provides an example list of strings to be displayed in the menu. Using a technique called *programming by example*, Peridot generalizes from the given examples to create a general-purpose procedure. Since programs are created using graphic techniques, Peridot can also be classified as a *visual programming* system. *Constraints*, which are relationships that are maintained automatically by the system, are used in two ways in Peridot. *Graphic constraints* tie various graphic objects together so that when one changes the others will be updated automatically. *Data constraints* are used to ensure that graphic objects are updated whenever any special variables, called *active values*, are updated.

2. RELATED WORK

Because the programming of user interfaces is difficult and expensive, there has been a growing effort to create tools to help with this task. Sometimes, the tools are simply a library of low-level procedures that create the components of the

interface. There may be procedures that display menus, on-screen buttons, scroll bars, etc. In this case, the library is usually called a "user interface toolkit," and the primary example of this is the Macintosh Toolbox [1].

Some user interface tools also help with the creation and management of these low-level components. These higher-level tools are often called *user interface management systems* (*UIMSs*) [29, 36, 40]. Some UIMSs have concentrated on managing the sequencing (or syntax) of the user interface; that is, what operations are available at each point in the interface and what happens after each operation is specified. In SYNGRAPH [37] the designer uses a BNF to specify the grammar of the interaction. Other systems [20, 21, 52] have used state transition diagrams for specifying the sequence of actions. These syntax-based approaches are most useful for creating user interfaces where a large amount of syntactic parsing is necessary or when the user interface has a large number of modes. However, most modern, highly interactive systems attempt to be mostly "mode-free" [50], so these UIMSs have not been successful for them.

Therefore, many UIMSs allow some part of the presentation of the user interface to be specified graphically. For example, Menulay [10] allows the designer to place text, graphic potentiometers, iconic pictures, and light buttons on the screen and to see exactly what the end user will see when the application is run. The interfaces that Menulay can create are essentially a series of mostly static frames, where the parts that can be manipulated by the user are programmed by hand in advance. Thus, Menulay allows the designer, who does not need to be able to program, to lay out the interface on the screen. Similar systems include Trillium [18], vu [44], DialogEditor [11], and the NeXT Interface Builder.

These systems are limited to using graphic techniques for specifying the placement and size of picture and user interface elements (e.g., where menus are located and what type of light button to place where). Some systems, such as Squeak [12], allow the properties of user interface elements themselves to be programmed textually, but Peridot is the only system that allows both the appearance and the dynamic behavior of the elements to be specified in a graphic, nontextual manner. Peridot supports this by incorporating ideas from a number of visual-programming and programming-by-example systems.

As mentioned above, *visual programming* is the use of graphics to create computer programs [25]. *Programming by example* is when the system generalizes from example values provided by the user. Many visual-programming and example-based-programming systems have been designed to provide nonprogrammers with programming capabilities. For example, Rehearsal World [16] uses the metaphor of a stage to allow the user to create teaching programs by connecting together preexisting "actors" (which are represented by pictures).

The visual programming systems that are most relevant to Peridot also use programming by example. The seminal system is Pygmalion [45], which supports programming using icons. SmallStar [17] allows the end user to program a prototype version of the Star office workstation. When programming, the user simply enters program mode, performs the operations that are to be remembered, and then leaves program mode. The operations are executed in the actual user interface of the system, which the user already knows. Since the system does not

use inferencing, the user must differentiate constants from variables and explicitly add control structures (loops and conditionals). This is performed after the demonstration is completed by editing a textual representation of the program.

Peridot extends this idea by using inferencing to avoid the need for the textual program representation; the system guesses the generalizations from the examples. Many previous systems that tried to infer from examples were not successful because the domain was too general. For example, one system [42] tried to generate LISP programs from examples of input/output pairs. Autoprogrammer [3] is typical of a class of programming-by-example systems that attempt to infer general programs using examples of traces of the program execution. The user gives all the steps, and the programs try to determine where loops and conditionals should go as well as which example values should be constants and which should be variables. Programming by example has been more successful in limited domains, such as generating editor macros [35] and simple transformations on pictures [24].

Another important component of Peridot is the use of *constraints* [22], which are relationships among objects and data that must hold even when the objects are manipulated. Peridot uses two kinds of constraints. *Graphic constraints* relate one graphic object to another, and *data constraints* ensure that a graphic object has a particular relationship to a data value.

The idea of graphic constraints was first used in Sketchpad [48]. Thinglab [4, 5] and its descendants [6] extended the ideas in Sketchpad to provide generalized programming with constraints. Juno [34] uses constraints in a drawing program. Constraints in Peridot and in some other systems, including Apogee [19] and GROW [2], are one directional. This means that for any property there can be at most one formula that calculates its value. Other systems, including Sketchpad, Thinglab, Juno, CONSTRAINT [51], and Magritte [15], use multiway constraints, which means that there can be multiple formulas for a property or that constraint expressions can be solved for any of the objects referenced. Multiway constraints usually require a complex constraint satisfaction algorithm that often cannot operate quickly enough for highly interactive user interfaces [7], whereas one-directional constraints, as in Peridot, use a much simpler, real-time algorithm (described in Section 6).

Data constraints ensure that a graphic object has a particular relationship to a data value. An example of a data constraint is that the diameter of a circle is the same as the value of a global variable, so the size of the circle will change if the variable is set. These are used in the Process Visualization System [14], which was influenced by "triggers" and "alerters" in database management systems [8]. They are also similar to the "Control" values in GRINS [38], except that they are programmed by example rather than textually and can be executed immediately without waiting for compilation.

The data constraints in Peridot can only be connected to *active values*. Active values are like parameters to a procedure, except that when their value changes the graphics that depend on them are automatically redisplayed based on the new value. Active values have been used in artificial intelligence systems and environments, such as Steamer [47], KEE [41], and LOOPS [46].

3. OVERVIEW OF PERIDOT

When creating an interface with Peridot, the designer must first name the procedure that will be called by the application to cause the interface to appear. This procedure can have parameters, which are used for aspects of the interface that change from call to call, but do not change while the interface is running, such as the strings in a menu. Constraints will ensure that the interface appropriately reflects whatever values for the parameters are supplied. Procedures also have *active values* associated with them for the parts that do change while running, such as the particular string that the mouse is over. A data constraint will ensure that the feedback graphics, such as a reverse video rectangle, shows the item that the active value refers to.

Figure 1 shows a typical configuration when running Peridot. The window shows the parameters to the procedure and an example value for each parameter. Below the parameters are the active values used by the procedure. The menu on the left is used for giving all of the Peridot commands. This menu is divided into two sections with the most common commands on the top. The center window shows the resulting user interface. The bottom window is used for printing error messages and for prompting the designer. The window on the right shows the code that Peridot is creating for this procedure. This window is mainly for debugging, since it is not necessary for the designer to view the code or to use the code to perform any operations in Peridot. The normal window configuration, in fact, does not even display this window, as shown in Figure 2b.

While the designer is creating a user interface, Peridot is continuously trying to guess how the various pieces are related to each other and how to generalize from the examples. Because any inferencing system will occasionally guess incorrectly, Peridot uses three strategies to protect against incorrect inferences. First, Peridot always asks the designer if guesses are correct by printing a message in the prompt window; second, the results of the inferences can be immediately seen and executed; and third, the inferences can be undone if they are in error.

4. EXAMPLE OF PERIDOT IN ACTION

The best way to demonstrate how easy it is to create a user interface with Peridot is to work through an example. Here, a property sheet will be created. Other examples are available in other papers [26, 28, 31], but it is easiest to appreciate how easy Peridot is to use by seeing a videotape [27] or live demonstration.

To start Peridot the designer presses in the Peridot logo window, and the Peridot prompt window is displayed (Figure 2a). In this window the designer types the name of the procedure, the names for all parameters and active values for this procedure, and an example value for each. Peridot then displays the menu and windows shown in Figure 2b and allows the designer to begin drawing the picture. In Figure 2b, the designer has created a gray rectangle to represent a "drop shadow" for a button. In Figure 2c the designer has drawn a black rectangle to represent the background of the button, and Peridot has noticed that this rectangle seems to be the same size as the gray rectangle, offset by a constant nine pixels. In the prompt area, it is asking the designer to confirm this constraint. The designer types y for "yes," and Peridot immediately adjusts the

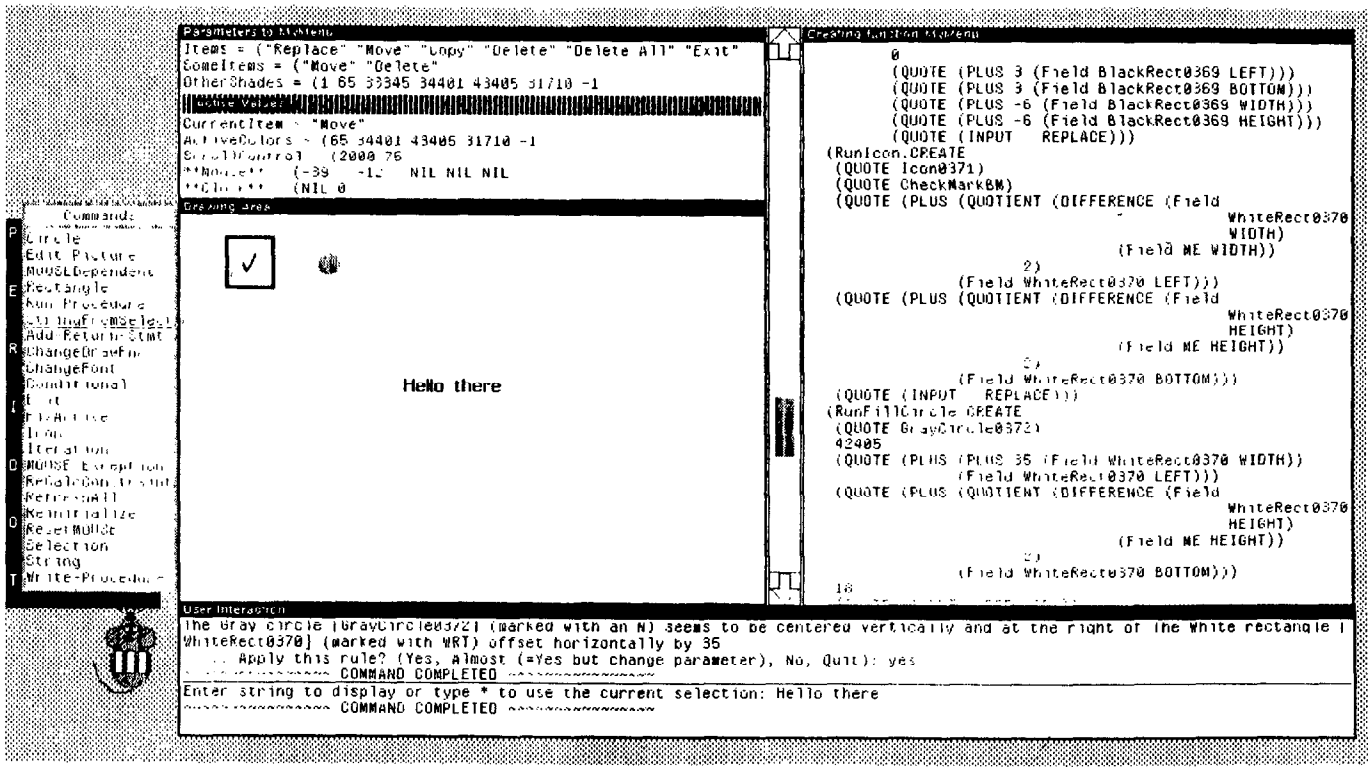
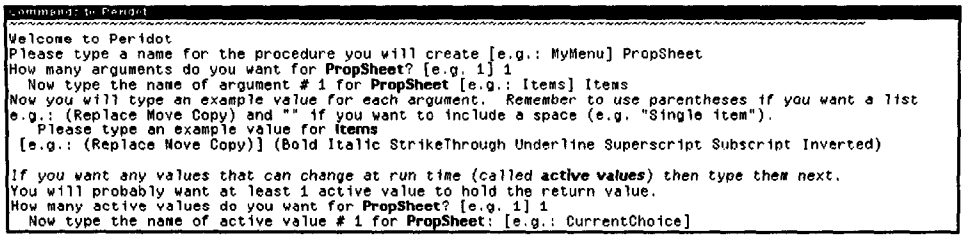
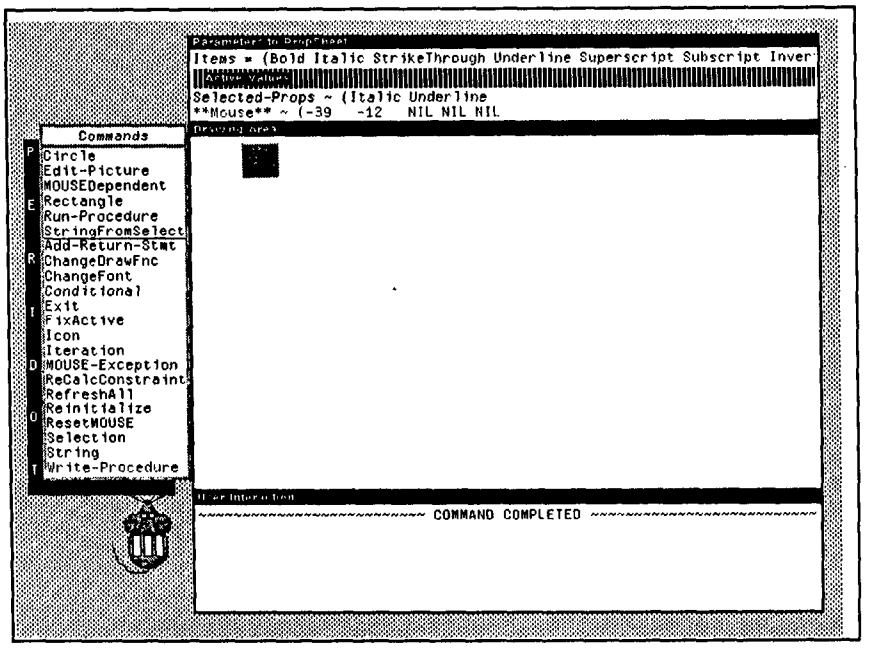


Fig. 1. The windows that are used when creating a procedure using Peridot. The top two windows display the parameters and active values, and example values for each. The large center window displays the user interface being created. The lower window displays the prompts and error messages from Peridot. The code window on the right is optional and is mainly used for debugging Peridot itself. The designer gives all commands using the Peridot command menu on the left. The top portion of the menu contains the most common commands. The menu and scroll bar were created using Peridot.



(a)

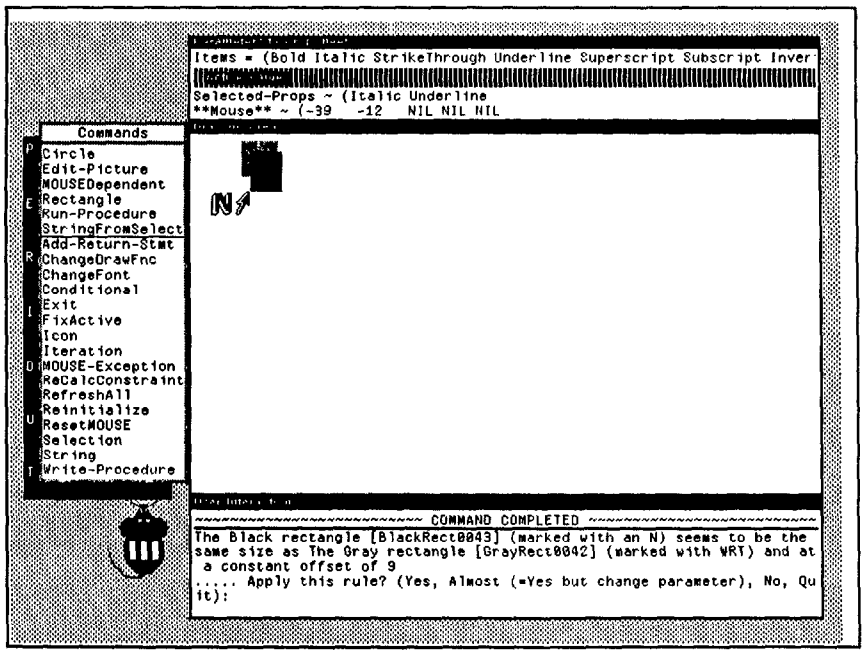


(b)

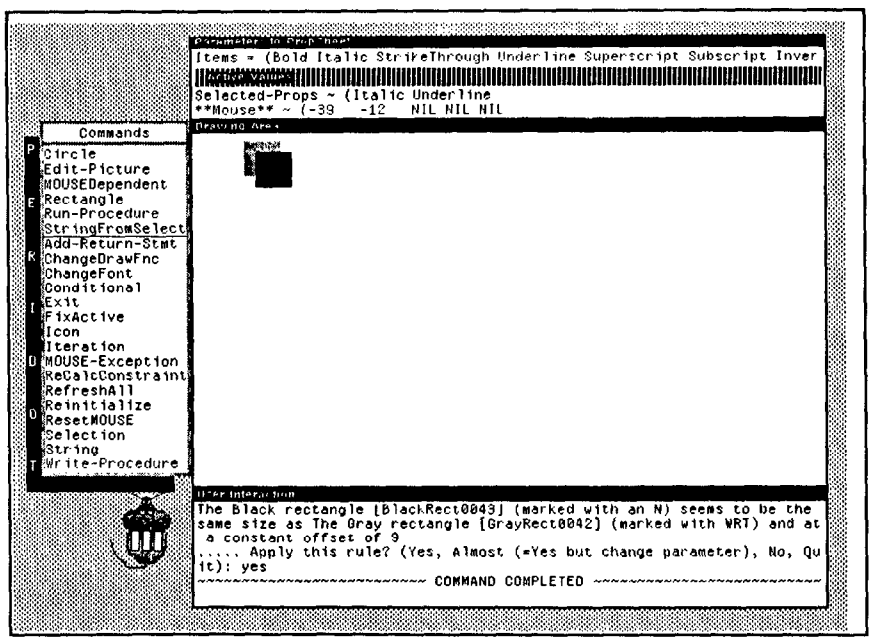
Fig. 2. A sequence of snapshots during the creation of check boxes using Peridot.

black rectangle to be exactly the same size as the gray one (Figure 2d). If the gray rectangle's size were now changed, the black rectangle's size would change also, since a graphic constraint has been established that keeps both rectangles the same size.

Next, the designer draws a white rectangle inside the black one (Figure 2e), and Peridot correctly infers that this rectangle should be evenly nested inside the black one. In Figure 2f the designer has selected the first element of the parameter "Items," which is the string "Bold," and has used that as the string to display. Peridot infers that it is centered to the right of the white rectangle. The code that is produced for this string refers to the first element of the first parameter, whatever that is, rather than to the constant string "Bold," so that any value used for the parameter will be displayed.

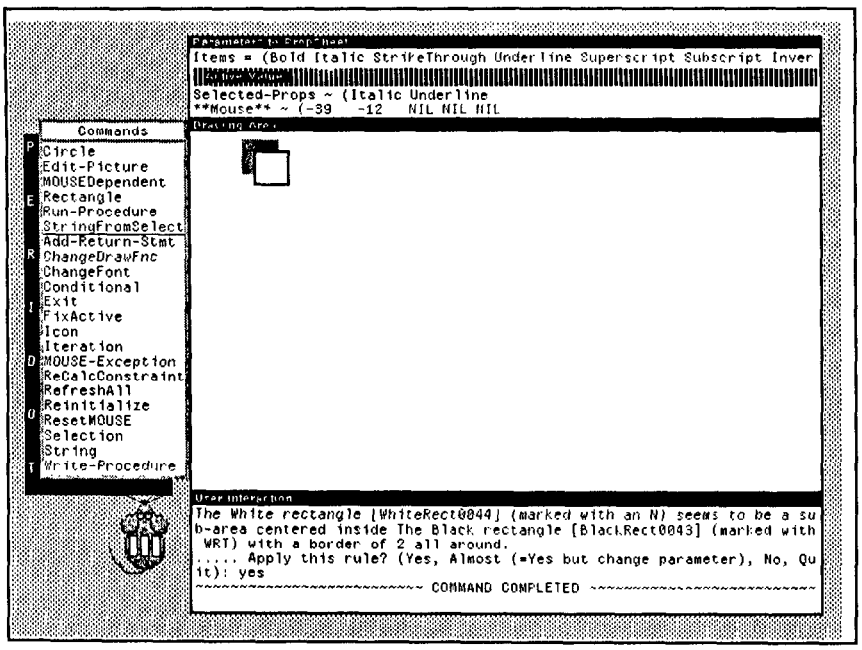


(c)

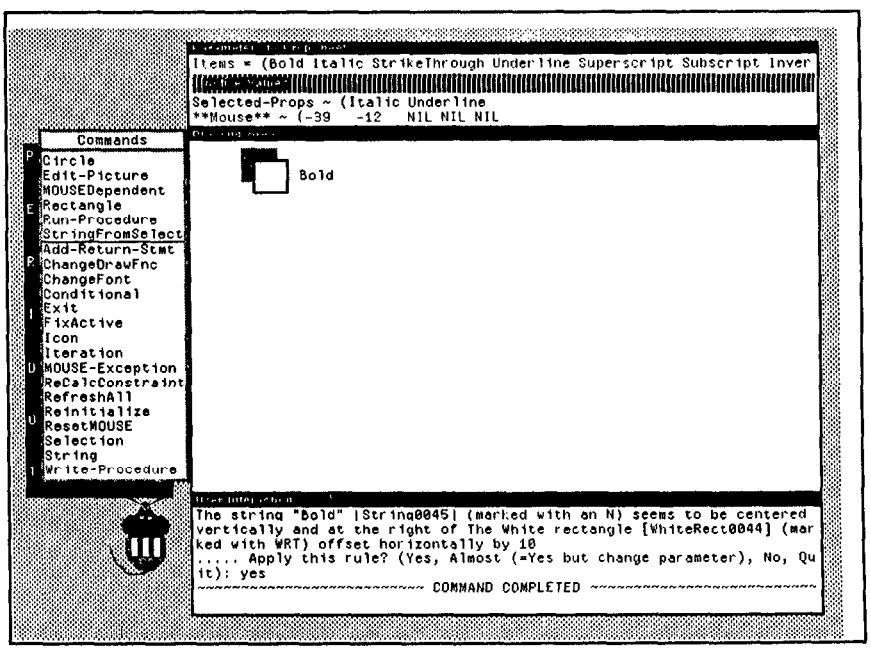


(d)

Fig. 2. (*Continued*).

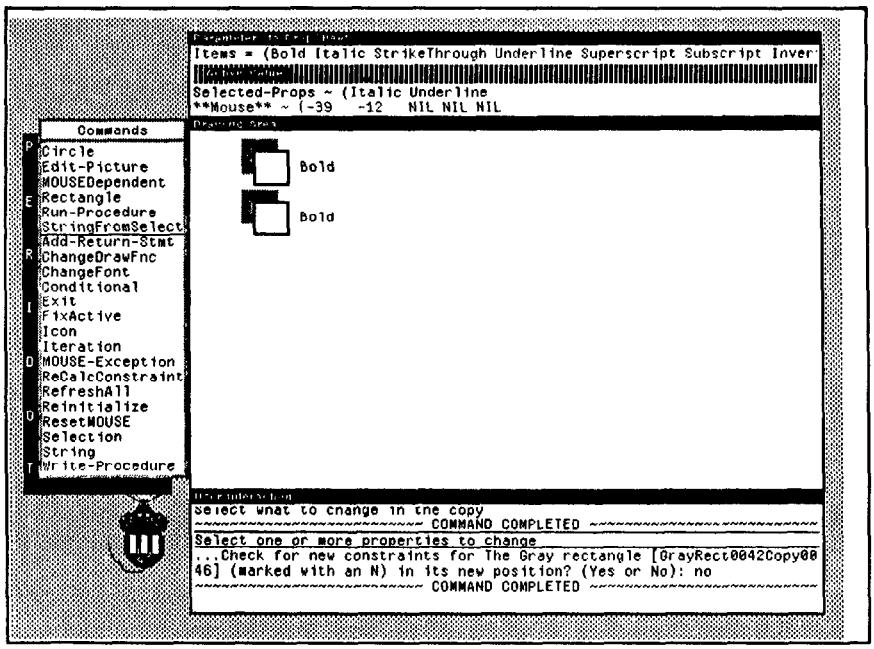


(e)

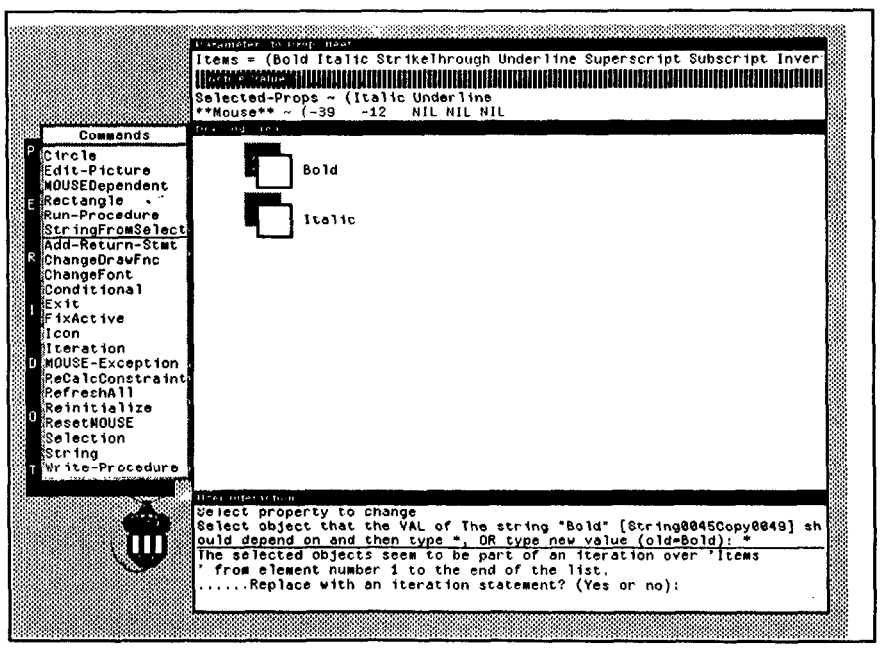


(f)

Fig. 2. (*Continued*).

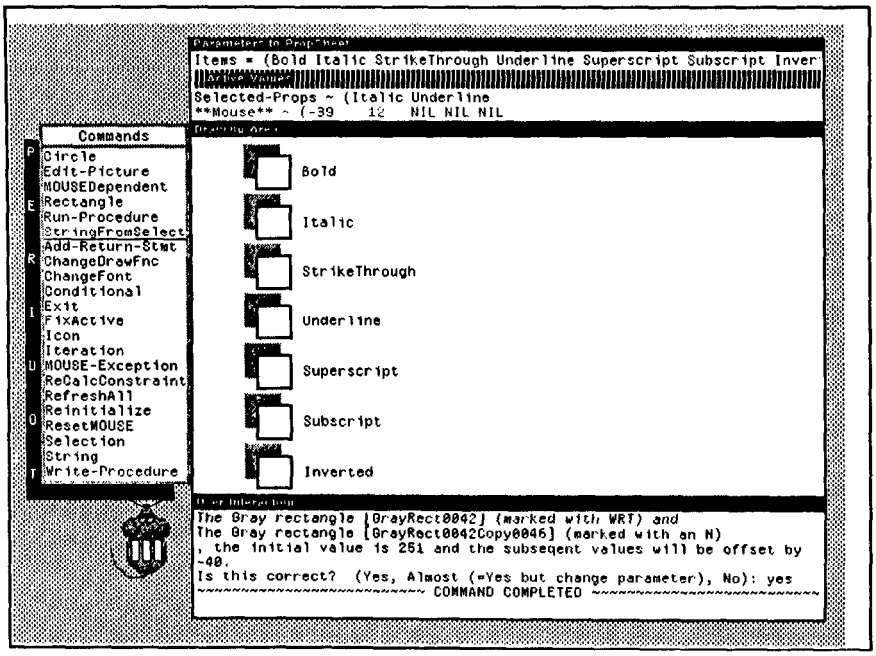


(g)

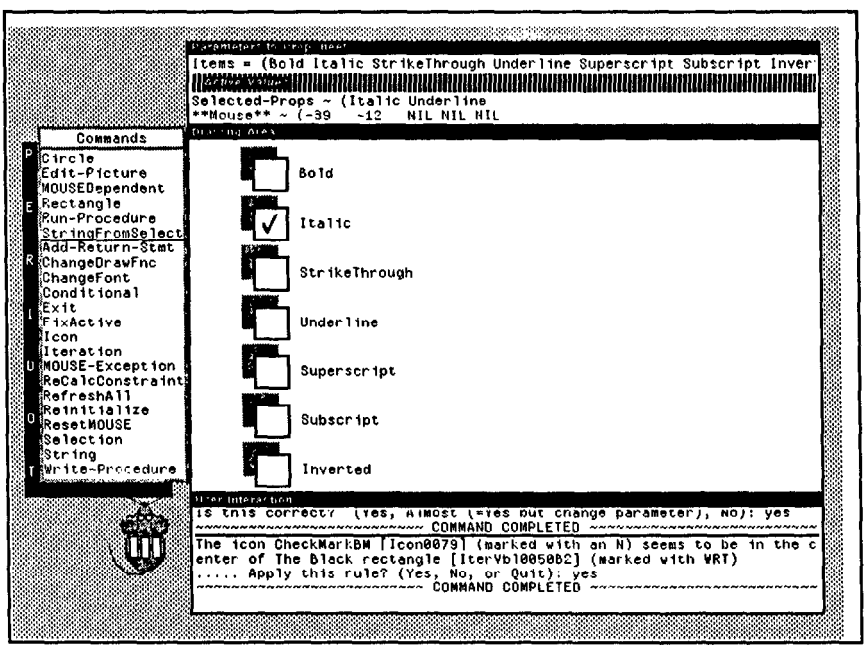


(h)

Fig. 2. (*Continued*).

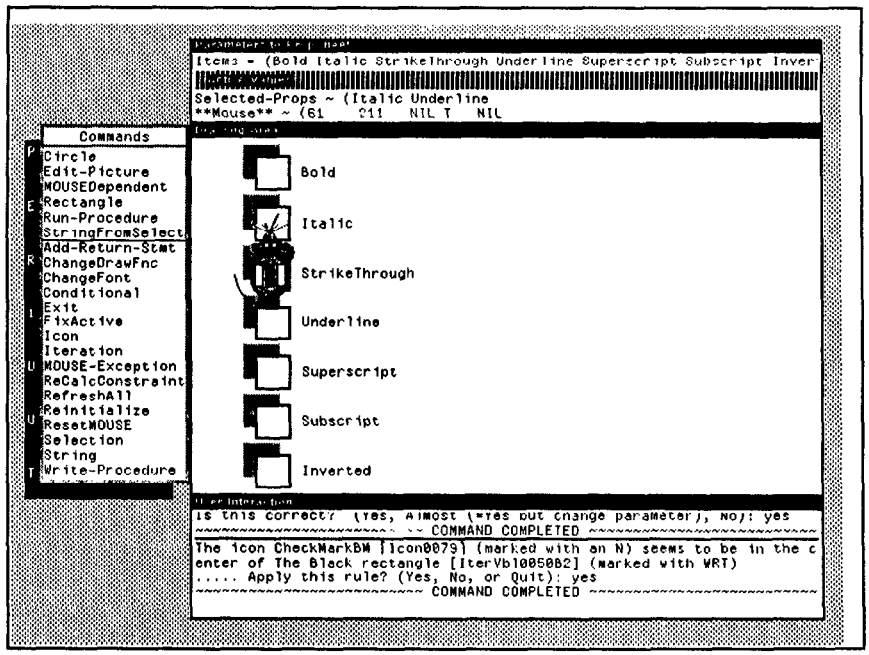


(i)



(j)

Fig. 2. (*Continued*).

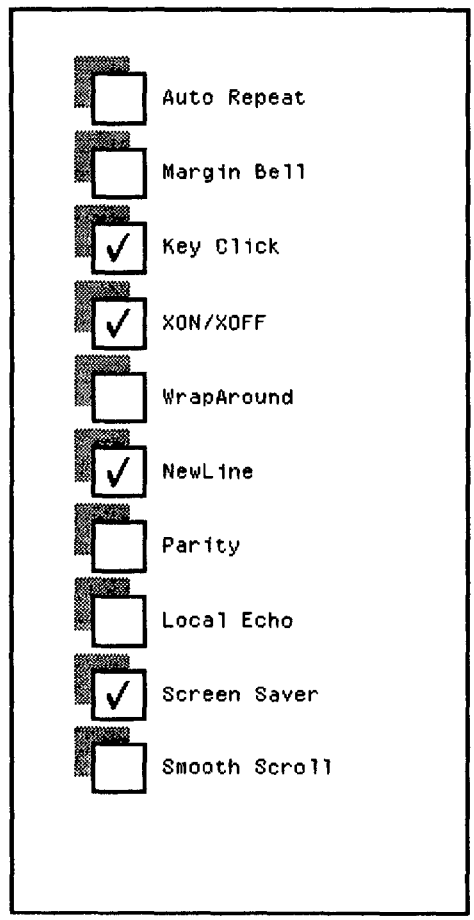


(k)

Fig. 2. (*Continued*).

Next, the designer selects all the objects created so far and specifies that they should be copied to a new position (Figure 2g). Peridot asks if it should look for constraints from the new copy to the old one, but this is not necessary since it is going to be part of an iteration. Next, the designer edits the second string to refer to the second element of the parameter (Figure 2h). At this point, Peridot notices that the designer has used the first two elements of a list in the interface, and asks whether the rest of the elements of the list should be displayed in the same way, as part of an *iteration* over all the elements of the list. The designer confirms this, and the rest are immediately shown (Figure 2i). In order to perform this conversion, Peridot has to determine which graphic objects should participate in the loop and how they should change in each cycle. Now the presentation aspects of the property sheet are finished.

Next, the designer places the iconic picture of a check mark centered inside one of the boxes (Figure 2j). This is used to show which items are selected. In order to demonstrate that this should be selectable by the mouse, the "simulated mouse" icon is used (see Figure 2k). The real mouse cannot be used, since it is used for giving Peridot commands [31]. The nose of the simulated mouse is placed over the check mark with the middle button down, and the MOUSEDependent command is given. Since there is only one active value (Selected-Props), Peridot guesses that the check mark should depend on this active value. Since the example value of that active value is a list, Peridot guesses



(l)

Fig. 2. *(Continued).*

that multiple items are allowed and that a check mark should appear for each one in the list. The designer is asked to confirm these guesses in the prompt window. Peridot then shows the check marks displayed in the boxes next to Italic and Underline, since these are the current value of Selected-Props. Finally, the designer is asked whether pressing the middle button should toggle, set, or clear the selected object, and the designer types t for toggle. The user interface is now complete, and either it can be tested with the simulated mouse, or else Peridot can be put into "Run Mode" and the real mouse can be used.

The PropSheet procedure that has been created can now be used outside of Peridot as part of application programs. It is parameterized as to the list of items that are displayed, so it can be called with an entirely different list of strings, even if that list has a different number of elements (Figure 2l).

5. LANGUAGE FEATURES

Because Peridot creates user interface procedures, it operates as a code generator. This section discusses some of the features of the language it can generate. Sections 6–9 discuss how constraints, programming by example, and visual programming are used to allow nonprogrammers to create sophisticated user interface procedures using Peridot.

The code generated by Peridot has a number of conventional parts: straight-line code, iterations, conditionals, and parameterized procedures.

5.1 Straight-Line Code

As the user is drawing objects, Peridot creates LISP code that will draw them for application programs. If the user edits an object, the code that generates it is modified. If properties of objects are fixed and unchanging, then their values will be constants. If the properties are to change at run time based on parameters to the procedure or end-user input, they are controlled by constraints. If the objects themselves appear and disappear at run time, they must be enclosed in conditionals or iterations.

A window can be displayed to see the code as it is generated (Figure 1). Unlike other systems, such as Juno [34], users are not allowed to edit this text code, since they are assumed not to be programmers, but it would not be difficult to provide this, if desired.

5.2 Iterations

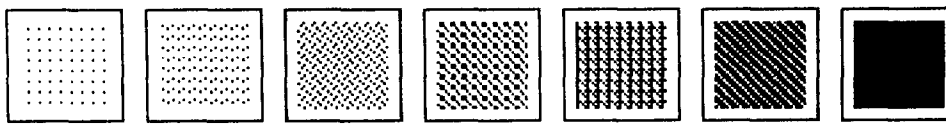
Iterations are important because they allow Peridot to support variable-length lists and they relieve the designer from having to perform tedious, repetitive actions. As shown in the example, Peridot infers iterations when two items from a list have been used.

There are two forms of iterations in Peridot. The most common form displays a copy of one or more graphic objects for each item of a list, as in the example of Figure 2i. If more sophisticated control is needed over the order of the items or over which ones should be displayed, a programmer would have to write a LISP procedure to filter the list and supply Peridot with a list of only the appropriate items in the correct order.

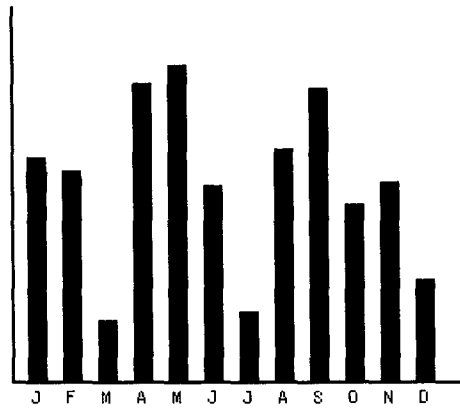
The items in the list can be used to control any property of the graphic objects in the iteration, including the text of the strings (Figure 2i), the halftone patterns for rectangles (Figure 3a), the heights for rectangles (Figure 3b), etc.

The other form for iterations is to display a set of objects for a specific number of times. Of course, some properties of the objects may change in each cycle (usually including the position). This is mainly useful for displaying a line of identical objects (see Figure 4). To get this form of iteration, the designer creates two copies of the objects to be repeated, selects them, and then executes the Peridot Iteration command.

Currently, iterations in Peridot can only be used when the items are in a single line (which can be horizontal, vertical, or even an arbitrary diagonal). It would be useful to extend Peridot to allow multiple lines of items, either as automatic wraparound or as nested iterations. If the data were presented as a single list,



(a)



(b)

Fig. 3. Iterations can be on any property of an object, for example, the halftone shades (a) or the heights for rectangles (b).

Fig. 4. Iteration for an integer number of times.

the designer would have to specify when to go to the next row or column, for example, by the maximum number of items on a line or by the maximum size. If the data were presented as a two-dimensional list of lists, Peridot could infer the appropriate layout from three example items (a reference plus one item in each dimension). Another useful extension might be to allow an iteration to go through multiple lists in parallel. For example, the heights of the bars in a bar chart might depend on values in one list, and the colors of the bars depend on values in another list. These extensions would require a small addition to Peridot's implementation, but its user interface would not have to change.

5.3 Conditionals

Conditionals in Peridot are used to support displayed feedback over one of a set of objects and to control an object blinking on and off. These "conditionals" are different from IF statements in conventional languages because they do not affect the flow of control. They are called "conditionals" here because they cause the display of an object to be conditional on a controlling variable.

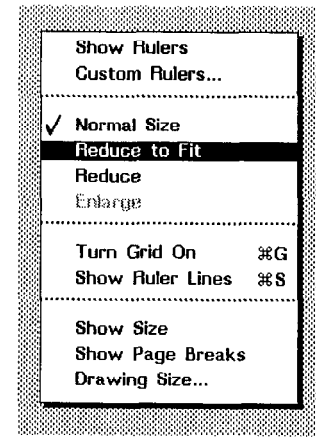


Fig. 5. A menu created using Peridot. The horizontal dotted lines and the gray items are exceptions to the normal way the text is displayed. The feedback black rectangle will not appear over the gray item "Enlarge" or over the dotted lines.

Like iterations, conditionals can only test the value of an active value or parameter to the procedure. Designers who have used Peridot have found this to be intuitive, and it can support most of the desired functionality.

Conditionals are created in a *postfix style*; that is, the designer first draws the graphic objects that are used as feedback when the conditional is true and then specifies what these objects should depend on. This allows the designer to use the standard drawing and editing commands to create the graphic objects. In Figure 2l the conditional objects are the check marks; and in Figure 5, the black rectangle, the check mark, the graying of "Enlarge," and the dotted lines are all implemented with conditionals.

The general form for conditionals is to display the graphic objects if the controlling variable has an appropriate value. If the graphic objects just blink on and off, then the controlling value can be T or NIL. The more common case is for the graphics to be displayed over one or more objects in an iteration. Here, the controlling value chooses which items of the iteration the value should be displayed over, either by index (e.g., the third item) or by name (the item containing the string "Reduce to Fit"). The variable can also be either a single value, where only one object can be selected, or a list of values, to allow multiple selection. The example value supplied by the designer is used by Peridot to guess which of these kinds of control is desired. Of course, the designer is queried to confirm the guess, since they can be ambiguous [28].

An interesting idea is to try to extend conditionals to handle many other kinds of optional or changing behavior. For example, conditionals might support parameters that control the way that the interface works and looks. A parameter might determine whether the items in a menu are left justified or centered, and another parameter might determine whether the left or middle mouse button is used in the interaction. In these cases, both possibilities would be demonstrated, and Peridot would prompt for the parameter or active value that would determine which one to use. This would allow Peridot's conditionals to be more like a general if-then-else programming construct. The Tinker system [23] has a similar facility, although there the code for each branch of the conditional must be entered textually rather than demonstrated.

Another extension would be to use conditionals to handle exceptions to an iteration. For example, in Figure 5, to attain the dotted lines requires an extra parameter that tells Peridot where to put them. A better scheme would be for the lines to appear whenever the iteration variable takes a special value (maybe "_" in this case). These could easily be inferred by example, since the cases that Peridot might notice are easily identified (such as that the iteration element has a particular value, that the value has a different type than the other elements, or a numerical property such as that the item is greater than zero). The exception mechanism might even be used to handle the automatic line breaks for the multiline iterations discussed in Section 5.2. Providing these extensions would not significantly change Peridot's user interface.

5.4 Parameters and Return Values

An important property of the code that Peridot generates is that the procedures are parameterized. In fact, they look like normal procedures that one might find in a conventional toolkit. As shown in Figure 2a, the designer specifies the names of the parameters before the interface is built. Since the procedures are implemented in LISP, the parameters are not typed.

This provision for parameters is the most significant difference between Peridot and other graphic user interface tools. Other systems, including NeXT's Interface Builder, vu [44], and DialogEditor [11], only allow the designer to specify a fixed set of values for the menus and buttons.

To specify a return value for the procedure, the designer must use the Add-Return-Stmt command. This requires that an active value be selected and allows the procedure to be exited based on the setting of this active value. A menu pops up that asks the designer whether the procedure should exit when the active value is set, when it is set with a value different from its old value (i.e., when it changes value), when it is set with a particular value, or when an interaction that uses this active value is complete (e.g., when the mouse button is released) [26]. Although this does not seem like many choices, it is sufficient to support most interaction techniques, including stopping when the mouse presses on a particular area (such as the STOP icon in Figure 6), and to make pop-up menus that go away when an item is selected or when the user releases outside the menu.

6. GRAPHIC CONSTRAINTS

One important reason that Peridot is more complicated than a conventional drawing package is that it must deal with the parameterization of the procedures. This implies that Peridot must know how various graphic parts of the interface change with different values for the actual parameters. For example, the pop-up menu of Figure 7a was defined with one set of strings, but must also work for a different set as shown in Figure 7b. Here, Peridot must know that the size of the shadow and outline rectangles must change based on the width of the widest string and the sum of the heights of all the strings.

As shown in Figures 2c and d, these graphic relationships are normally inferred automatically as the user interface is drawn. Section 8.2.1 discusses the inferencing aspects of Peridot. It is also possible to specify explicitly the relationships by selecting two objects and providing an arbitrary arithmetic expression that relates

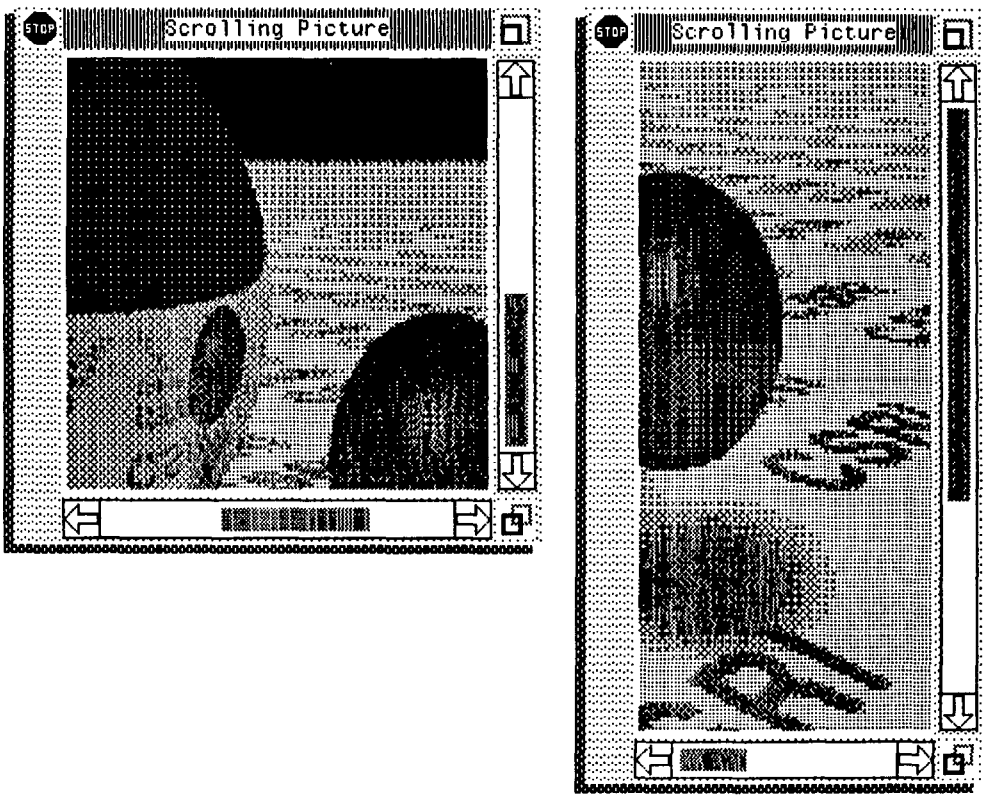
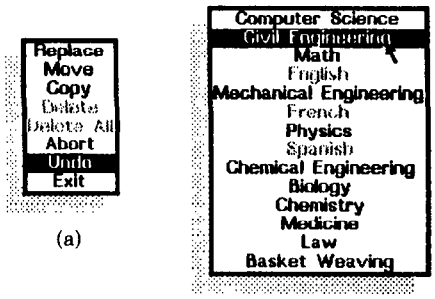


Fig. 6. Two views of an interface created entirely by Peridot. The various parts (icons, scroll bars, title string, decorations) are all constrained to the shadow rectangle, so they stay the appropriate size and position when the window changes size. The picture inside the window can be scrolled horizontally or vertically either by pressing and moving the indicator boxes in the scroll bars or by pressing on the arrows. The window's size can be changed by pressing on the icon at the upper right, and the position can be changed by pressing on the icon at the lower right. The window is destroyed and the user interface procedure exits when the user presses on the icon at the upper left. An application program is called to display the picture inside the window whenever any of the scroll areas are used or the window changes size, but all of the interactions are handled entirely by Peridot and were created by demonstration without programming.

their properties. To make it easier on the designer, a special command is provided to make some property (e.g., the width) of two objects be offset by a constant amount. The designer simply selects the two objects, and Peridot asks whether the current offset between the two objects should be used. If not, the designer can type in a new value.

After a relationship is either inferred or explicitly specified, Peridot creates a graphic constraint so that the relationship will be maintained if the picture is edited or if different parameters are used at run time. The constraints used in Peridot differ markedly from constraints in previous systems because they are simple and efficiently implemented. The primary reason for this is that only one-directional constraints are necessary. The reverse relationship is saved at design



(a) (b)

Fig. 7. Peridot needs to know that the sizes of the white, black, and gray rectangles depend on the actual strings to be displayed. The list of strings is passed as a parameter to the menu display procedure.

(a) (b)

Fig. 8. Originally, in (a), the gray rectangle depends on the black rectangle. When the gray rectangle is later made to depend on the string (b), the black rectangle is made to depend on the gray rectangle.

time in case the designer edits the picture. For example, when creating the button shown in Figure 8, the first step is to create the black and then the gray rectangles, as shown in Figure 8a. At this point, the gray rectangle's size and position depend on the size and position of the black rectangle. Next, the designer adds the string, and Peridot infers that the size of the gray rectangle should depend on the size of the string. Since constraints are only one directional, this would remove the constraint that connected the gray and black rectangles. Peridot notices this and asks the designer if the constraint should be reversed. The question is asked because it is often the case that the user wants to remove or change the constraints rather than reverse them, in order to change the way the picture looks. If the designer confirms that the constraints should be reversed, then the previous constraint is removed and the reverse asserted. In Figure 8b, this will cause the size of the black rectangle to depend on the size of the gray rectangle.

The dependencies of an object's attributes are often cascaded. For example, in Figure 7, the LEFT position of the white rectangle depends on the LEFT position of the black rectangle, which, in turn, depends on the LEFT position of the gray rectangle. Peridot is careful to reverse all the necessary constraints so that the interface stays consistent.

In addition, the dependencies may go forward in the drawing order as well as backward. Normally, when an object is drawn, it depends only on objects drawn previously, so the numerical values of their attributes are available when the object is drawn. However, if a relationship has been reversed or the user explicitly edits an attribute to depend on some object, an object may be drawn before the

object it depends on is drawn. For example, the width of the gray rectangle depends on the width of the string in Figure 8, but the rectangle is drawn before the string. The drawing order of objects cannot be changed, however, since newer objects can obscure older objects.[1] Therefore, the calculation order must be different from the drawing order.

Fortunately, it is easy to handle these cases with an efficient, recursive, one-pass algorithm. The "Field" function, which returns attributes of objects, uses the value of the attribute if it has been calculated. If it has not been calculated, then it evaluates the appropriate constraint. This may recursively call Field, but no attribute will ever be evaluated more than once because cycles are not allowed.

The code to create the rectangles and strings in Figure 8 might look like the following:

```
(CreateRectangle ' BlackRect0001
  BLACK
  30                                          ;x
  40                                          ;y
  '(PLUS 10 (Field GrayRect0002 WIDTH))       ;width
  '(PLUS 10 (Field GrayRect0002 HEIGHT)))     ;height
(CreateRectangle ' GrayRect0002
  Gray
  '(PLUS 5 (Field BlackRect0001 LEFT))        ;x
  '(PLUS 5 (Field BlackRect0001 BOTTOM))      ;y
  '(PLUS 6 (Field String0003 WIDTH))          ;width
  '(PLUS 6 (Field String0003 HEIGHT)))        ;height
(CreateString ' String0003
  "Confirm"
  '(PLUS 3 (Field GrayRect0002 LEFT))         ;x
  '(PLUS 3 (Field GrayRect0002 BOTTOM)))      ;y
```

The time complexity of this algorithm is clearly constant per object (i.e., linear in the number of objects).

The one-directional graphic constraints in Peridot have proved to be sufficient for handling all the relationships that occur in user interface elements. Operations that appear to require two-directional constraints are usually handled in Peridot using active values. For example, if there were two graphic sliders that both modified the same value, a system like ThingLab [5] would have two-directional constraints that would keep each slider the size of the other one. In Peridot, however, each slider would depend on a single active value, and when either slider was manipulated with the mouse, the active value would be set, so both sliders would be adjusted. This technique eliminates the need for multidirectional constraints in most cases. This still will not handle the case, however, where the sliders represent different values, like the classical Fahrenheit to Celsius converter. Peridot's constraints cannot be used in this case. Fortunately, this situation does not arise in the low-level user interface elements that Peridot is designed to create, so multidirectional constraints have not proved necessary. Application procedures attached to different active values can be used to implement this kind of high-level relationship, if necessary.

[1] This is often called the "painter's algorithm."

7. DATA CONSTRAINTS

Peridot uses active values with data constraints to control any values that can change at run time, and these have proved to be powerful, efficient to implement, and easy for the designer to use. As described earlier, active values are like parameters to the procedure except that, when they change at run time, graphics are updated immediately.

Active values can control objects in many ways. The value can be a number that varies in a range, such as the scroll bar in Figure 9; it can control where marks are shown, as in the property sheet of Figure 2 and the menu of Figure 5; it can contain the position of an object being dragged with the mouse; etc. In fact, any property of an object can be made to depend on an active value.

Active values can be set by the application program at any time to update the graphics. For example, an application program might update the active value controlling the size of the indicator in the scroll bar of Figure 9 if the percentage of the file that is visible changes. In addition, application routines can be attached to active values, and these will be called when the active value changes. Therefore, active values are also used to pass information back to the application programs. This provides an appropriately abstract interface to the applications, since they can deal in their own high-level units (e.g., 0 to 100 for the percentage of the file seen in Figure 9, and the string names of the font properties in Figure 2k) and can be totally independent of how these values are represented graphically or set by input devices. Therefore, the graphics can be changed arbitrarily, and the application code is not affected.

While the user interface is being designed, the active values are displayed on the screen in the top Peridot window (Figure 1), and the displayed value is updated when the value changes. This makes the system more understandable, since the state of the system is always visible; the designer does not have to try to remember the values of the variables. Another factor that makes active values easy to use is that the designer can type in new values for the active value using the FixActive command. This can be used to check that the graphics change appropriately.

Another advantage of active values is that they easily support multiprocessing. Different processes in the application program can update independent active values, and the appropriate updates will happen in parallel.[2] Since input devices are implemented using active values, it is also trivial to support multiple input devices active at the same time, such as one in each hand [9].

Data constraints are efficiently implemented in Peridot. Each active value keeps a list of the graphic objects that depend on it and the application procedures to call. When the active value is updated, Peridot simply runs through these lists and performs the appropriate actions. In LISP all variable assignment is performed using functions (such as SETQ and SET). Similarly, to set an active value, an application simply calls the Peridot function SETActive. Currently, SETActive directly calls any application procedures attached to active values,

[2] Applications must deal with any issues of synchronization and mutual exclusion using the standard mechanisms built into Interlisp [54]. This has not proved to be a problem with the interface elements created using Peridot, since the various input devices usually control independent parts of the interface (e.g., one device controls the size, and the other controls the position).

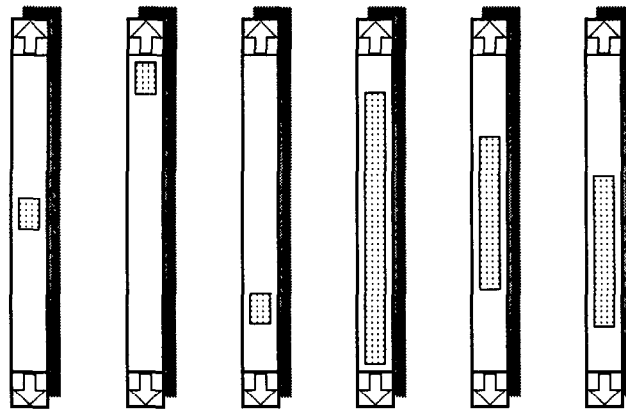


Fig. 9. Six views of the same scroll bar. The indicator changes size based on the percentage of the file seen. The user can press on the arrows or slide the indicator to move around in the file, but the size of the indicator is controlled by an application program.

but it would be possible to use message passing, remove procedure calls, or many other mechanisms to implement this connection in a distributed environment. The only requirement is that the implementation be fast enough so that application programs can be called in inner loops of mouse tracking to provide filtering on the feedback. Naturally, the implementation of the connection between the active values and the graphics is hidden from both application programs and the designer.

8. PROGRAMMING-BY-EXAMPLE ASPECTS

As described earlier, Peridot requires that the designer provide examples of typical values for the parameters and active values of the procedure. These values are used to display the user interface while it is being designed. In addition, Peridot automatically guesses from the examples the appropriate graphic constraints and control structures, and how actions depend on the mouse.

Allowing the designer to work on example data while developing an interface with Peridot is a crucial component and one of the primary contributors to Peridot's success. Otherwise, the designer could not see what the interface would look like. (What would be displayed while designing a menu if there were no list of strings?) The examples allow Peridot to use direct manipulation techniques so that the design process is concrete, rather than abstract. The interface seems more like a paint program, such as Apple MacDraw, than a programming system.

The motivation for this style is that people make fewer errors when dealing with specific examples rather than abstract ideas [45]. The programmer does not need to try to keep in mind the large and complex state of the system at each point of the computation if it is displayed on the screen. In addition, errors are usually visible immediately.

The following subsections discuss in detail how Peridot uses programming by example.

8.1 How Examples Are Used

Many PBE systems require the user to provide multiple examples in order to generate code. In some cases, Peridot infers code from single examples. This is possible because the designer is required to explicitly give a command to cause Peridot to perform the inferencing. For example, the designer issues the MOUSEDependent command to tell Peridot to look at the mouse position and to infer the generalization for the operation. For iterations, however, the designer is required to give two examples, and Peridot can therefore usually infer the need for an iteration without an explicit command from the user. For example, in Figure 2h, Peridot guesses that an iteration is needed, since there are two sets of buttons and since two elements of the parameter Items have been used.

Peridot also allows the designer to demonstrate conditionals that display special graphics and that serve as exceptions to the normal way the mouse dependencies work. For example, some items of the menu might be shown in gray if they are illegal, and horizontal lines might replace certain items (as in Figure 5). The designer can demonstrate that special actions should happen when the mouse is over these areas. For example, the reverse video feedback rectangle in Figure 5 will not appear over the items shown in gray or over the lines. These are sometimes called "negative examples," since they show the system what *not* to do. The conditions under which these exceptions can happen in Peridot are fairly limited, but the graphic response that is shown for exceptions is totally under the control of the designer.

8.2 How Inferencing Is Used

In order to make Peridot easier to use, it automatically guesses certain relationships. This frees the designer from having to know when and how to specify these relationships. Peridot uses simple condition-action rules [13] to implement these guesses. This approach is called *plausible inferencing* or *abduction* in the artificial intelligence literature. The condition part of the rules determines whether the rule seems to apply in the current context. As shown in the example of Section 4, if the condition passes, then the designer is asked whether to apply the rule or not using an English message attached to the rule. If the designer answers "yes," then the action part of the rule is applied, which changes the code of the procedure in order to add a graphic constraint.

The rules in Peridot are simple—much simpler than those used in typical artificial intelligence systems. Furthermore, there are only about 60 rules used in Peridot. The goal was to see if simple mechanisms would be sufficient, which seems to be true. Much of the complications of true rule-based "expert systems," such as learning, explaining, and backtracking, are not needed when there are only a few rules.

Peridot uses rule-based inferencing in four ways: to infer the graphic constraints that relate one object to another, to infer when control structures are appropriate, to infer how to create the control structures, and to infer how the mouse should affect the user interface.

8.2.1 *Inferring Graphic Constraints.* As shown by the example in Section 4, Peridot infers how the various graphic objects are related to each other. Previous

systems that have used constraints, such as Thinglab [5] and Juno [34], have required that the constraints be explicitly specified by the user. This has made these systems difficult to use. Previous picture beautification systems, such as PED [39], automatically infer a form of constraints, but these systems are not popular because they guess incorrectly too often and because it is difficult to determine what the program has done and to repair any damage. One reason that Peridot is more successful is that it guesses correctly more frequently, since it only needs to deal with the relationships that are typical in user interfaces, rather than all possible relationships that might be used in a general drawing. If the designer wants other relationships, they can be explicitly specified, or if they occur frequently, a programmer can easily add them to the rule set. Another reason for Peridot's success is that it assumes that guesses will occasionally be incorrect. Therefore, it always reports to the designer the rule that it is planning to apply and allows the designer to confirm or prevent its application. This gives the designer confidence that the system is not mysteriously doing strange and possibly erroneous things. In addition, the results of the inferences are always immediately visible (the objects redraw themselves after every rule is applied), so the designer can view the results and see whether they were correct or not.

Another benefit of inferring graphic constraints is that they allow the designer to draw the picture quickly and sloppily, and then Peridot automatically "beautifies" the picture by enforcing the constraints. This technique seems to be faster than using gridding or than explicitly specifying constraints, even though Peridot occasionally guesses incorrectly. Designers who have used Peridot feel that this benefit outweighs the disadvantages of taking the time to answer the questions and to correct the occasional erroneous inferences.

The rules that Peridot applies are specific to the types of objects drawn. For example, it is more likely for a string to be centered at the top of a rectangle than it is for another rectangle to be. Some of the rules specify all of the properties of an object. Examples of these are that a rectangle is the same size as another rectangle, that it is nested inside the other rectangle, or that a string is centered vertically to the right of a rectangle. Other rules only constrain some of the properties of an object. For example, one rule might cause the width and left of a rectangle to be constrained by another rectangle, and another rule may constrain the top and height by a string. These two rules are used by the feedback black rectangle in Figure 5. Other rules that constrain only some properties of objects are that they have the same size (which does not constrain the position) or that one is centered inside another (no constraint on the sizes). In general, there are constraints for most of the simple relationships found in typical user interfaces. There are currently 50 rules, and these are all listed in [28]. Of these, 16 were added based on user testing. Since most of the additional rules were added from the initial users and no new rules were needed for later users, it is expected that few new rules will be needed in the future.

The rules are expressed as simple LISP expressions, with a test, a constraint, and a message string for prompting the user. The tests naturally have thresholds built into them, so the user does not have to draw the graphics exactly. When the user draws or edits an object, Peridot goes through the rules in order, trying each test. The order is determined by the types of the objects, by the specificity

of the rule (the rules that constrain all of the properties of the object are checked first), and by which ones seemed to be the most common. Reference [28] shows the ordering for all the rules.

The message part of the rule is the string used to prompt the user for whether to apply the rule or not, as shown in Section 4. If the designer answers "yes," then the constraint is applied. If the constraint has parameters, such as how far apart the objects should be, the designer can answer "almost" and supply a new value for the parameters. If the designer answers "no," then other rules are attempted. For graphic constraints, Peridot almost always infers the correct rule within three guesses, and about 80 percent of the time, it is correct on its first guess.

8.2.2 *Inferring Control Structures.* Unlike most recent example-based programming systems, Peridot automatically infers when control structures such as iterations are appropriate. The criterion for these inferences in Peridot is straightforward. Iterations are inferred whenever the first two elements of a list are used, as shown in the example (Figure 2h). To create a dependency on an active value or a parameter, the designer must explicitly select an element of these in the upper window and then specify which property of the object depends on the selection. Peridot automatically keeps track of these dependencies and looks for uses of consecutive elements.

Conditional control structures are automatically inferred when objects depend on the mouse, as discussed in Section 8.2.4. In addition, the designer can explicitly specify that either a conditional or an iteration is desired by executing commands from the menu.

After Peridot decides that a control structure is necessary, it must then decide what graphic objects participate in it. As shown in the property sheet of Figure 2i, multiple graphic objects may be drawn in each cycle of an iteration (here, the black, white, and gray rectangles and the string are repeated in each cycle). To make the determination easier, Peridot requires that objects be drawn in the same order in each cycle. Clearly, the designer will have to be careful to create objects in the correct order, but the Copy command does this automatically.

There are other restrictions on inferred iterations in the current implementation. First, the two sets of objects must be created contiguously. This makes it significantly easier to determine how many objects to include in the cycle. Another requirement is that the element that comes from the list must be the last item drawn in each cycle. Again, this makes it easier on the implementation, since it can simply count the items in between the two items that come from the list. If the element is not the last item drawn (e.g., in Figure 2h, if the string had been drawn before the white rectangle), Peridot will still notice that an iteration is desired, but it will not be able to identify automatically which elements participate in the iteration. In this case, a message is printed, asking the designer to select all the items that are part of the iteration. Users of Peridot have not even noticed either of these restrictions, and the restrictions would not be hard to remove if it would be appropriate.

8.2.3 *Differentiating Variables from Constants.* After the objects that participate in a control structure are identified, Peridot must determine which prop-

erties of the objects are constant and which change. For example, in the property sheet of Figure 2i, Peridot determines that the text value of the string and the y value of the rectangles and the strings should change in each cycle of the iteration, but the rest of the properties remain constant.

It has been found with previous systems that inferring variables from constants is difficult, but Peridot's simple mechanism has been successful. Again, this is due to the limited domain; graphic objects in user interfaces change typically in simple ways.

Peridot compares each pair of objects and determines which attributes change and which are constant. For the attributes that change, Peridot only detects straightforward differences, for example,

—changing by a fixed amount (e.g., the y position of the rectangles in Figure 2i): The difference between the values is added to each subsequent cycle;
—depending on the item of the list (e.g., the string names in Figure 2i): Each cycle uses the next item from the list; and
—using the same relationship between the pair for all subsequent items (e.g., in the menu of Figure 5, the first string is at the top of the rectangle, so it will have a special constraint, but each subsequent string is under the previous string): The constraint in all items after the first is made to refer to the corresponding item in the previous cycle.

Peridot can easily determine which one of these to use by looking at the values of the properties. This simple strategy makes the inferencing straightforward. If Peridot detects that a control structure is plausible, but it cannot calculate how to change the objects, the designer is notified.

The major complication in generating the code for the control structures is ensuring that the constraints refer to the correct objects. When the designer creates the example drawing, the inferred graphic constraints refer to the particular example objects. When these are generalized into a control structure such as an iteration, however, Peridot must ensure that the constraints refer to the appropriate objects. For example, in Figure 2i each of the strings must be centered on the right of the correct corresponding rectangle. To solve this problem, Peridot generates the object names when the iteration is executed at run time, and then constructs the constraints based on these generated names [28].

The automatic inferencing of graphic constraints (Section 8.2.1) might also be classified as differentiating variables from constants, since the properties of objects that have constraints can change and those that do not have constraints retain their original constant values.

8.2.4 *Inferring Mouse Operations.* Another form of control structure that Peridot infers is how the mouse should affect the graphics. This includes which objects should be affected by the mouse, and when and how they should change. When the designer gives the MOUSEDependent command, Peridot looks under the simulated mouse to determine which objects are affected and where the mouse should be for the operation to be active (as shown in Figure 2k).

The designer specifies *when* the operation should happen by toggling the state of the buttons on the simulated mouse. The interaction can start after single or

multiple buttons presses (e.g., double-clicking) and either on the down or up transition of the button.

Next, Peridot infers *which* object should be affected by the mouse. Usually, it is the object under the simulated mouse's nose (as in Figure 2k), but otherwise, the designer can select the appropriate object or objects that serve as the feedback graphics.

Then, Peridot infers *how* the objects should change with the mouse. The possibilities are

(1) to choose one or more out of a set of objects (e.g., controlling which objects are selected in the property sheet (Figure 2k) or menu (Figures 5 and 7),
(2) to move in a fixed range (e.g., the scroll bar of Figure 9),
(3) to move or change size freely (e.g., to control window placement and size (Figure 6), or
(4) to blink on and off in place.

Peridot guesses which of these is appropriate by looking at the constraints on the graphic objects that are affected by the mouse. For example, the check mark in Figure 2j is centered inside an object that is part of an iteration, so Peridot guesses that it should pick items out of the iteration. If the object is constrained to move inside another object, then Peridot guesses that it should operate as a range like a scroll bar. Otherwise, Peridot guesses that the object should either blink on and off or move freely.

Peridot also needs to know *where* the interaction should operate. This is usually the object under the mouse's nose (but ignoring the feedback object). If that object is a member of an iteration, however, Peridot asks the user if the operation should perform whenever the mouse is over *any* of the elements of the iteration. In Figure 2k the check-mark interaction works for any of the boxes, not just the example one it was demonstrated over.

In order to confirm all of these inferences, Peridot asks the designer a series of questions after the MOUSEDependent command is selected. Some additional questions allow the designer to specify information that cannot be inferred from the demonstration. This includes which active value is affected by this interaction, and whether the operation should operate continuously while the mouse button is held down (i.e., have the feedback follow the mouse while the button is depressed, as in a menu) or only perform once when the button goes down. Another question determines whether the interaction adds the item under the mouse to the selected set, removes it, or adds it if it is not there and removes it if it is (toggle). In Figure 2k the designer used toggle, which is the default.

Including confirmation of the inferences, creation of an interaction requires answering about eight questions, all with single-letter responses. Peridot almost always is correct in the guesses for interactions, since there is a small number of possibilities for each choice and these are easy to identify from the demonstration.

Once all of the properties have been determined, code is generated that causes a special "interactor" object to be created at run time. This object will wait for the appropriate start event when the mouse is over the correct object and will then set the active value appropriately, which will cause the feedback to move. If

desired, the designer can add application notification procedures or filters to this active value.

9. VISUAL PROGRAMMING ASPECTS

Peridot qualifies as a visual-programming system, since it uses graphic techniques to create programs. The techniques used by Peridot are significantly different from most other visual-programming systems, however, because Peridot also uses programming by example. Many other visual-programming systems have not been successful because the graphic presentation of the program is not sufficiently abstract to overcome the disadvantage that the graphic form takes significantly more physical space to display [25]. In Peridot, however, the graphic form of the program is exactly the user interface that the designer is creating, so this is clearly well matched with the designer's needs. Also, the graphic user interface generally takes less room to display than the associated code. Another advantage in Peridot is that the system is not trying to address general-purpose programming, as in many other visual-programming languages. Therefore, more specialized techniques can be used.

Some parts of the user interface are not fully visible in Peridot. For control structures, the designer only sees the result, and there is no indication whether the objects were created due to an iteration or a conditional. This does not seem to be a problem, however, since it is usually obvious to the end user where the control structures are. In addition, it is generally not necessary for the designer to remember which graphics come from control structures and which do not, since the iterations and conditionals are created and maintained automatically by Peridot and since the individual objects in them can be selected directly and edited.

Mouse dependencies are even more abstract and do not appear in the normal graphic display. The designer must either exercise the interface or give a command to have the interactors listed in order to know what has been created. The listing appears in the prompt window and shows the name of each interactor, which active value it uses, and which objects it operates over. Since the design time is short and the procedures are small, the designer does not forget what has been created, so the listings are seldom needed.

In general, the disadvantages of having to learn a special language that describes the control structures and mouse dependencies seem to outweigh the problem of not having a visual representation of these structures, so one is not used in Peridot.

One of the problems of many visual-programming systems is that they cannot handle large programs due to a lack of modularization. In Peridot this is not a problem, since parameterized procedures are created that can be combined into full interfaces. Each user interface element is defined separately and encapsulated in its own procedure, so the designer can create interfaces out of small, modular, well-structured pieces.

10. EDITING PROGRAMS

Since Peridot is designed to be a prototyping tool, it is imperative that it support easy editing of all parts of the interface. When editing the graphic aspects, the

designer can simply select graphic objects and then give an editing command. These commands allow the objects to be erased, copied, or changed either by using the mouse or by typing in new values for the properties. It is easy to edit the graphic parts of the interface, as it is to change a picture in a drawing program such as Apple MacDraw.

It is harder to edit control structures and mouse interactions, since they do not have visual representations on the screen that can be selected. Some systems have required the user to learn a textual representation for the actions in order to allow editing (e.g., [17]), but this is undesirable because a language might be too hard for nonprogrammers to learn. Therefore, Peridot uses more direct techniques.

For editing control structures, the designer can simply select any graphic object and give an editing command. If that object is part of a control structure, Peridot will inquire whether a modification to the control structure itself is desired or whether there should be an exception to the normal way the control structure works. Exceptions are discussed in Section 8.1.

If the designer specifies that the control structure itself should be edited, then Peridot returns the display to the original objects from which the control structure was created. For an iteration, this is the original two sets of elements, and for a conditional, it is the original one element. For example, if the designer decides to change the size of the boxes in Figure 2i, Peridot would first return the display as in Figure 2h before allowing the edits. Now the designer can use all the normal editing commands to change the picture as desired. When editing is complete, then the Iteration or Conditional command is given to reinvoke the control structure.

This technique is used for three reasons. First, it is easier to ensure that the designer's edits always make sense. Otherwise, if the designer changed the fourth item of a list, what would this mean? Second, if multiple items are generated by the control structure, the designer might make intermediate edits (such as deleting an object from one group) that would cause Peridot to be unable to show the control structure consistently. Third, the list controlling the iteration or conditional might have only 1 or 0 items in it when the designer performed the edit, in which case there would not be two groups of objects for iterations or one for a conditional, so there would be nothing for the designer to select. Returning to the original two groups of objects allows the designer to have full freedom to edit in any way desired, using all the conventional editing commands.

It is even harder to edit mouse interactions because there is nothing to select. Peridot provides two ways to edit interactions. First, an interaction can be redemonstrated, and Peridot will inquire if the new interaction should replace the old one or run in parallel. Running in parallel is often used, since it is common to provide two different ways to produce the same effect. For example, in the scroll bar in Figure 9, pressing on the arrows or moving the indicator up and down affects the same active value.

The second way to edit interactions is to select an active value and give the DeleteInteractions command. Peridot then prints in the prompt window a description of each interaction that affects that active value, and asks if it should be deleted.

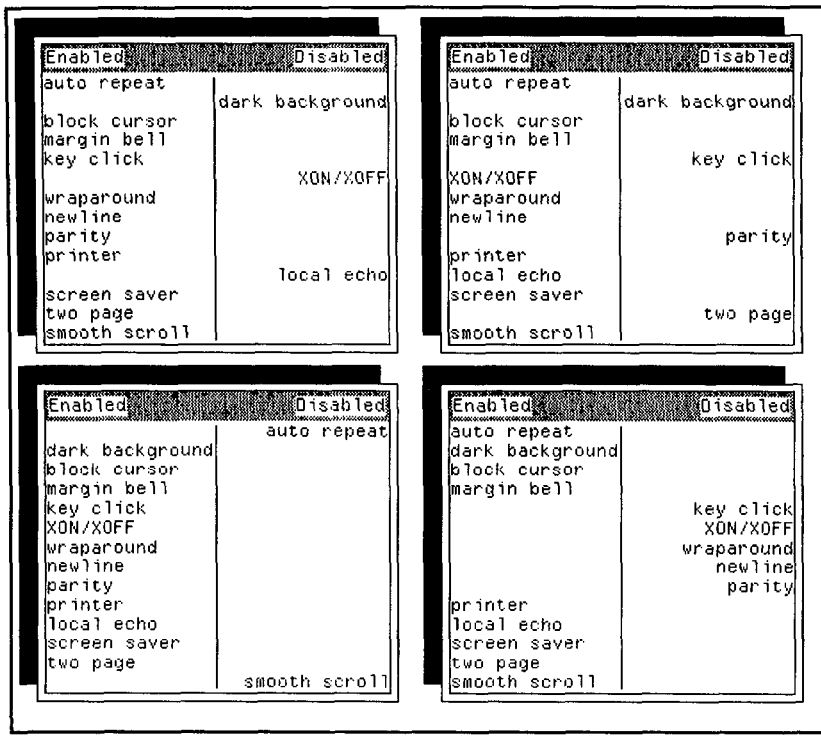
Any system that allows editing will have a certain "grain," finer than which, the items must be reentered from scratch. For example, drawing packages rarely allow a rectangle to be edited into a circle (keeping some of its properties); instead, the rectangle must be erased, and a circle drawn instead. Similarly, the grain chosen in Peridot for mouse interactions is one entire interaction. Since individual interactions are small (e.g., setting an active value when the left mouse button goes down over a menu item), this should not be burdensome. A complex interface, such as the scroll bar in Figure 9, is typically constructed from a number of small interactions, each of which takes only a few seconds to define. The added complexity for the designer of learning extra editing commands does not seem appropriate, given the ease of respecification. If this turns out to be obnoxious to future designers, however, then it would not be too difficult to allow the various parts of the interaction (the active area, the buttons, the action, the exceptions, etc.) to be edited separately.

11. USING PERIDOT-CREATED PROCEDURES

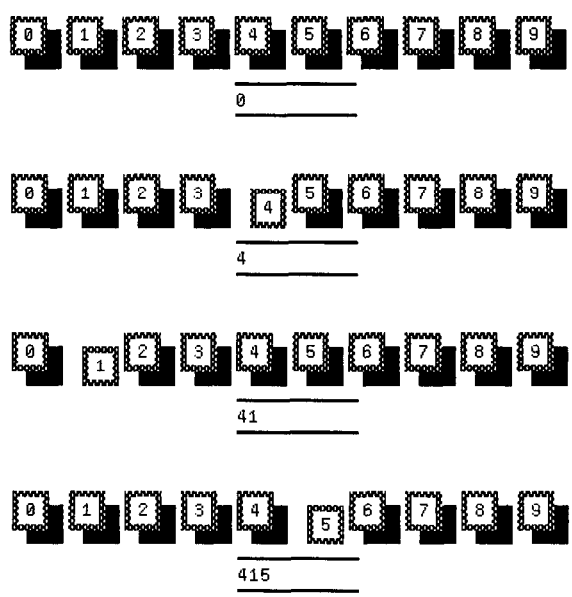
There are three ways that user interface procedures created by a UIMS can be attached to application programs at run time, and Peridot supports all three. With *UIMS control* (also called *external control*) the user interface procedures call the application when the user gives a command. Peridot supports this by attaching application procedures to active values and calling the procedures when the active value changes, as discussed in Section 7. With *application control* (also called *internal control*) the application simply calls the user interface procedures when input is desired. This is the model of most user interface tool kits, such as the Macintosh Toolbox [1]. Peridot supports this by allowing the designer to specify conditions under which the user interface procedure should exit and return a value (as discussed in Section 5.4). The final type of control is called *mixed*, which is a combination of UIMS control and application control, and clearly Peridot supports this also.

12. EVALUATION OF PERIDOT

It is difficult to quantify formally the range of user interfaces that Peridot can create, since there is no comprehensive taxonomy of interaction techniques. Informally, it is easier to describe Peridot's range by example: It can create menus of almost any form (with single or multiple items selected), property sheets, light buttons, radio buttons, scroll bars, two-dimensional scroll boxes, percent-done progress indicators, graphic potentiometers, sliders, iconic and title line controls for windows, dynamic bar charts, and many others. Thus, Peridot can create almost all of the Apple Macintosh interface (all but parts that use text input), as well as many new interfaces, such as those that use multiple input devices concurrently. For example, my advisers asked if Peridot could make a menu where the items move back and forth as they are selected (Figure 10a) and a window where pressing in different areas causes it to grow from different sides (like the X/10 uwm window manager), and Peridot was able to handle both. Other interesting effects that Peridot can provide is a button where the body of the button appears to go into the screen in "3-D" because it moves to cover its



(a)



(b)

Fig. 10. The graphic response to the mouse actions in Peridot is only limited by the creativity of the designer. In (a), text items move left and right, and in (b), number-pad buttons pretend to move in three dimensions.

shadow when pressed (see Figure 10b), and animations between the initial and final forms.

Peridot does not handle any keyboard input, but the ideas in Peridot could be extended to handle the keyboard and other types of input devices. Peridot-style demonstrational techniques cannot be used to create complex, data-dependent behavior, such as gridding and semantic feedback, so it is important to allow these to be specified another way. In Peridot, this can be handled using conventional LISP procedures that are attached to active values as filters. As it is currently implemented, Peridot also cannot handle objects that are created dynamically at run time. For example, Peridot cannot be used to draw or define the behavior of circuit elements that the end user will create from a palette (although the palette could be designed). Similarly, Peridot will not handle displays of trees or graphs or other application-specific data. Also, Peridot does not handle higher-level sequencing of the low-level user interface elements; that is, to get a submenu to appear after a menu item is selected, a LISP procedure must be written.

In order to evaluate how easy Peridot is to use, an informal experiment was run where 10 people used the system for about 2 hours each. Of these people, five were experienced programmers, and five were nonprogrammers who had some experience using a mouse. The results of this test are encouraging. After about $1\frac{1}{2}$ hours of guided use of Peridot, the subjects were able to create a menu of their own design unassisted. This demonstrates that one basic goal of Peridot is fulfilled: Nonprogrammers are able to create user interface elements using Peridot.

In addition, programmers will appreciate using Peridot to define graphic parts of user interfaces, since it is faster and more natural than conventional programming. As a small, informal experiment, six expert programmers implemented a particular menu using their favorite hardware and software environments. Some wrote the menu by hand, and others modified existing code. The results were that with Peridot the time to create the menu ranged from 4 to 15 minutes, but programming took between 50 and 500 minutes [28]. Therefore, using Peridot appears to be significantly faster.

13. FUTURE WORK

Unfortunately, the implementation of Peridot can no longer be run. We are building on the success of Peridot in two directions. First, we are creating a large-scale user interface development environment based on many of the ideas in Peridot. This new system, called Garnet [33], contains an object-oriented graphics system, a constraint system [49], a package that encapsulates the input handling [30], and a user interface editor called Lapidary [32]. The goals of Garnet are to demonstrate that the ideas in Peridot can be used to develop real user interfaces for actual programs, and to investigate new algorithms and techniques for specifying and implementing user interfaces using examples and constraints. Garnet will also try to overcome most of the limitations discussed in Section 12.

Another direction is to investigate other applications for programming-by-example-style user interfaces. I believe that this technique can be used to provide

programming capabilities to nonprogrammers in many domains, including text formatting, business graphics, and data and scientific visualization.

14. CONCLUSIONS

Peridot uses constraints, programming by example, and visual programming to allow nonprogrammers to create user interfaces by demonstration. Unlike many previous attempts to use these techniques, they have been effectively used and efficiently implemented in Peridot. Graphic and data constraints are used to maintain important relationships, programming by example is used so that these relationships can be automatically inferred from examples, and visual programming allows the designer to create the user interface graphically and to see the interface as it develops. As the user interface is designed, Peridot creates reasonably efficient code that can be used with actual application programs. Therefore, the user interface designer is not necessarily just prototyping; the actual end-user interface code can be produced, even by a nonprogrammer.

The algorithms in the generated code are comparable to those that would be created by hand, and do not require any complex constraint-satisfaction techniques. The procedures that are created can have parameters and return values, just like the procedures normally found in interaction technique libraries. In addition, calls to application programs from within the user interface are appropriately parameterized. Therefore, Peridot promotes structured design and creates well-structured code (unlike many previous UIMSs). Peridot is the first user interface management system to provide all of these capabilities.

Peridot also demonstrates that visual programming, programming by example, and constraints can be successfully integrated in practical, useful, and easy-to-use systems. One important challenge now is to find other application areas that can also effectively use these techniques.

ACKNOWLEDGMENTS

I want to thank Xerox Canada, Inc., for the donation of the Xerox workstations and Interlisp environment on which Peridot was built. I would also like to thank my thesis adviser, Bill Buxton, for his support and good ideas, and Bernita Myers and the referees for help with this paper. The School of Computer Science at Carnegie Mellon University supported the preparation of this paper.

REFERENCES

1. APPLE COMPUTER, INC. *Inside Macintosh.* Addison-Wesley, Reading, Mass., 1985.
2. BARTH, P. An object-oriented approach to graphical interfaces. *ACM Trans. Graph. 5*, 2 (Apr. 1986), 142–172.
3. BIERMANN, A. W., AND KRISHNASWAMY, R. Constructing programs from example computations. *IEEE Trans. Softw. Eng. SE-2*, 3 (Sept. 1976), 141–153.
4. BORNING, A. Thinglab—A constraint-oriented simulation laboratory. Tech. Rep. SSL-79-3, Xerox Palo Alto Research Center, Palo Alto, Calif., July 1979.
5. BORNING, A. The programming language aspects of Thinglab: A constraint-oriented simulation laboratory. *ACM Trans. Program. Lang. Syst. 3*, 4 (Oct. 1981), 353–387.
6. BORNING, A. Defining constraints graphically. In *Human Factors in Computing Systems, Proceedings of SIGCHI 86* (Boston, Mass., Apr. 13–17). ACM, New York, 1986, pp. 137–143.

7. BORNING, A., AND DUISBERG, R. Constraint-based tools for building user interfaces. *ACM Trans. Graph. 5*, 4 (Oct. 1986), 345–374.
8. BUNEMAN, O. P., AND CLEMONS, E. K. Efficiently monitoring relational databases. *ACM Trans. Database Syst. 4*, 3 (Sept. 1979), 368–382.
9. BUXTON, W., AND MYERS, B. A study in two-handed input. In *Human Factors in Computing Systems, Proceedings of SIGCHI 86* (Boston, Mass., Apr. 13–17). ACM, New York, 1986, pp. 321–326.
10. BUXTON, W., LAMB, M. R., SHERMAN, D., AND SMITH, K. C. Towards a comprehensive user interface management system. In *Proceedings of SIGGRAPH 83. Comput. Graph. 17*, 3 (July 1983), 35–42.
11. CARDELLI, L. Building user interfaces by direct manipulation. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software* (Banff, Alberta, Canada, Oct. 17–19). ACM, New York, 1988, pp. 152–166.
12. CARDELLI, L., AND PIKE, R. Squeak: A language for communicating with mice. In *Computer Graphics, Proceedings of SIGGRAPH 85* (San Francisco, Calif., July 22–26). ACM, New York, 1985, pp. 199–204.
13. CHARNIAK, E., AND MCDERMOTT, D. *An Introduction to Artificial Intelligence.* Addison-Wesley, Reading, Mass., 1985.
14. FOLEY, J. D., AND MCMATH, C. F. Dynamic process visualization. *IEEE Comput. Graph. Appl. 6*, 2 (Mar. 1986), 16–25.
15. GOSLING, J. Algebraic constraints. Tech. Rep. CMU-CS-83-132, Computer Science Dept., Carnegie Mellon Univ., Pittsburgh, Pa., 1983.
16. GOULD, L., AND FINZER, W. Programming by rehearsal. Tech. Rep. SCL-84-1, Xerox Palo Alto Research Center, Palo Alto, Calif., May 1984. (A short version appears in *Byte 9*, 6 (June 1984).)
17. HALBERT, D. C. Programming by example. Ph.D. thesis, Computer Science Division, Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, Calif., 1981.
18. HENDERSON, D. A., JR. The Trillium user interface design environment. In *Human Factors in Computing Systems, Proceedings of SIGCHI 86* (Boston, Mass., Apr. 13–17). ACM, New York, 1986, pp. 221–227.
19. HENRY, T. R., AND HUDSON, S. E. Using active data in a UIMS. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software* (Banff, Alberta, Canada, Oct. 17–19). ACM, New York, 1988, pp. 167–178.
20. JACOB, R. J. K. A state transition diagram language for visual programming. *Computer 18*, 8 (Aug. 1985), 51–59.
21. JACOB, R. J. K. A specification language for direct manipulation interfaces. *ACM Trans. Graph. 5*, 4 (Oct. 1986), 283–317.
22. LELER, W. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley, New York, 1988.
23. LIEBERMAN, H. Constructing graphical user interfaces by example. In *Proceedings of Graphics Interface, GI 82* (Toronto, Ontario, Canada, May). 1982, pp. 295–302.
24. MAULSBY, D. L., AND WITTEN, I. H. Inducing procedures in a direct-manipulation environment. In *Human Factors in Computing Systems, Proceedings of SIGCHI 89* (Austin, Tex., Apr. 30–May 4). ACM, New York, 1989, pp. 57–62.
25. MYERS, B. A. Visual programming, programming by example, and program visualization: A taxonomy. In *Human Factors in Computing Systems, Proceedings of SIGCHI 86* (Boston, Mass., Apr.). ACM, New York, 1986, pp. 59–66.
26. MYERS, B. A. Creating interaction techniques by demonstration. *IEEE Comput. Graph. Appl. 7*, 9 (Sept. 1987), 51–60.
27. MYERS, B. A. Creating user interfaces by demonstration: The Peridot user interface management system. 15 minute video tape. *Siggraph Video Rev. 59*, 2 (April, 1990). School of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa.
28. MYERS, B. A. *Creating User Interfaces by Demonstration.* Academic Press, Boston, 1988.
29. MYERS, B. A. User interface tools: Introduction and survey. *IEEE Softw. 6*, 1 (Jan. 1989), 15–23.
30. MYERS, B. A. Encapsulating interactive behaviors. In *Human Factors in Computing Systems, Proceedings of SIGCHI 89* (Austin, Tex., Apr. 30–May 4). ACM, New York, 1989, pp. 319–324.

31. MYERS, B. A., AND BUXTON, W. Creating highly interactive and graphical user interfaces by demonstration. In *Computer Graphics, Proceedings of SIGGRAPH 86* (Dallas, Tex., Aug. 18–22). ACM, New York, 1986, pp. 249–258.
32. MYERS, B. A., VANDER ZANDEN, B., AND DANNENBERG, R. B. Creating graphical objects by demonstration. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (Williamsburg, Va., Nov. 13–15). ACM, New York, 1989, pp. 95–104.
33. MYERS, B. A., GIUSE, D., DANNENBERG, R. B., VANDER ZANDEN, B., KOSBIE, D., MARCHAL, P., PERVIN, E., MICKISH, A., AND KOLOJEJCHICK, J. A. The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp. Tech. Rep. CMU-CS. 90–117. Carnegie Mellon School of Computer Science, March 1990.
34. NELSON, G. Juno, a constraint-based graphics system. In *Computer Graphics, Proceedings of SIGGRAPH 85* (San Francisco, Calif., July 22–26). ACM, New York, 1985, pp. 235–243.
35. NIX, R. P. Editing by example. *ACM Trans. Program. Lang. Syst. 7*, 4 (Oct. 1985), 600–621.
36. OLSEN, D. R., JR., ED. ACM SIGGRAPH workshop on software tools for user interface management. *Comput. Graph. 21*, 2 (Apr. 1987), 71–147.
37. OLSEN, D. R., JR., AND DEMPSEY, E. P. Syngraph: A graphical user interface generator. In *Computer Graphics, Proceedings of SIGGRAPH 83* (Detroit, Mich., July 25–29). ACM, New York, 1983, pp. 43–50.
38. OLSEN, D. R., JR., DEMPSEY, E. P., AND ROGGE, R. Input–output linkage in a user interface management system. In *Computer Graphics, Proceedings of SIGGRAPH 85* (San Francisco, Calif., July). ACM, New York, 1985, pp. 225–234.
39. PAVLIDIS, T., AND VAN WYK, C. J. An automatic beautifier for drawings and illustrations. In *Computer Graphics, Proceedings of SIGGRAPH 85* (San Francisco, Calif., July). ACM, New York, 1985, pp. 225–234.
40. PFAFF, G. R., ED. *User Interface Management Systems.* Springer-Verlag, New York, 1985.
41. RAMAMOORTHY, C. V., SHEKHAR, S., AND GARG, V. Software development support for AI programs. *Computer 20*, 1 (Jan. 1987), 30–40.
42. SHAW, D. E., SWARTOUT, W. R., AND CORDELL GREEN, C. Inferring Lisp programs from examples. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence, IJCAI 75* (Tbilisi, USSR, Sept.). 1975, pp. 260–267.
43. SHNEIDERMAN, B. Direct manipulation: A step beyond programming languages. *Computer 16*, 8 (Aug. 1983), 57–69.
44. SINGH, G., AND GREEN, M. A high-level user interface management system. In *Human Factors in Computing Systems, Proceedings of SIGCHI 89* (Austin, Tex., Apr. 30–May 4). ACM, New York, 1989, pp. 133–138.
45. SMITH, D. C. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought.* Birkhäuser Verlag, Basel, 1977.
46. STEFIK, M., BOBROW, D. G., AND KAHN, K. M. Integrating access-oriented programming into a multi-paradigm environment. *IEEE Softw. 3*, 1 (Jan. 1986), 10–18.
47. STEVENS, A., ROBERTS, B., AND STEAD, L. The use of a sophisticated graphics interface in computer-assisted instruction. *IEEE Comput. Graph. Appl. 3*, 2 (Mar.–Apr. 1983), 25–31.
48. SUTHERLAND, I. E. SketchPad: A man–machine graphical communication system. In *AFIPS Spring Joint Computer Conference.* AFIPS Press, Reston, Va., 1963, pp. 329–346.
49. SZEKELY, P. A., AND MYERS, B. A. A user interface toolkit based on graphical objects and constraints. In *Proceedings of the ACM Conference on Object-Oriented Programming; Systems Languages and Applications*; OOPSLA 88. *SIGPLAN Not.* (ACM) *23*, 11 (Nov. 1988), 36–45.
50. TESLER, L. The Smalltalk Environment. *Byte Mag. 6*, 8 (Aug. 1981), 90–147.
51. VANDER ZANDEN, B. T. Constraint grammars—A new model for specifying graphical applications. In *Human Factors in Computing Systems, Proceedings SIGCHI 89* (Austin, Tex., Apr. 30–May 4). ACM, New York, 1989, pp. 325–330.
52. WASSERMAN, A. I., AND SHEWMAKE, D. T. Rapid prototyping of interactive information systems. *ACM Softw. Eng. Notes 7*, 5 (1982), 171–180.
53. WILLIAMS, G. The Apple Macintosh computer. *Byte 9*, 2 (Feb. 1984), 30–54.
54. XEROX CORPORATION. *Interlisp Reference Manual.* Xerox Corp., Pasadena, Calif., 1983.

Received March 1988; revised June 1989 and October 1989; accepted October 1989