Brigham Young University

BYU ScholarsArchive

Faculty Publications

1998-01-01

Macros by Example in a Graphical UIMS

Dan R. Olsen Jr.
dan_olsen@byu.edu

Jonathan Turner

Stephen Bart Wood

John R. Dance

Follow this and additional works at: https://scholarsarchive.byu.edu/facpub

Part of the Electrical and Computer Engineering Commons

Original Publication Citation

Olsen, D. R., Jr., and J. R. Dance. "Macros by Example in a Graphical UIMS." Computer Graphics and Applications, IEEE 8.1 necessary when editing micros by example (1988): 68-78

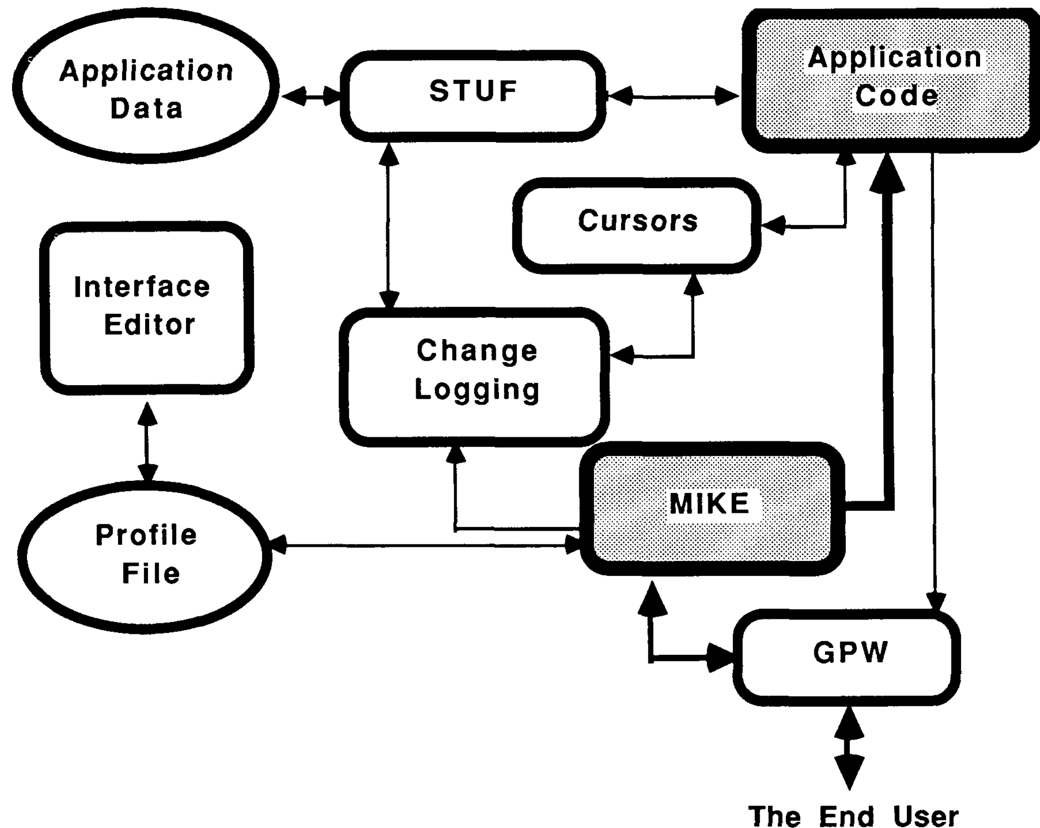
BYU ScholarsArchive Citation

Olsen, Dan R. Jr.; Turner, Jonathan; Wood, Stephen Bart; and Dance, John R., "Macros by Example in a Graphical UIMS" (1998). *Faculty Publications*. 654.
https://scholarsarchive.byu.edu/facpub/654

This Peer-Reviewed Article is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Faculty Publications by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.

Macros by Example in a Graphical UIMS

Dan R. Olsen, Jr.
Brigham Young University
John R. Dance
Apple Computer, Inc.



A macro facility that allows end users to extend interactive graphical applications is presented as part of a user interface management system (UIMS). Such macros are expressed by example; that is, the end user programs the macro in the application's generated user interface. Problems with macros by example in graphical applications are explored, and requirements to accommodate such a facility are defined for the UIMS dialogue model. Existing UIMS models are reviewed relative to these requirements, and the unique facilities of the MIKE (Menu Interaction Kontrol Environment) semantics-based model are presented. The implementation of the macro-by-example system is discussed, as well as the particular implementation of a multicommand UNDO facility, which is necessary when editing macros by example.

For the past several years many researchers, including ourselves, have been working on user interface management systems (UIMS). The purpose of such systems is to greatly speed the development of interactive graphical applications. A number of such systems have been built, and there is evidence that they do reduce the cost of software development, as well as enhance the reliability and consistency of the resulting applications.

 0272-1716/88/0100-0068$01.00 1988 IEEE

Authorized licensed use limited to: Brigham Young University. Downloaded on January 29, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

These gains, however, carry a price. Each UIMS, of necessity, carries with it a bias toward a particular set of interactive styles. Some programmers will still choose to implement a user interface by hand rather than use a UIMS, so as to preserve a freedom of style. We feel that programmers will be truly motivated to use a UIMS if it provides new user interface capabilities not readily implementable without a UIMS approach.

This article describes a capability allowing end users to customize their user interfaces by adding new commands built as macros out of existing commands and to integrate the new commands into the dialogue of the user interface. We are particularly interested in specifying macros by example or demonstration. Our work is heavily influenced by Halbert's *Programming by Example.*[1] In Myers' taxonomy of visual specification techniques,[2] our system would be classified as "macros with examples" rather than "macros by example," but the latter term seems to be more widely used.

Obviously, a macro capability could be added to an application without using a UIMS, but, as we demonstrate, a truly rich macro capability requires implementation of most of a UIMS's functions. In particular, we show that a semantics-based UIMS such as MIKE (Menu Interaction Kontrol Environment)[3] is more readily adapted to such a facility than a syntax-based UIMS. A semantics-based model begins with a specification of the application code to be invoked by the user interface and then refines the interactive facade that is placed over this code.

We also discuss the issues that arise in defining graphical macros by example and some of the deficiencies of existing approaches. Then we determine the suitability of various UIMS models for solving these problems and show how the unique features of the command-based model used in the MIKE UIMS make possible some solutions to the problems identified. Finally, we review MIKE's macro-by-example facility.

Macros by example in interactive programs

Many interactive programs have supported macros by example in the form of simple keystroke macros. In such systems the user turns on macro recording, performs the desired operation or operations, and then stops the recording and stores the macro. When the macro is stored, some control character or sequence is bound to the macro. Whenever the invocation sequence occurs, the saved keystrokes are fed to the program as if they had been entered by the user. This is an intuitive and easily taught method for combining primitive capabilities into larger ones. It is also relatively easy to implement by filtering the input, so that the remainder of the application need not know of its existence.

Reviewing macros by example in a graphical environment, we have identified the following issues that must be addressed:

- Problems inherent in preserving graphical interactions for later use.
- Parameterization or generalization of macros.
- Visual presentation of the macro body.
- Editing of macros while in demonstration mode.
- Specification of other than straight-line flow of control.
- Integration of macros into the existing interactive dialogue.

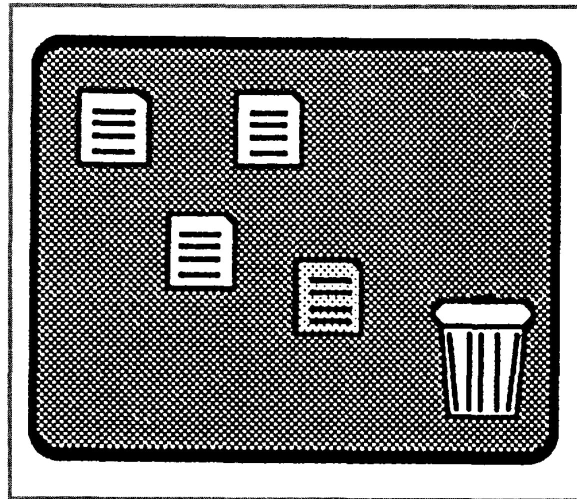


Figure 1. A desktop.

Preserving graphical interactions

We might consider adding to an application the capability of preserving graphical interactions by recording sample and event requests and then playing them back as a macro. This does not work, however, as can be shown by the example in Figure 1.

In some applications the user deletes a file by selecting the file's icon with the mouse and then selecting the trash can icon. The input events consist of two mouse selections. The problem arises when the user saves these events as part of a macro body for some later execution. These two mouse selections have meaning at later macro invocation time only if the file to be deleted and the trash are at the same locations as at macro definition time, which is rarely the case for the file and sometimes not the case for the trash can.

What is really needed is a specification of some file name, or other location-independent file identifier, and the specification that it is to be deleted. Saving only the first mouse location specifies a file at a particular loca-

Authorized licensed use limited to: Brigham Young University. Downloaded on January 29, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

tion, which is overconstrained. For the second mouse selection, the trash can icon—or rather its associated semantic meaning—should be saved. The problem of the second selection can be resolved by saving logical events (i.e., the trash can icon selection) rather than physical events (i.e., the mouse location). However, the semantic references illustrated by the file selection are not available in a simple sequence of either physical or logical events. The issue is treated in greater depth by Halbert.[1]

Keystroke macros work in text editors because the screen location is an artifact of the semantic current insert position. The interface is defined as a mapping between inputs and semantic actions, which are reflected on the screen. In a text editor, a given input event is semantically defined regardless of the appearance of the screen.

In our graphical example the inputs are meaningless without the picture, and saving them for future use out of context is of little or no value in reproducing semantic meaning. The problem lies in the fact that graphics is a tool for manipulating objects in semantic domains unrelated to graphics. In the example above, the operations are defined on files, not on points or icons. The points and icons are a presentation—they are visualization vehicles for the underlying semantics.

An effective macro system must therefore save semantic operations rather than the actual input events originally used to express them. The problems that arise when trying to make this mapping are discussed below.

Distinguishing the semantic operand

The first problem is distinguishing exactly what the semantic operand is. In the preceding example the operand was the file and the action was delete, as indicated by selecting the trash can. If, however, the trash can was not selected but rather another location on the desktop, then the operation is to move the file, and the second point is a location to move it to. In this second case the point itself is the operand because it is a geometric location within the desktop.

Moving the file icon to another window, however, is not simply a geometric positioning. A change of directory is implied in addition to the change of position. Further complications might arise if the user opens several directory windows before finding the desired one. The intermediate steps are unnecessary to the semantic meaning of the action; they simply support visual aids. Because of the dialogue model used, our implementation resolves some but not all of these issues. Several remain open research topics.

Parameterization by example

A second problem is one of parameterization of macros. In simple keystroke macros there are no parameters. The behavior of the macro is defined entirely in terms of current application state information such as the current insert position or the current search pattern. In direct-manipulation interfaces we attempt to be as modeless as possible; thus such current state information is not as meaningful. When creating a macro by demonstrating an interactive command sequence, we must differentiate between example parameter values and constants. To build an archive macro that is to take the selected file and place it in a special archive directory, the example sequence would be to move the file's icon into the archive directory's window. In this example, the selected file is a parameter that will change from invocation to invocation while the destination directory will remain constant.

It is also important to differentiate between the file parameter and the dialogue fragment used in specifying its example value. In Halbert's work[1] the parameterization is inferred from the example and then shown in a special macro window. The end user can then edit this macro specification later to make the differentiations between constants, parameters, and parameter arguments. Our approach is less sophisticated, but we believe that it is more direct.

Presenting and editing macros

When creating macros by demonstrating them, a problem arises in presenting the macro body itself. Most keystroke macro systems do not provide any ability to view the macro. Tinker[4] presents the code being created as a Lisp function. In Halbert's work, the commands being entered are presented in a separate window using English and pictographs. What is not provided in these systems is the ability to edit the macro while demonstrating it.

Keystroke macros, aside from not providing an editable presentation of the macro, have the problem of differentiating between keystrokes that are part of the example and those intended to edit the example. This problem is alleviated somewhat in an environment supporting overlapping windows, since any event directed at the macro window can be considered part of the editing process. A more difficult problem, however, is updating the state of the interactive application in response to the editing process, while in demonstration mode.

Since each command is executed as the macro is recorded, if a user wants to move backward in a macro, the steps must be undone so that the application is in an appropriate state. Take, for example, the sample macro in Figure 2. The user may want to move back to the Open Picture command by pointing at it with the mouse. The problem that arises is the undoing of all the intervening commands. The requirement of being able to undo an arbitrary number of commands imposes a serious constraint on how the interactive application is implemented. This need for a semantic UNDO has led most systems to edit macros off line rather than in example or demonstration mode.

Authorized licensed use limited to: Brigham Young University. Downloaded on January 29, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

Control flow in macros by example

Complexity is created because macros by example lead most naturally to a straight-line flow of control. Such macros are not particularly valuable because they do not allow intelligent or flexible services to be programmed. There are several approaches to this problem. Tinker[4] has the user provide multiple examples of the macro. The system then detects the difference and asks for a predicate that differentiates between the examples. In Programming by Example, iterative and conditional constructs must be added later when editing the macro off line. Specifying conditionals requires predicates for making the decision. Most interactive applications do not have a concept of Boolean predicates, because the interactive user—rather than the user interface—is expected to make all the decisions.

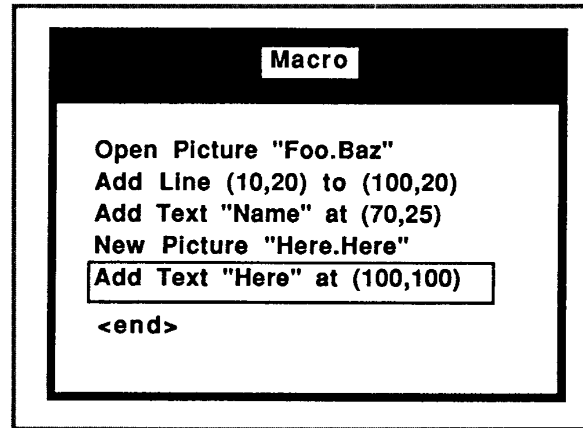


Figure 2. Example macro.

Integrating macros into the application's user interface

A final problem in extending an application by user-defined macros is integrating them into the existing application dialogue. Most graphical applications that allow such macros provide special syntax for invoking the macros, and they frequently do not provide the same level of user friendliness in binding arguments to macros as they do for built-in commands. Usually the macros are an awkward add-on to the original application.

Macros by example in a UIMS

Our work does not necessarily solve all of the problems described above. In most cases we have used solutions developed by others. Our primary interest was to provide macros by example in a UIMS. Providing such a macro capability in a UIMS allows any application built with the UIMS to have such a capability. We have found that some of the features present in a UIMS significantly ease the development of a macro-by-example system.

To support macros by example in a UIMS it must be possible to perform the following actions without disrupting the normal functioning of the user interface:

- Log complete semantic commands rather than primitive input events while the user is invoking them to demonstrate the macro body.
- Identify individual commands or closure points and generate a textual or pictographic presentation of that command when displaying the macro body.
- Undo multiple commands so as to make editing of the macro possible.
- Explicitly identify and reference macro parameters.
- Integrate macro invocation into the dialogue of the existing application, including the binding of arguments to the macro invocation.

Now we will review the dialogue models used in existing UIMSs relative to how they might satisfy these requirements. We will then present the dialogue model of the MIKE UIMS and show how it is better suited for supporting macros by example. Following this will be a short discussion of how macros by example are actually supported in MIKE.

UIMS dialogue models

UIMS models can be divided into several sets. The first consists of systems based on a syntactic dialogue specification, including grammar-based systems[5,6] and transition network systems.[7-9] In a formal language sense and in the kind of features they support, both approaches are essentially equivalent. The process is one of accepting inputs, comparing them against the present parse state, invoking some semantic action, and proceeding to a new state. A related model is the dialogue tree, such as the one in Tiger.[10] It has many of the same characteristics as grammatical approaches but is tied to its menu-traversal algorithm rather than a grammar or state machine.

A second set of UIMS models consists of object/event systems. Such systems typically use a combination of input events and selected visual objects to make a binding to a particular semantic action.[11-14]

Both syntactic and object/event models have some difficulty in meeting the requirements for macros by example presented above.

Limitations of syntactic and event models

Because a UIMS is in control of the dialogue, it is relatively easy to log either input events or semantic actions. In the syntactic or event models, however, it is somewhat more difficult to identify complete semantic commands.

Authorized licensed use limited to: Brigham Young University. Downloaded on January 29, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

Typically the specification of a command and its arguments is spread over several semantic actions. For example, in drawing a line there might be one action to save the first point, a second to save the other point, and a third to actually get the line drawn. The sequence would be further obscured by actions to generate the appropriate prompts, set up the menus, or handle rubberbanding of the line. Rubberbanding is especially a problem because the semantic action log would now contain hundreds of echoing actions that are not useful in a macro context.

An event- or syntax-based UIMS does not have the necessary information to decide which actions should be saved. Such UIMSs could be augmented with special conventions and possibly attribute-evaluation techniques to define complete semantic commands, but characteristics allowing this are not inherent in syntactic or event-based models. Object/event models, as in SmallTalk-style messages, have many of the same characteristics found in MIKE and would be amenable to techniques similar to those described in this article. Such models support rather complex messages with semantic meaning rather than simple physical or logical events.

The presentation of the macro body also becomes a problem because semantic actions are typically hidden from the interactive user. As such they are not in an externally presentable form. We would like such a presentation of a semantic command to be as close as possible to the actual prompts and echoes used to produce it interactively. In syntactic or simple event UIMS models, the prompt and echo information is typically integrated into the semantic actions and is not available for the UIMS's use.

The need for an UNDO is a problem that none of these UIMSs can solve on their own. Constructing a macro involves semantic actions that modify data structures not under the UIMS's control. Without access to these structures, it is impossible to undo the changes.

Some UIMSs have a syntactic-level UNDO facility for rubbing out or canceling partially completed commands. Syngraph,[6] for example, has rubout nonterminals that log the input events and update the prompts and menus without calling any semantic routines. This feature allows for rubout, because only information under UIMS control has been modified. When the rubout nonterminal is completed, the logged sequence is reevaluated and the semantic actions performed. Once this has been done, that dialogue fragment can no longer be rubbed out.

A similar facility is found in Tiger,[10] where an "implicit reject" is possible. In this case special semantic routines are called to handle semantic recovery. Here again, only a localized portion of the dialogue can be undone. This is not a capability sufficient for editing macros in demonstration mode.

Parameterization of macros is particularly difficult in syntactic or event-based UIMSs because there is no semantic distinction between an argument and a command. All semantics are treated as atomic code fragments or messages. Any concept of action/argument bindings is buried in the semantics. If the UIMS is not able to differentiate between argument specifications and actions, it will not be able to specify that a particular argument is to be parameterized.

The inability to differentiate between arguments and actions causes additional problems when the user wants to integrate parameterized macros into the dialogue. There are only three options for specifying the arguments when invoking a macro: The user must specify the arguments in some special (frequently textual) syntax, which is out of sync with the rest of the dialogue style; modify the grammar, transition network, or event handlers, a procedure usually beyond the capabilities of end users; or specify the parameterization off line in a special macro edit mode.

All of these options are possible and may actually be worthwhile to achieve end user dialogue extensibility, but they are not really satisfying. We have sought to make macro specification occur as much as possible in demonstration mode, so that the process will be as natural as possible for an end user of the application's interface.

Macros by example and MIKE

MIKE (Menu Interaction Kontrol Environment) is a UIMS that grew out of concepts of programming language design rather than formal languages or interactive device-handling concepts. Only a very sketchy discussion of MIKE[3] is provided here. In MIKE the basic definition of an interactive interface is a set of data types and a set of functions and procedures that can operate on data objects of those types. This approach is similar, if not identical, to the objects-and-actions model proposed by others.[15] It is essentially data abstraction and could be adapted quickly to an object-oriented model. For example, if we are designing a simple drawing package, the following object types could be defined as having some meaning to the interaction:

Point, String, DrawPrimitive, Picture

On the basis of these types we might then define the following commands and functions:

NewPicture(Name: String)
Create a new picture.

PictureName(Name: String): Picture
Identify a picture by name.

IdentifyPicture(Location: Point): Picture
Identify a picture by pointing at it.

Authorized licensed use limited to: Brigham Young University. Downloaded on January 29, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

OpenPicture(Pict: Picture)
Open an existing picture.

DeletePicture(Pict: Picture)
Delete an existing picture.

AddLine(P1, P2: Point)
Add a line primitive to the open picture.

AddText(At: Point; Txt: String)
Add a text primitive to the picture.

MovePrim(Prim: DrawPrimitive, To: Point)
Move the selected primitive to a new location.

DeletePrim(Prim: DrawPrimitive)
Delete the selected primitive.

PickPrim(Where: Point): DrawPrimitive
Select a primitive by pointing at it.

Although this is a rather simple application, it illustrates MIKE's dialogue model. Each of these procedures and functions is an application routine exposed to the interactive user. In addition to the object types DrawPrimitive and Picture, which are defined by the application itself, there are a number of types that are predefined and automatically supplied by MIKE. They are

Integer, Real, Point

Key
A one-byte code identifying a function button or keyboard key.

String
A character string.

These predefined types identify specific interactive techniques automatically provided by MIKE. It is possible to add other techniques such as rubberband lines, rubberband rectangles, or file names. An approach used in MIKE is to make as many different input techniques as possible available simultaneously. For example, in specifying an integer value, a user has the following options:

- Type in an integer number.
- By typing its name, select a function that returns an integer.
- Select from a menu a function that returns an integer.
- Select an integer function by striking the function button bound to it.
- Select an integer function or value by picking an active viewport that has such an integer item bound to it.

The controlling factor is the desired operand type. This arrangement allows the end user to select the most natural approach, rather than forcing the designer to determine it. Because these techniques for selecting an option and performing any needed interaction are localized in MIKE, it is possible to differentiate between an argument and its specification.

We have sought to make macro specification occur as much as possible in demonstration mode, so that the process will be as natural as possible for an end user of the application's interface.

The above description of the user interface to this drawing application is readily understood by or explained to anyone with some familiarity with computers. The basic notion is that a command does something to its operands and/or returns a value. The operands and result-value types are given names from the user's application domain. Such interactive command definitions are specified using MIKE's interface editor, which is itself written using MIKE. Note also that the semantic functions such as PickPrim form an integral part of the argument specification and allow the application to expand the set of argument types beyond the primitive ones provided by MIKE.

From such procedure and function definitions, MIKE creates a default syntax for interactively specifying commands. A menu is created from all the procedures that do not return a result. The user can select a command from this menu, and a prompt is issued for the command's first parameter. The name of the parameter is used as the prompt, and the type of the parameter identifies what is wanted.

Using this parameter type, MIKE creates a menu of all functions returning that type. If the type corresponds to one of the primitive types, then the corresponding interactive techniques are enabled. The process continues until a complete command has been entered, at which point the semantic procedure and/or functions that have

Authorized licensed use limited to: Brigham Young University. Downloaded on January 29, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

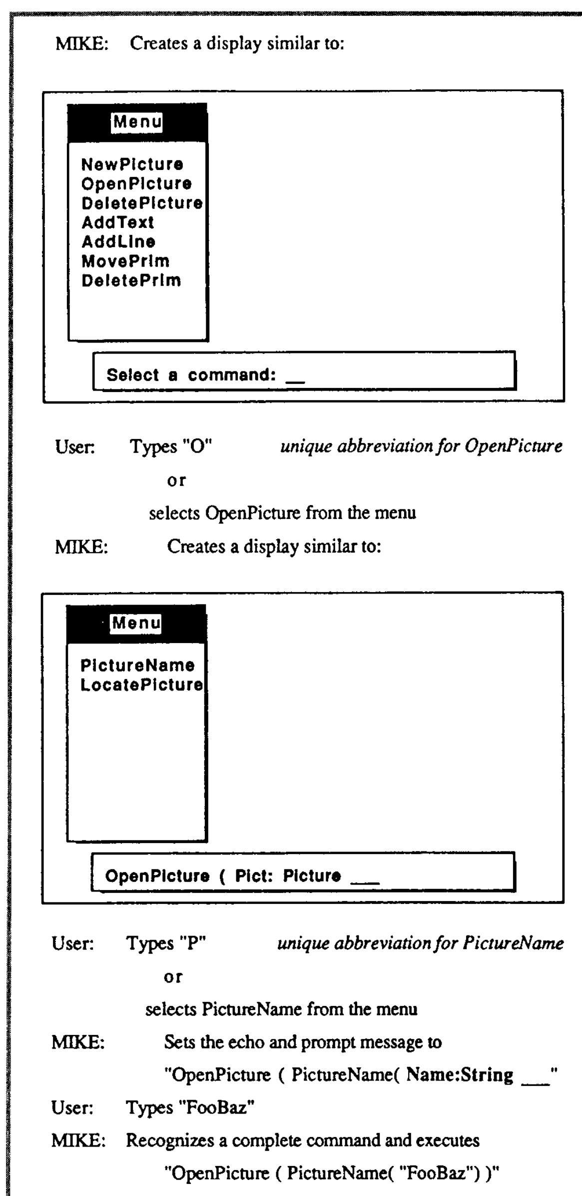


Figure 3. Example dialogue.

been specified are called. The example dialogue in Figure 3 illustrates the procedure.

The default syntax shown in Figure 3 is not always user friendly. For example, selecting LocatePicture from a menu and then pointing at the picture is rather awkward. Using MIKE's interface editor, the dialogue designer can form the menus into trees, draw icons and attach commands or partial expressions to them, and bind function buttons or other input events to commands, as well as change the echoes and prompts to more English-like statements. In addition, the interface editor can attach commands such as LocatePicture to windows so that the function is automatically selected when the window containing picture names is selected.

All these tailoring operations are performed in a WYSIWYG style easily learned by nonprogrammers. End users with sufficient expertise to define their own macros should have little trouble tailoring their interfaces using MIKE.[3] The interface definition is stored in a profile file. Because MIKE does not evaluate any of the application's actions until after the complete command has been parsed, it is possible to provide the same kind of rubout facilities found in most command-line interpreters. Rubout undoes the effect of the last command selection or operand input. The echoes and menus are all restored to what they were previously. Subsequent invocations of rubout can continue backward until the entire command has been removed. The entire command can be removed in one event with the Cancel command.

MIKE's support of macros-by-example requirements

Because of the nature of MIKE's dialogue model, the logging of complete semantic commands is relatively straightforward. The presentation of a command in the macro body is a matter of using the command echo that MIKE already generates automatically. Identification of arguments that should be parameterized is easy because the action/argument relationship is clearly defined. Once a macro has been created, it is a command with a set of parameters. As will be shown later, the macro parameter types are implicitly determined while creating the macro. From MIKE's point of view, in terms of the dialogue model, a macro is the same as a primitive application procedure. Once a macro is defined, it will by default appear in the global menu, and the interface editor can tailor its interactive presentation using all the techniques available for primitive commands.

MIKE's dialogue model and architecture then provide five of the six facilities required of a UIMS to support macros by example. MIKE alone, however, will not solve the UNDO problem, which must be solved for the macro editing facility. This facility requires the STUF data management facility and the cursors package discussed in the next section.

Implementation of macros by example

Before discussing how macros by example have actually been added to MIKE, we show in Figure 4 the overall architecture of an application. In this architecture the graphics package GPW forms the interface between the software and the interactive display and input devices. This is a windowing package which, like many others, supports overlapping windows. As such, it queues input

Authorized licensed use limited to: Brigham Young University. Downloaded on January 29, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

events for MIKE as well as window redraw events. MIKE parses the input events on the basis of the interface description found in the profile file.

As discussed earlier, the profile file is created by the interface editor and defines the entire user interface. The only time that MIKE itself modifies the profile file is when it stores or modifies macros. MIKE calls on the application functions and procedures that have been defined for it whenever a complete command has been recognized. The roles of STUF, cursors, and change logging are discussed below in conjunction with the implementation of UNDO.

Macro definition

In a submenu of the main menu, MIKE provides a set of macro definition and manipulation commands. Most of MIKE treats these special commands as application commands, allowing their user interface to be tailored by the interface editor. These commands are divided into recording commands and editing commands. The recording commands are defined as follows:

Create(Name: String)
Creates a new macro. The name is actually a menu pathname to place the macro in MIKE's tree of menus.

NewParameter(Name: String)
Adds a new parameter to the open macro.

StartRecord
Starts recording the macro example.

StopRecord
Stops recording the macro example.

CancelRecord
Terminates recording without saving the macro.

When the user creates a macro, a special macro window opens and is kept on top of all other windows on the screen. This window can be moved around or resized so as not to interfere with the application while the macro is being demonstrated. With the NewParameter command the user can define parameters for the macro by name. Initially these parameters are typeless.

To create a macro the user invokes the StartRecord command and begins providing an example of how the macro should behave. Again, most of MIKE does not know that commands are being saved in a macro body. The interface proceeds as normal until an application command is to be invoked. Then the command expression is passed to the routine that actually does the executing. This routine checks to see if macro recording is turned on and, if it is, calls the appropriate routines to log the command into the macro body. Then the routine calls the application to have the operation actually performed.

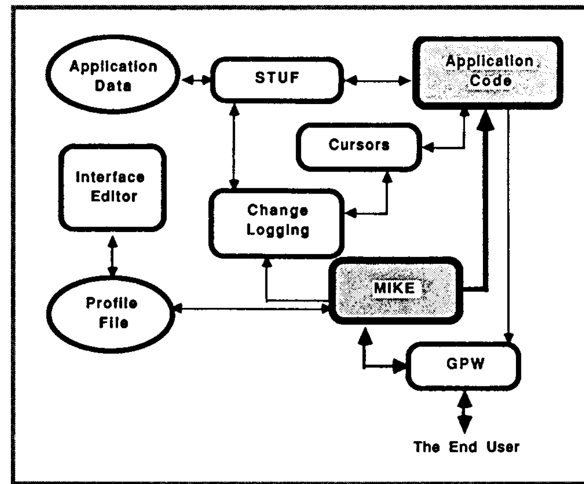


Figure 4. System architecture.

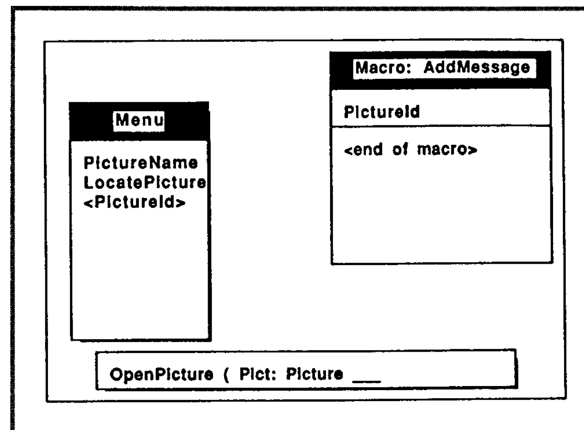


Figure 5. Creating a macro.

When commands are logged into the body, the macro window is also updated to reflect the addition of the command. Since MIKE's command-echo mechanism is driven by the same internal command representation used in executing the command, it is a simple matter to formulate the command presentation in the macro window using the same algorithms and interface information.

Macro definition proceeds as the user expresses the commands in the user interface, as is normally done. When initially defined, the parameters are typeless. Because of this, MIKE places the name of the parameter in angle brackets (⟨ParmName⟩) in every menu associated with an argument type. To reference a parameter, the user selects it from the menu. A schematic view of how this might be done is shown in Figure 5. At the

Authorized licensed use limited to: Brigham Young University. Downloaded on January 29, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

stage illustrated in the figure, the parameter is assigned the type of argument for which it will be used.

An example value for the parameter is required for the demonstration of the macro to continue to function properly. A prompt is issued for the example value, and the input parsing proceeds normally. When the command that used the parameter is in fact logged, the value expressed for the parameter is stored with the parameter's definition, and the command is logged into the macro body with a parameter reference rather than the example value given for the parameter. Once a parameter receives a type and an example value, its name will appear only in the menu associated with that type. Any subsequent use of the parameter in the macro body will select the parameter's name, and its attached example value will be used. This approach is not as sophisticated as Halbert's generalized data descriptions, but it is rather direct and easily learned, and does not intrude much into the normal user interface.

Macro editing

Making mistakes while demonstrating a macro body is very easy, and it is frequently desirable to go back and edit an existing macro. The commands provided for editing are as follows:

DisplayAMacro(MacroName: String)
This will open the specified macro in the macro window so that it can be edited.

Cut
Cut the current statement out of the macro.

Paste
Paste a statement in at the current position.

UpArrow
Move up one command.

DownArrow
Move down one command.

GoTo(Where: Point)
Move the current command position to the one pointed at by Where.

The DisplayAMacro command opens an existing macro for editing. To edit a macro, however, the user must have example values for its parameters. The sample parameter values specified at macro definition time are not saved, because they were specific to that example context. DisplayAMacro recursively calls the MIKE parser to obtain the example values from the end user. Remember that all editing is done in demonstration mode. Within the macro window one of the macro commands is highlighted as the current statement position. This statement is the next command in the macro to be executed. The Cut command deletes the current statement from the macro and saves it. The Paste command inserts the saved command immediately in front of the current statement and makes it the new current statement. Obviously, a richer set of editing commands could be provided, but those shown above are sufficient.

Positioning the current statement pointer is the most difficult part of macro editing while in demonstration mode, because the state of the application must coincide with this current position. Selecting the DownArrow command (normally bound to the down arrow key on the keyboard) has the same effect as single stepping through the macro in a debugger. MIKE takes the current statement, passes it to the execution routines for evaluation, and moves to the next statement. Selecting UpArrow, however, requires that the statement immediately in front of the current one be undone. The exact process for undoing is discussed below.

The GoTo command is a quick way to move through the macro body by pointing at a statement in the macro window. Normally, GoTo is bound to the macro window itself, so that selecting a point in the macro window automatically invokes GoTo using that point. GoTo is implemented by moving up or down in the macro body until the selected command is reached. Therefore, it must be possible to undo an arbitrary number of commands.

Note that these movements within the body of a macro are possible at any time during original demonstration or later editing. The user could, for example, back up to a statement by selecting it, demonstrate two or three commands to be executed before the selected statement, and then move down to the end of the macro body and continue on. Invoking any application command while in macro edit mode will insert the demonstrated command immediately in front of the currently selected command in the macro body. This is not an off-line process; all of it can be done as a natural part of the user interface.

As discussed above, a simple straight-line macro is not very interesting. To support IF and WHILE control structures, we used the approach found in Pygmalion.[16] When the user gets to the point where an IF test is desired, the IF command can be selected from the macro menu. Since MIKE knows about data types, it then presents a menu of all application functions that return Booleans, and enters a Boolean expression as the predicate. This expression is then evaluated and, depending on the result, the True or False branch is chosen, where the next logged commands are placed.

To leave the IF, the user selects a statement beyond the end of the IF. (MIKE always maintains a special selectable dummy statement at the end of every block.) A difficulty arises when the user wants to specify the other branch of the IF. Doing this requires that the macro be saved and a new set of example parameters that produce the opposite predicate result be specified. The user then

Authorized licensed use limited to: Brigham Young University. Downloaded on January 29, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

moves down to the IF and into the other branch, where statements can again be expressed by example. The user can create WHILE loops in a similar fashion. The only difficulty is that when a statement inside the body of the loop is selected, there is ambiguity as to which iteration of the loop has been selected as the example. Because DownArrow always moves to the next statement in execution order, moving through multiple iterations of the loop is possible, but still somewhat awkward.

Implementation of UNDO

There are several possible ways to provide a facility for undoing the results of some action.[17] One of these is to log all semantic actions and then replay them up to a point just prior to the portion to be undone. This is easily done in a UIMS possessing access to all the semantic calls that have been made, but it is not nearly fast enough for use in macro body cursor movement. This approach also requires the ability to restore the application state existing before the beginning of the log.

A second and better approach is to have inverse operations for every semantic action. In fact, any good user interface should include mechanisms for reversing any operation. The difficulty lies in the fact that such inverses frequently depend on the end user's intelligence. For example, if the user draws a line, he or she can then delete it. What is required, however, is that the end user, knowing that delete is the reverse of draw, will reselect the line to be deleted. Expecting such intelligence in the end user is not unreasonable. Expecting such intelligence in the UIMS, however, can lead to problems.

An automatic inverse facility would need to have more semantic knowledge about exactly how to select the correct line for deletion, and would need to know that select followed by delete is the appropriate inverse for drawing a line. Such actions are obvious to end users but rather messy to encode for the automatic handling of all possible cases. Adding such semantic knowledge to MIKE's understanding of commands is entirely possible, but there are some subtleties in expressing invertibility that we have not completely explored.

A third approach—the one MIKE uses—is to monitor the data structures of the application itself and undo any changes to them. This is the approach used in many database management systems because the DBMS has control over all changes to the data. Regardless of what an application may do, the DBMS has final control, because ultimately changes to the database are the only real side effects.

STUF and undoing application data changes

Our research into editing templates[18] and other automatic data display and editing tools[19,20] has led us to develop STUF (STrUctured Files) as a data management system. STUF manages its structures in main memory and supports records, unions, and arrays of varying length in a form easily loaded to and from secondary storage. STUF provides the facilities of a database management system, while allowing the efficiency of accessing and manipulating data using normal data typing facilities. Like a DBMS, STUF has knowledge of all modifications to its data and can log sufficient information to allow any such operations to be undone. The details of STUF are discussed elsewhere.[21] What is important is that any changes to application data managed by STUF can be undone.

Undoing other side effects

In addition to the application's database there is other information that must be undone. In most interactive applications, there are "current data objects" that control the browsing or editing operations. Such objects might be the currently selected window, the text insertion point, or the currently open file. Such data references are called *cursors*, and they are the data references commonly associated with graphics or text cursors. Such cursors are integer numbers or pointers and are managed by the cursors package.

The cursors package monitors all changes to cursors and thus can log them. Cursors was originally implemented to support interrelationships and control updates between windows. For example, one window may have a list of files with one of them being the current file. Another window may show the contents of the current file. The current file then is a cursor that affects both windows. Any change to the current file must cause an update to both windows.

The final set of information to be undone is the graphics display itself. This UNDO is performed by notifying GPW that all of the screen has been damaged. The notification, in turn, generates events to cause the application to redraw all visible windows. The redraw capability is already necessary to support window repair. The entire screen must be damaged because MIKE does not know what changes to the screen have been made by the command being undone. On the VAXstation, redraw is easily done with enough speed for UNDO.

To perform UNDO in MIKE the following steps are taken:

1. Before an application command is invoked, MIKE calls the change-logging facility to place a mark on the change stack.
2. MIKE invokes the application's command procedure and functions.
3. The application then performs whatever operations it must. In the process, every change to the cursors and STUF files will be logged by calls to change logging. All information saved by the application between calls to semantic routines must be

Authorized licensed use limited to: Brigham Young University. Downloaded on January 29, 2009 at 15:11 from IEEE Xplore. Restrictions apply.

stored in a cursor or STUF. The change-logging software has been generalized so that other data-storage models can also use it to log their changes, but to date STUF and cursors have been sufficient for us.

4. When an UNDO is to be performed, MIKE calls change logging and requests that all changes down to the most recent mark be undone. Change logging handles this via calls to STUF and cursors.
5. After the application data has been restored, MIKE has GPW generate the necessary redraw events, which MIKE then refers to the appropriate application routines.

Summary and conclusions

We believe that the future of user interface management systems lies in their ability to provide features beyond those normally found in hand-coded applications. The unique semantics-based dialogue model of MIKE has allowed us to create a macro facility in which macros are specified by example. Such a facility allowing end users to extend their interfaces is thus available automatically to all applications built with MIKE and its sibling software packages.

There are still some issues to be resolved. The macro facility does not have local variables. Along with these, it would also be of value to include a facility like Halbert's set iterators as an alternative to the WHILE loop.

The current approach does not have the same ease of expressing visual behavior found in Peridot.[22] Note, however, that in its present form Peridot is not a facility allowing an end user to extend the interface: Macros by example still have a problem in operand references. If, for example, the user selected a file from a desktop window, the value received would probably be an internal pointer to the file. Such internal pointers are not necessarily valid in future invocations of the macro. What is really needed is the file name.

This example illustrates the dichotomy between interactive specification and specifications to be preserved over time. Macros by example still do not fully resolve such problems. They do, however, provide an important new capability for user interface management systems.■

References

1. D.C. Halbert, *Programming by Example*, Xerox Office Systems Division, Palo Alto, Calif., 1984.
2. B.A. Myers, "Visual Programming, Programming by Example and Program Visualizations: A Taxonomy," *Proc. SIGCHI 86: Human Factors in Computing Systems*, ACM, New York, 1986, pp. 59-66.
3. D.R. Olsen, "Mike: the Menu Interaction Kontrol Environment," *ACM Trans. Graphics*, Oct. 1986, pp. 318-344.
4. H. Lieberman and C. Hewitt, "A Session with Tinker: Interleaving Program Testing with Program Writing," *Conf. Record 1980 LISP Conf.*, Stanford Univ. Press, Stanford, Calif., 1980.
5. M. Green, "The University of Alberta User Interface Management System," *Computer Graphics* (Proc. SIGGRAPH 85), July 1985, pp. 205-214.
6. D.R. Olsen and E.P. Dempsey, "SYNGRAPH: A Graphic User Interface Generator," *Computer Graphics* (Proc. SIGGRAPH 83), July 1983, pp. 43-50.
7. R.J.K. Jacob, "A State Transition Diagram Language for Visual Programming," *Computer*, Aug. 1985, pp. 51-59.
8. W.M. Newman, "A System for Interactive Graphical Programming," *SJCC 1968*, Thompson Books, Washington, D.C., 1968, pp. 47-54.
9. D.R. Olsen, "Push-Down Automata for User Interface Management," *ACM Trans. Graphics*, July 1984, pp. 177-203.
10. D.J. Kasik, "A User Interface Management System," *Computer Graphics* (Proc. SIGGRAPH 82), July 1982, pp. 99-106.
11. W. Buxton et al., "Towards a Comprehensive User Interface Management System," *Computer Graphics* (Proc. SIGGRAPH 83), July 1983, pp. 35-42.
12. J.L. Sibert, W.D. Hurley, and T.W. Bleser, "An Object Oriented User Interface Management System," *Computer Graphics* (Proc. SIGGRAPH 86), Aug. 1986, pp. 259-268.
13. P.P. Tanner et al., "A Multitasking Switchboard Approach to User Interface Management," *Computer Graphics* (Proc. SIGGRAPH 86), Aug. 1986, pp. 241-248.
14. D.S.H. Rosenthal, "Managing Graphical Resources," *Computer Graphics*, Jan. 1983, pp. 38-45.
15. W.M. Newman and R.F. Sproull, *Principles of Interactive Computer Graphics*, McGraw-Hill, Hightstown, N.J., 1979.
16. D.C. Smith, *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*, doctoral dissertation, Stanford Univ., Stanford, Calif., 1975.
17. J.E. Archer, R. Conway, and F.B. Schneider, "User Recovery and Reversal in Interactive Systems," *ACM Trans. Programming Languages and Systems*, Jan. 1984, pp. 1-19.
18. D.R. Olsen, "Editing Templates: A User Interface Generation Tool," *CG&A*, Nov. 1986, pp. 40-45.
19. P.F. MacKay, *Fixed Forms Display of Arbitrary Data Structures*, master's thesis, Brigham Young Univ., Provo, Utah, 1986.
20. E. Scott, *Generalized Parsing and Display of Textually Represented Data Structures*, master's thesis, Brigham Young Univ., Provo, Utah, 1985.
21. D.R. Olsen, "STUF: Structured Files for Interactive Programs," tech. report, Computer Science Dept., Brigham Young Univ., Provo, Utah, 1986.
22. B.A. Myers and W. Buxton, "Creating Highly Interactive and Graphical User Interfaces by Demonstration," *Computer Graphics* (Proc. SIGGRAPH 86), Aug. 1986, pp. 249-258.

Dan R. Olsen, Jr., has been an associate professor of computer science at Brigham Young University for the past three years. Previously he was on the computer science faculty at Arizona State University. His primary interests are software tools for generating user interfaces, and interactive programming languages.

Olsen received his BS and MS degrees from Brigham Young University, and his PhD from the University of Pennsylvania.

Olsen can be contacted at the Computer Science Department, Brigham Young University, Provo, UT 84602.

John R. Dance is a software engineer at Apple Computer in the Development Systems Group. His interests are in programming environments, tools, and prototyping. He received his BS and MS degrees in computer science from Brigham Young University in 1984 and 1986.

Dance can be contacted at Apple Computer, Inc., 20525 Mariani Ave., Cupertino, CA 95014.

Authorized licensed use limited to: Brigham Young University. Downloaded on January 29, 2009 at 15:11 from IEEE Xplore. Restrictions apply.