SQUISH: A GRAPHICAL SHELL FOR UNIX†

Tyson R. Henry
Scott E. Hudson
Department of Computer Science,
University of Arizona,
Tucson, Arizona 85721

ABSTRACT

Because of its power and flexibility, the UNIX operating system has become an almost standard tool for computer science research. However, the UNIX user interface has been criticized as being cryptic, hard to learn, and hard to use, particularly for novice users This paper presents a graphical shell interface for UNIX that preserves the power and flexibility of UNIX, while improving its ease of use.

KEYWORDS: Graphical User Interface, UIMS, Direct Manipulation, Declarative Programming

1. Introduction

Because of its power and flexibility, the UNIX operating system has become an almost standard tool for computer science research. However, the UNIX user interface has been criticized as being cryptic, hard to learn, and hard to use [7], particularly for novice users (a study of the UNIX user interface can be found in [2]). This paper presents a graphical shell interface for UNIX that improves its ease of use, while preserving its power and flexibility.

The Squish system (a Simple Quick User Interface SHell) is a graphical shell for UNIX. UNIX commands are represented by graphical interfaces instead of cryptic names. These interfaces graphically present all the options and required arguments to the command. They allow command instantiation to be performed in a direct manipulation [5, 10, 11] manner. The Squish system also allows commands to be composed into more powerful commands by forming pipelines (a pipeline is a series of commands in which the output of one command is used as the input of another).

There are hundreds of UNIX commands. Creating graphical user interfaces for all UNIX commands would be very difficult. By providing a simple but flexible method for specifying user interfaces, the Squish system provides access to arbitrary subsets of UNIX commands--preferably the most common and most difficult commands. Graphical interfaces for commands are specified in the Squish specification language.

†UNIX is a trademark of AT&T Bell Laboratories. This work was supported by the National Science Foundation under grant IRI-8702784.

In the Squish framework, commands--presented in plain English--are displayed in a menu. Each command has a corresponding Squish interface. In a typical Squish interface, most, if not all, options and arguments are also clearly presented in plain English. Options are selected by using a pointing device to directly manipulating simulated buttons and other graphical interaction objects. pushing simulated buttons. Consequently, the user need not recall all the different command names and all of their options.

This represents an improvement for both the novice and the expert users. The novice does not need to learn the cryptic command names and the equally cryptic option names of the most commonly used commands. The expert, on the other hand, can use the Squish framework for seldomly used commands--the ones he or she never seems to remember.

Clearly an expert user would become frustrated by the restriction of using only a subset of commands and as the novice becomes more familiar with UNIX, they will also find the subset restrictive. Unlike some graphical shells (for example [6]), the Squish system alleviates this problem by allowing access to the complete set of UNIX commands. While graphical interface to new commands can always be added, Squish provides a textual interface in addition to the graphical one. The UNIX command created from the graphical interface is displayed in an editable window. This provides a bridge between the two interfaces allowing the expert user to quickly check the validity of the command while allowing the novice to learn the textual command.

In order to provide an interface specification language powerful enough to describe any UNIX command, yet easy enough to be practical, new UIMS techniques have been developed. In particular, the system makes extensive use of derived data. The graphical presentation of the interface is controlled by a set of equations that state how graphical images can be derived from the internal data values or state of various interaction objects. While the Squish specification language has been developed to enable a graphical shell for UNIX, the concepts developed here can clearly be applied to a more general UIMS.

2. The Squish User Interface

Typical shell commands in a UNIX system consist of a series of individual commands. These commands are connected by *pipes* so that the output from one command becomes the input for the next command. Such a sequence of commands is called a *pipeline*. The primary purpose of the Squish system is to create pipelines. Rather than use tex-

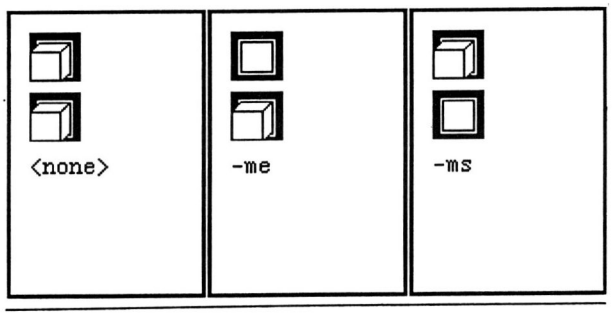

Figure 1. Simple Button Interface

tual syntax for pipelines, the Squish system uses graphical syntax. Components of a pipeline are placed in windows from left to right. Each such window implements the interface to the options and arguments of a single command or tool.

This section uses examples to introduce the concepts behind the Squish system. These examples illustrate how the interface to a single command can be constructed. The next section presents the composition of individual commands into pipelines.

An individual Squish interface consists of a series of interaction objects in a window. These objects are currently declared with a simple textual specification language. Future versions of the system will use a graphical specification technique. Figure 1 shows three views of a very simple Squish interface. It consists of two mutually exclusive buttons and a text item. Figure 2 shows the textual specification for this interface.

In the Squish system, all interaction objects are typed. A system supplied library defines the set of interaction object types. Object types definitions can be added to the library (see section 4 for a detailed description). This sample specification contains three objects -- two objects of type button and one object of type textbox. The implementation of an object type provides the basic behavior of an object. It defines the basic appearance of the object and the way it responds to input events. However, most of the details of the appearance and behavior of the object can be customized by defining *attributes* of the object.

An attribute is simply a value attached to the object. Certain attributes have specific predefined meanings and are used by the object type to implement the behavior of the object. For example, the attributes x and y (attached to each object shown in Figure 2) control the placement of the object on the screen. Others attributes, such as options.macro_str, may be introduced by the user as needed. Attributes may either be *intrinsic* or *derived*. An intrinsic attribute simply stores a value and is declared with an initial value such as:

```
x := 10;
```

A derived attribute, on the other hand, is computed in terms of other attribute values using a defining expression or *derivation rule* (introduced with a "=" instead of a ":="). For example, the ms_button.y attribute in Figure 2 is defined in terms of the me_button.y and me_button.h attributes (representing the y position and the height of the me_button object, respectively). This is done using the derivation rule:

```
y = me_button.y + me_button.h + 5;.
```

This rule states that whenever either of the attributes involved change, the value of ms_button.y will also change,

```
button me_button
        x := 10;
        y := 10;
        result = if (highlighted) then "-me" else "";
        action := { ms_button.highlighted := 0; };
end me_button;

button ms_button
        x = me_button.x;
        y = me_button.y + me_button.h + 5;
        result = if (highlighted) then "-ms" else "";
        action := { me_button.highlighted := 0; };
end ms_button;

textbox options
        x = ms_button.x;
        y = ms_button.y + ms_button.h + 5;
        macro_str = strcat(me_button.result, ms_button.result);
        string = if macro_str = "" then "<none>" else macro_str;
end options;
```

Figure 2. Specification for Simple Button Example

and the graphical image presented to the user will be updated accordingly. This automatic update of the graphical image is the key to the power and flexibility of the Squish system.

Each object type provides a set of predefined attributes that have special meaning for the type. All types contain the following attributes:

x	x position of the object.
y	y position of the object.
h	height of the object.
w	width of the object.
visible	controls whether object appears on screen or is invisible.
enabled	controls whether object accepts or ignores input events.
highlighted	controls appearance of the object in a type specific way.
result	provides the *result string* for the object (see section 3).

In addition, other predefined attributes may be associated with particular object types. For example the action attribute is predefined for the button type. This attribute provides a sequence of statements to be executed when the button is pressed. In addition, the button type provides an additional series of attributes that control the button's image when it is in the four different states indicated by the enabled and highlighted attributes. Note that these attributes all have appropriate default values and normally are not explicitly specified by the interface designer.

A more careful examination of Figure 2 reveals how derived data is used to implement most of the interface's functionality. Note that options.string is defined indirectly in terms of the result attribute of the two buttons. These attributes are in turn derived from the highlighted attribute. Finally, the value of highlighted is controlled by the object type implementation on the basis of user actions. This allows the interface to respond automatically to the three possible states of the interface as shown in Figure 1. The derivation rules given have established a direct connection between user actions and the graphical appearance and

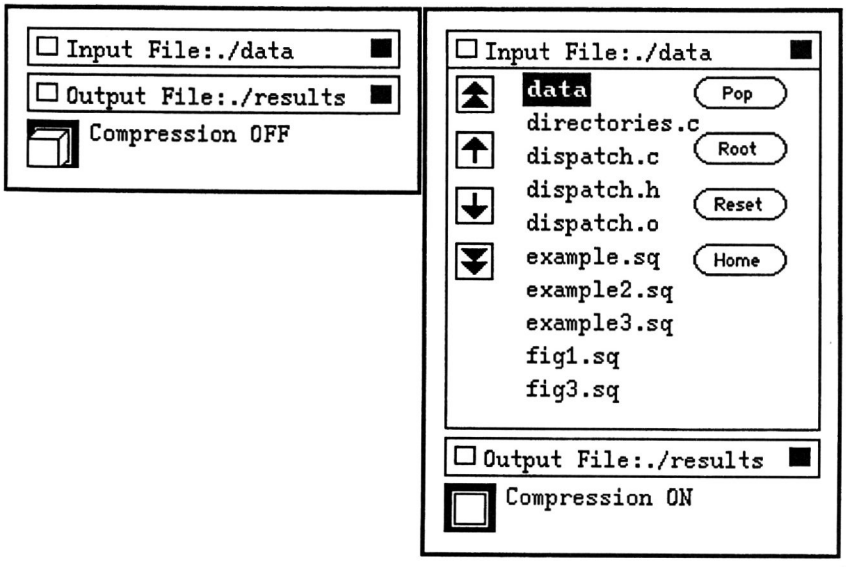

Figure 3. Example of Dynamic Presentation

behavior of various objects. Since the expressions used for derivation rules are programmable, it is possible to create quite arbitrary connections between user actions and the resulting graphical feedback. For example, in this program the string "<none>" is displayed when neither button is selected. A similar derivation rule might be attached to the enabled or visible attribute of another object to provide more sophisticated context dependent feedback. Rules allow lexical, syntactic, and semantic feedback to be handled in a single framework.

It is also important to note that Squish specifications are primarily declarative in nature. A set of equations are declared to hold true. The system is responsible for reevaluating attributes and updating graphics automatically to respond to any and all changes. This makes specifications much easier to write than a typical imperative specification. For example, Figure 3 shows two views of the interface whose specification is given in Figure 4.

The above example introduces a more complicated object type, a file_picker. This type allows the user to select a file by pointing at its name. An object of this type has two different appearances, open or closed. In the open state, it displays alternatives files from the currently selected directory, and allows the user to navigate through the hierarchical file system. In the closed state the object only displays the pathname of the selected file.

Because a declarative specification is used, it is easy to make the presentation of this interface adapt automatically. In this example, each object is positioned in terms of the objects above it. The below and left_of operators are simple shorthand for the obvious expressions involving y and h or x and w attributes. This specification arranges for all other objects to change their position automatically whenever either of the file_picker objects changes size (i.e., changes from closed to open) -- no explicit graphical update commands are needed. In addition, no case analysis is needed to handle all of the possible configuration of the system. This is particularly important since the number of configurations grows exponentially with the number of objects. If all the possible configurations had to be explicitly considered, it would be very difficult to build and maintain interfaces that modified their layout in a flexible, dynamic manner. With a declarative specification in the above form, all configurations

```
file_picker in_filename
        x := 10;
        y := 10;
        prompt := "Input File:";
end filename;

file_picker out_filename
        x = in_filename.x;
        y = below in_filename by 5;
        prompt := "Output File:";
end filename;

button cmp_btn
        x = out_filename.x;
        y = below out_filename by 5;
end compress;

textbox compress_lab
        x = left_of cmp_btn by 5;
        y = cmp_btn.y;
        string = if cmp_btn.highlighted
                        then "Compression ON"
                        else "Compression OFF";
end compress_lab;
```

Figure 4. Specification for Figure 3

```
button A
        result = if (highlighted) then "Button A" else "";
        ...
end A;

textbox A_label
        string := "Button A";
end A_label;

button B ... end B;

textbox B_label ... end B_label;

button C ... end C;

textbox C_label ... end C_label;

box all_buttons
        x := 20;
        y := 20;
        rows := 3;
        cols := 2;
        children A, A_label, B, B_label, C, C_label;
end all_buttons;
```

Figure 5. An Example Grouping Object

can be handled automatically.

In addition to shorthand notations such as the above, below, right_of, and left_of operators, the Squish system allows other shortcuts for common operations. The most important is the existence of objects types for managing the placement of other objects. For example, the type box arranges for any group of objects to be laid out in a rectangular array. Figure 5, shows a specification using a box object. The way this object type works is to supply derivation rules for the x and y attributes of the objects declared to be its *children*. These derivation rules are established so as to place the objects in rows and columns Since this placement is done with derivation rules instead of constant values, it is dynamic rather than static. That is, whenever a child object changes size all sibling objects adjust their position automatically to maintain the row and column structure. Using this same mechanism, a number of different object types can be created for *composing* objects in a hierarchical manner. Since arbitrary derivation rules may be used, the relationships between sibling objects can be quite complex.

3. The Graphical UNIX Shell

The last section illustrated how to specify graphical interfaces using the Squish specification language. The Squish system provides a framework for combining these individual graphical interfaces, each representing a UNIX command, into pipelines. Each interface is built from the instantiation of a group of Squish objects. The textual specification of an interface is kept in a single source file. Following UNIX convention, these files are called a *scripts*.

Currently, the user selects commands to be placed in the pipeline from a menu of commands. Frequently used commands are placed at the highest level of the menu. Future versions will replace this textual menu with a set of icons that can be dragged in placed (similar to the Macintosh Desktop). When a command is added to the pipeline, its Squish interface is displayed. The interfaces for each command in the pipeline are laid out on the screen from left to right in the same order as the commands in the pipeline.

Figure 6 shows the Squish pipeline framework. This sample pipeline consists of five commands that work together to format a document. The input file is specified by the first tool. The file is then piped to the **bib** bibliography formatting tool. The output from **bib** is then sent to a tool for performing keyword substitutions, then to a formatting tool, and finally to a printer selection tool. The user interacts with the interface for each tool separately choosing the correct options and arguments in order to build the complete pipeline.

Every Squish script contains a special object for specifying its window. The height and width attributes of this *window* object control the height and width of the window for the individual interface. The height and width of the entire Squish framework is derived from the heights and widths of all the individual windows.

Each interface window builds a result string (the attribute result from the window) that corresponds to its current configuration of options and arguments. The result strings from all the interfaces are combined using a pipe character (the attribute pipe_char from the window, usually "|" or ";"). This string represents the complete pipeline. In Figure 6, this string appears after the "Do It" button. Since this string is an attribute, it can be derived in an arbitrary way from other attributes using an arbitrary derivation rule. Displaying this string forms a bridge between the graphical and the textual representations. It allows the expert user to quickly verify the command while it provides the novice user a good means to learn UNIX commands.

The bottom window in Figure 6 is an instantiation of the UNIX shell. When the "Do It" button is pressed, the command string is sent to this window as if it had been typed in by the user. The results from executing the pipe are displayed in the shell window. Commands can be executed in the background by selecting the "In Background" button. The "Control Scrolling" button pipes the output into the UNIX **more** utility to prevent the output from scrolling by before the user can read it.

Using an explicit shell window allows the user to type in commands at will. This allows very simple commands (e.g. ls or cd) to be typed directly, and allows error messages and other feedback to be placed in context. In addition, it allows the user to use commands not included in the subset of commands built into Squish. The ability to type commands at will also allows pipelines to contain interactive commands. For example, a command might include an invocation of an editor to prepare a file--the editor would run within the shell window.

When the Squish system starts up, it reads the file *.Squishrc* in the user's home directory. This file contains a set of tuples relating Squish scripts to labels. The labels for each of these scripts will be placed in the first level menu. Other scripts are found in standard locations specified by a search path similar to the normal shell search path.

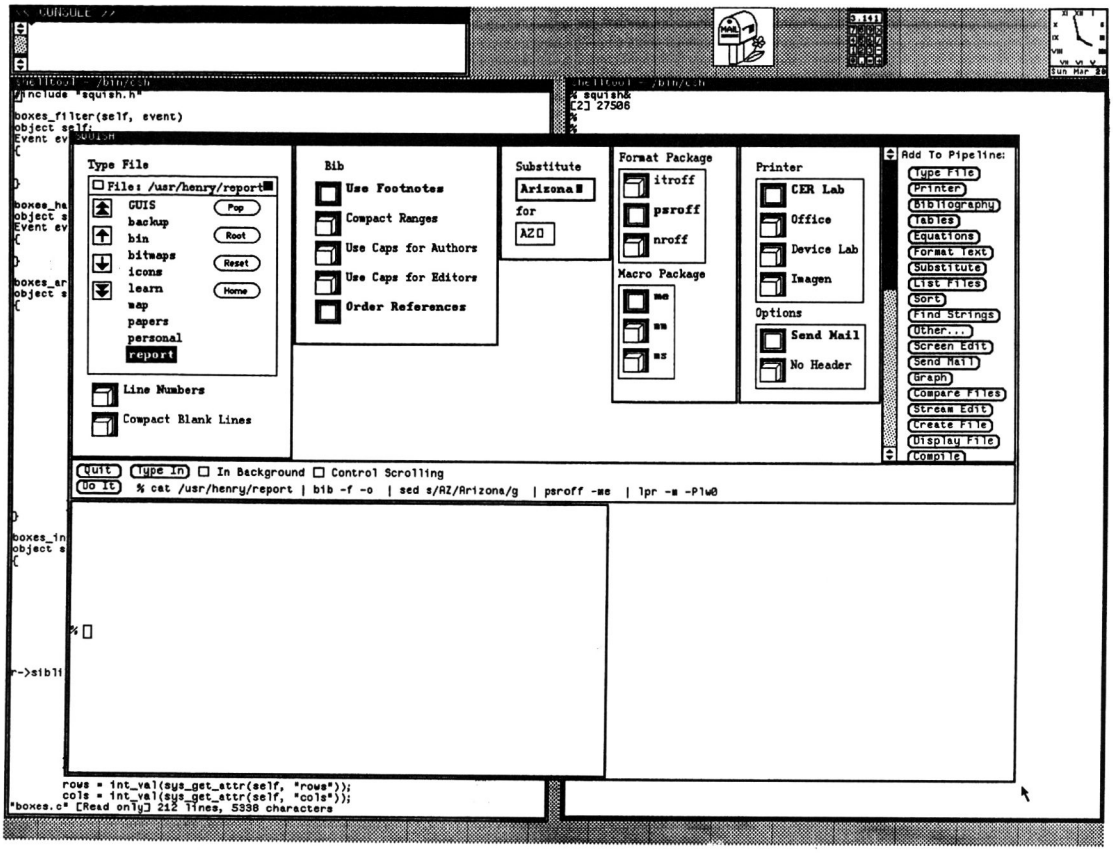

Figure 6. The Overall Squish Framework

4. Squish Internals

The Squish system is written in C making use of Lex and Yacc for parsing support. It currently runs on Sun workstations using the SunView window package. The system consists of approximately 6000 lines of C source code excluding code used to implement object types. Currently, the object type library for the system contains a small set of types (evaluators, buttons, text input/output, file picker, etc.). New types are being added based on experience gained writing Squish scripts.

The Squish system can be easily extended with new types. Type definitions use four procedures written in C. These procedures include: a filter procedure to decide which input events to accept, an event handler to act upon accepted input events, an artist procedure to display the object on the screen, and an initialization procedure. The procedures are bundled with the name of the type and a list of predefined attributes with initial values or derivation rules. These items are placed in a table of object types for use by the Squish system.

The primary job of the procedure implementing a type is to handle inputs and to redraw the associated interaction object. When an input event arrives from the host window system it is first passed to a series of *binding* objects. These binding objects each accept a certain class of events and implement a particular event binding protocol. Events that are not accepted by one binding object are passed to the next until the event is consumed. A particular event binding object will attempt to deliver the input event to one or more interaction objects on the basis of its event binding protocol. Currently there are two different event binding protocols in the system.

The first event binding protocol handles textual input. Keyboard related events are bound using a currently selected object paradigm. Under this paradigm, there is a single currently selected text object that receives all keyboard events. Routines are available for changing the current object. This binding protocol implements a *click-to-type* style interface for text input objects.

The second protocol is positional. Events bound under this protocol are sent to objects that are *under* the cursor. The positional protocol uses a prioritized list of objects that is ordered in front to back drawing order. Events are first sent to the top-most object. If this object rejects the event, it is passed to the next object, and so forth.

When an attempt is made to bind an event to an object using the positional binding protocol, several tests are performed to determine if the object should be given the event. Each object maintains a rectangle indicating its area of interest and a mask indicating which input event types it should receive. If the event is within the area and matches the event mask, it is passed to the object's filter procedure. This procedure examines the attributes of the object in order to implement context dependent rejection of certain events.

If the filter procedure does not reject the event, it is passed to the object's event handler procedure. This procedure acts upon the event based on the event type and the state of the object as expressed by its attributes. The event handler also has the option of consuming or rejecting the event--rejected events are passed to the next object in the object binding list.

In addition to other actions, a typical event handler updates the graphical image associated with the object. To do this, it modifies attributes of the object that control its appearance, then flags the object as *dirty*. The existence of dirty objects causes the system to redraw the contents of one or more windows. This is done by calling the artist procedure associated with each object in the window in a back to front order. The artist is responsible for drawing the graphical image of the object based on the values of its attributes. In order to improve the appearance and speed of window redraw, windows are first drawn in an off-screen bitmap, then placed on the screen with a single operation. Although this approach of total redraw after each change is potentially wasteful, our experience shows that for workstations such as the Sun, redraw performance is more than adequate for most interfaces. Since changing one attribute may cause many others to be reevaluated, this approach also avoids extensive analysis to determine the exact screen area modified. However, it should be noted that this approach would not be acceptable if the host window package did not support drawing on off-screen bitmaps, or some other form of double buffering. For example, the X window system as presented in [9] does not support double buffering of any kind.

5. The Squish Specification Language

Section 2 informally introduced the Squish textual specification language; this sections presents a more detailed description. As presented in section 2, a Squish script consists of a series of object declarations. Each object declaration is introduced with a type name followed by an object name and a series of attribute definitions. Attribute definitions for intrinsic attributes are of the form:

<attr_name> := <constant_expr>;

Definitions for derived attributes have the form:

<attr_name> = <expr>;

Expressions have a familiar syntax. This syntax includes all the arithmetic operators found in Pascal plus a conditional (if-then-else) operator and several shorthand operators such as below and left_of. Finally, expressions may call arbitrary C functions. These functions must be installed in a table of external functions and linked with the Squish system.

Attribute values in Squish come from a fixed set of types including integer, string, bitmap, and code. The type of a value may change dynamically. Type checking is performed at runtime using tagged values. Bitmap values represent a rectangular array of pixels and can be used to customize the images used for various types. For example, button objects have four bitmap valued attributes that provide the graphical image of the object when it is enable and highlighted, disabled and unhighlighted, etc. By redefining these attributes, it is possible to change the images presented and hence the apparent action of the button object. For instance, a two position toggle switch or a simple push button could be both be implemented simply by supplying new bitmap attribute values.

Attributes of type code are introduced with braces ("{" and "}"). The code within these braces has a Pascal like syntax, and can include the following types of statments:

Assignment	using <object>.<attr> := <expr>;
Conditional	using if-then-else or if-then
Loop	using while or repeat-until loops
Procedure Call	to arbitrary C routines

Code valued attributes are typically used with action attributes. The event handling routine for an object type can request that the system execute a code value by passing the code to the interpreter.

Although the derivation rules used by the system are quite expressive, they do have limitations. Most importantly, they do not allow circular definitions--attributes cannot be directly or indirectly defined in terms of themselves. In situations where objects are mutually dependent, such as the mutually exclusive buttons in Figure 1, action attributes can be used to implement the required feedback. However, use of action attributes to implement feedback is the exception rather than the rule. Squish derivation rules are assumed to be applicative. In particular, the ordering of any side-effects occurring in external C routines is not guaranteed and is in general unpredictable.

Currently, the system uses a very simple attribute evaluation algorithm. An integer time-stamp is kept with each attribute that indicates when it was last evaluated. Whenever an assignment to an attribute is made, the *current time* is incremented. When an attribute value is requested, its time-stamp is checked. If the time-stamp is older than the current time, the attribute is recursively reevaluated and its time-stamp is reset to the current time. This algorithm performs considerably more work than necessary since it reevaluates attributes that could not have changed value. Unfortunately, the standard optimal algorithm for attribute evaluation [8] cannot be used in this framework since it can only handle attributed trees and not the arbitrary graphs found here. An algorithm capable of handling this class of problems and only updates the optimal set of attributes is discussed in [3, 4]. This algorithm will be used in later versions of the system. An alternate efficient algorithm for this type of attribute evaluation is discussed in [1].

6. Conclusions

This paper has introduced the Squish graphical shell for UNIX. The system offers advantages for both the expert and novice user, while preserving the power and flexibility of the UNIX system. In order to implement the system, new UIMS techniques have been developed. In particular, the system makes extensive use of declarative specifications in the form of data derivation rules. This use of derived data makes flexible and dynamic presentations easier to construct by placing most of the burden of graphical update on the system rather than the interface designer.

References

1. B. Alpern, A. Carle, B. Rosen, P. Sweeney and K. Zadeck, Incremental Evaluation of Attributed Graphs, *IBM Research Report RC 13205*, October 1987.
2. L. De Leon, W. G. Harris and M. Evens, Is There Really Trouble With UNIX, *Proceedings of CHI '83*, Boston, Dec. 1983, 125-129.
3. S. E. Hudson and R. King, A Generator of Direct Manipulation Office Systems, *ACM Transactions on Office Information Systems 4*(April 1986), 132-163.

4. S. E. Hudson, Incremental Attribute Evaluation: An Algorithm for Lazy Evaluation in Graphs, *University of Arizona Technical Report*, Aug. 1987. Tech. Rep. 87-20.

5. E. L. Hutchins, J. D. Hollan and D. A. Norman, Direct Manipulation Interfaces, in *User Centered Systems Design*, D. A. Norman and S. W. Draper (ed.), Lawrence Erlboum Associates, Hillsdale, New Jersey, 1986, 87-124.

6. B. Jovanovic and J. D. Foley, A Simple Interface to UNIX, *George Washington University Tech. Report GWU-IIST-86-23*, 1986.

7. D. A. Norman, The Trouble With UNIX, *Datamation*, Nov. 1981, 139-150.

8. T. Reps, T. Teitelbaum and A. Demers, Incremental Context-Dependent Analysis for Language-Based Editors, *ACM Trans. Prog. Lang. and Systems 5*(July 1983), 449-477.

9. R. W. Scheifler and J. Gettys, The X Window System, *ACM Transactions on Graphics 5*(April 1986), 79-109.

10. B. Shneiderman, The Future of Interactive Systems and the Emergence of Direct Manipulation, *Behaviour and Information Technology 1*(1982), 237-256.

11. B. Shneiderman, Direct Manipulation: A Step Beyond Programming Languages, *Computer 16*(August 1983), 57-69.