1988

An Approach to Support Automatic Generation of User Interfaces

Prasun Dewan

Marvin Solomon

Report Number:
88-761

AN APPROACH TO SUPPORT AUTOMATIC GENERATION OF USER INTERFACES

Prasun Dewan
Marvin Solomon

CSD TR-761
April 1988

An Approach to Support Automatic Generation of User Interfaces

PRASUN DEWAN
Purdue University
and
MARVIN SOLOMON
University of Wisconsin

In traditional interactive programming environment, each application individually manages its interaction with the human user. The result is duplication of effort in implementing user interface code and non-uniform—hence confusing—input conventions. This paper presents an approach to support automatic generation of user interfaces in environments based on algebraic languages.

The approach supports the editing model of interaction, which allows a user to view all applications as data that can be edited. An application interacts with a user by submitting variables (of arbitrary types) to a *dialogue manager*, which displays their *presentations* to the user, and offers "type-directed editing" of these presentations. Applications and dialogue manager communicate through a protocol that allows a presentation to be kept consistent with the variable it displays.

A particular implementation of the approach, called *Dost*, has been constructed for the Xerox Development Environment and the Mesa programming language. Dost is used as a concrete example to describe the editing model, the primitives to support it, and our preliminary experience with these primitives. The approach is compared with related work, its shortcomings are discussed, and suggestions for future work are made.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—user interfaces; D.2.6 [**Software Engineering**]: Programming Environments; D.3.3 [**Programming Languages**]: Language Constructs

General Terms: Design, Languages

Additional Key Words and Phrases: Editing, input/output, object-oriented systems, persistent data structures

1. INTRODUCTION

In traditional interactive programming environments, the responsibility of providing the user interface of an interactive program is divided among the program itself, subroutine libraries, the programming language, and the operating system. The interactive program, however, shoulders most of the burden of providing its user interface. This allocation of responsibility has at least three drawbacks:

First, implementation of user interfaces is *expensive*. Typically, an interactive program is concerned with scanning and parsing input, reporting errors, converting correct input into its internal representation, and displaying

This research was supported in part by the National Science Foundation under grant MCS-8105904 and by the Defense Advanced Research Projects Agency (DoD) under Naval Research Laboratory contract No. N00014-85-K-0788.
Authors' addresses: P. Dewan, Department of Computer Science, Purdue University, W. Lafayette, IN 47907; M. Solomon, Computer Sciences Department, University of Wisconsin—Madison, Madison, WI 53706.

results. The code to perform these tasks can be the major portion of an interactive program. A survey of commercial programs showed that display generation and management code constituted 40-60 percent of the source text of the programs sampled [39].

Second, different user interfaces offer *inconsistent* modes of interaction. Different interactive programs usually offer different ways to enter operations, and often the same operation is called by different names. For example, on a UNIX[1] workstation, the 'delete' operation might be invoked in a variety of ways depending on context: The user might delete a window by pointing to it with a mouse and selecting an operation from a pop-up menu; delete a file by typing 'rm' and the file name; delete a character from a file by invoking an editor, moving the cursor to the desired location, and typing 'x'; delete a process by typing 'kill' and the process identifier; and so on.

Third, most of the interfaces are *primitive* because they do not offer several 'friendly' features that are hard to implement. Examples of these features are menus and templates for input data, incremental feedback, operations to view information at various levels of detail, and operations to redo or undo other operations. Faced with the difficulty of implementing such features, an application programmer might well decide that the effort is not worthwhile, and settle for a more primitive interface.

These three problems can be corrected with automatic generation of user interfaces. This idea requires the design of an application-independent model of interaction together with an environment that supports the model.

Recent work in user interface design has suggested that *editing* can be used as a general model of interaction [3,4,7,9,24,26,34]. The model allows the user to view all applications as data that can be edited. We illustrate the model through Fraser's example of a directory manager that allows the user to manipulate a directory by editing its listing [7].

Figure 1 illustrates how a user may edit a UNIX-style listing of a directory. Figure 1(a) shows an initial listing of a directory containing two files. Changes to the directory listing are reflected in the directory itself. For instance, editing a name field changes the name of a file (Figure 1(b)), editing an access field modifies access rights to a file (Figure 1 (c)), and deletion of an entry causes the corresponding file to be destroyed (Figure 1(d)).

Interface presented by the directory manager to manipulate a directory is similar to the interface presented by a text editor to edit a text file. There is, however, an important difference. The directory manager cannot allow the user to make *arbitrary* changes to a directory listing. For instance, the user must not edit the *date* or *size* field of an entry, insert the character 'q' in an access field, or change or delete an entry without appropriate authorization. The first of these restrictions could be accommodated by an editor that supported "read-only" fields, but the second and third restrictions are intimately linked to the semantics of the the application (directory manipulation, in this case). In the following sections, we shall present a software environment that allows an application to delegate the mechanics of editing interaction to a generic tool, while retaining control over application-specific restrictions.

---

[1]UNIX is a registered trademark of AT&T Bell Laboratories

Access	*Links*	*Owner*	*Size*	*Date*	*Name*
drw-rw-r--	1	joe	512	Jun 23 1985	src
-rw-rw-r--	1	joe	111	Oct 13 1984	todo

(a) Initial Listing

Access	*Links*	*Owner*	*Size*	*Date*	*Name*
drw-rw-r--	1	joe	512	Jun 23 1985	src
-rw-rw-r--	1	joe	111	Oct 13 1984	done

(b) User Edits File Name

Access	*Links*	*Owner*	*Size*	*Date*	*Name*
drw-rw-r--	1	joe	512	Jun 23 1985	src
-rw-rw-rw-	1	joe	111	Oct 13 1984	done

(c) User Edits Access Field

Access	*Links*	*Owner*	*Size*	*Date*	*Name*
drw-rw-r--	1	joe	512	Jun 23 1985	src

(d) User Deletes File

Figure 1
Editing a Directory

It is easy to see how other applications may present editing interfaces. A 'process manager' can allow a user to edit a visual representation of the current processes to insert a process, delete it, or change its status. An interactive debugger can allow a user to control the execution of a particular process by editing a representation of its state. A 'printer manager' can allow a user to edit a representation of the printer queue to submit or cancel a print request. An executive may allow a user to enter commands by editing a representation of the history of previous commands. In general, any interactive application can present an editing interface by displaying a visual representation of its data, allowing the user to edit the representation in a syntactically and semantically consistent fashion, and reacting appropriately to a change in the representation.

The editing model is not only general, but also has the following pleasant properties:

- It has been successfully used by several applications such as text editors, spreadsheets, language-oriented editors [6, 12, 28, 41, 45, 46], form editors [31, 33], and document editors [19, 37].
- It leads to uniformity since the different interfaces share a common set of editor commands. For instance a single 'delete' command can be used to delete a file from a directory, a process from the list of active processes, a file from a line printer queue, a window from the screen, or a user from the list of current users.

- It removes the traditional distinction between 'edit time' and 'run time'. In traditional environments, the interaction with a typical application can be divided into two phases. During the first phase, a text editor is used to compose the input of a program. Subsequently, during the 'run time' of the program, the input is checked for errors, and output is produced. The editing model combines these two phases into a single phase. As a result the user receives incremental feedback.

This paper describes an approach to extending conventional programming environments to support the editing model of interaction. (By conventional, we mean an environment based on algebraic programming languages such as Pascal, Modula, and Mesa.) A particular implementation of this approach, called *Dost*, has been constructed for the *Xerox Development Environment* (XDE) operating system and the *Mesa* programming language [40]. We shall use Dost as a concrete example to illustrate the main features of our approach. Later, we will discuss the features of XDE and Mesa that are particularly good or bad for supporting the approach and speculate on the difficulty of applying it to other environments.

The remainder of this paper is structured as follows. Section 2 contains an overview of Dost. We describe its main components and present an example to illustrate how an interactive application looks to the end user and to the application programmer. Section 3 is the major portion of the paper. We survey the main concepts of our approach and show how a traditional environment such as XDE is extended to support the editing model of interaction. Section 4 describes the Dost implementation and reports on our experiences with it. Section 5 compares our approach with form development systems, generators of language-oriented editors, and other related work. Section 6 discusses potential directions for future work. Finally, section 7 summarizes our results.

A preliminary version of these results were presented previously [5]. More details may be found in the first author's Ph.D. dissertation [4].

2. OVERVIEW

Dost differs from XDE and other traditional environments in several important ways. Application programmers do not write programs; instead they create *classes*. (Dost is thus an example of an "object-oriented system."[13]) Users of Dost do not 'run' programs, instead they 'edit' *objects*. An object is an instance of a class. The class describes the data encapsulated by the object and the *methods* to manipulate them. The methods of an object are invoked in response to *messages* from other objects and users (through dialogue managers, discussed later).

An object is associated with one or more *presentations* that display the data encapsulated by the object. Changes in the presentation cause corresponding changes in the object. Similarly, changes in the object due to internal changes or messages from other objects cause its presentations to be updated.

Between each object and a user is a *dialogue manager*. A dialogue manager handles user interaction on behalf the object. It offers the user a structure editor interface to modify the presentations of an object. It announces user-caused changes to a presentation in messages to the object. Similarly, it updates a presentation for the user in response to messages from the object.

Thus from the point of view of objects, the user appears to be another object that can send and receive messages. From the point of view the user, the objects appear to be data that can be edited. The dialogue manager acts as an intermediary between the object and the user, translating between the languages of object interaction and user interaction.

A dialogue manager is provided automatically by the environment. As a result, an application programmer is concerned only with the specification of the user interface of an object and not its implementation.

2.1. Example

We illustrate the main features of Dost through a sample class 'Bibliography', which defines bibliographic databases. Each instance of the class manages a database, and allows a user to add, delete, and modify entries. We first describe how an end user interacts with an instance, and then show the code that must be written by an application programmer to define the desired behavior.

The User's View

To interact with an object, the user first loads its presentation(s) into a Dost *window*. A Dost window is like an XDE text window except that it is managed by a dialogue manager instead of a text editor. The user then proceeds to edit the presentation by executing a series of generic *editing commands*. In the following discussion, these commands are indicated by words in **boldface**. In the actual Dost implementation, the commands can be executed by selecting from a menu in the window's command pane. The most common commands are also bound to mouse buttons or dedicated keys on the keyboard.

Figure 2 shows the structure of an empty Dost window. Let us assume the user wants to create a new instance

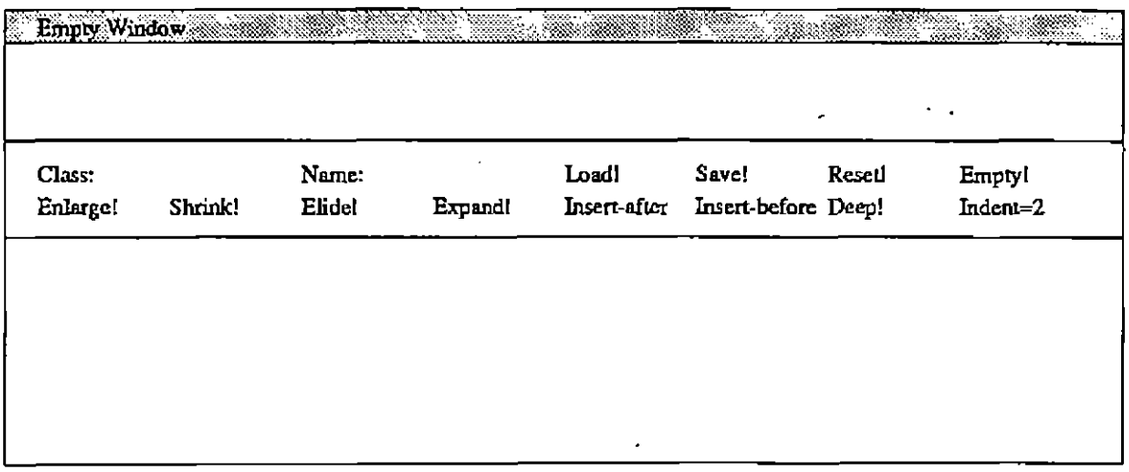


Figure 2
An Empty Dost Window

of the class 'Bibliography'. He fills the class and instance ('myBib') names in the appropriate fields of the window and executes the **load** command. Figure 3 shows the result of executing the command. A new object called 'myBib' is created and its presentation is loaded in the window. Initially, the object contains no entries. Therefore its presentation contains the *placeholder*

<ReferenceList>

which indicates that a new list of reference entries may be added to the presentation.

Figure 4 shows the sequence of actions a user may employ to replace this placeholder with a list of references. Each box displays the current presentation of the object. (For brevity, only the bottom pane of the window is shown.) An arrow indicates an editor command invoked by the user, and the shaded text in a box indicates the operand of the editor command.

The user can use the **expand** command to replace the initial placeholder with a template for a new reference. The template contains placeholders for the fields of the entry. In our example, these may be replaced to enter the author name, title, and the kind of entry desired. An entry may be a reference to a journal, book, or technical report.

To enter an author name, the user chooses the 'author' field using the **select** command and then uses **replace** to replace the placeholder with the appropriate name. The 'title' field may be entered in a similar fashion. The user may use the **menu** command to request a list of valid choices for the 'referenceKind' field. In the example, the user selects a 'Book'. In response, the dialogue manager replaces the field with fields appropriate to a book reference. In this example, there is only one such field, which prompts the user for the publisher of the book. If the user were to select 'Journal' or 'TechRept', the dialogue manager would prompt the user for a journal name or the name of an institution.

Figure 4 illustrates other commands available to the user. The user can select the whole entry by executing the **enlarge selection** command, and then hide its details by executing the **elide** command. The effects of these two

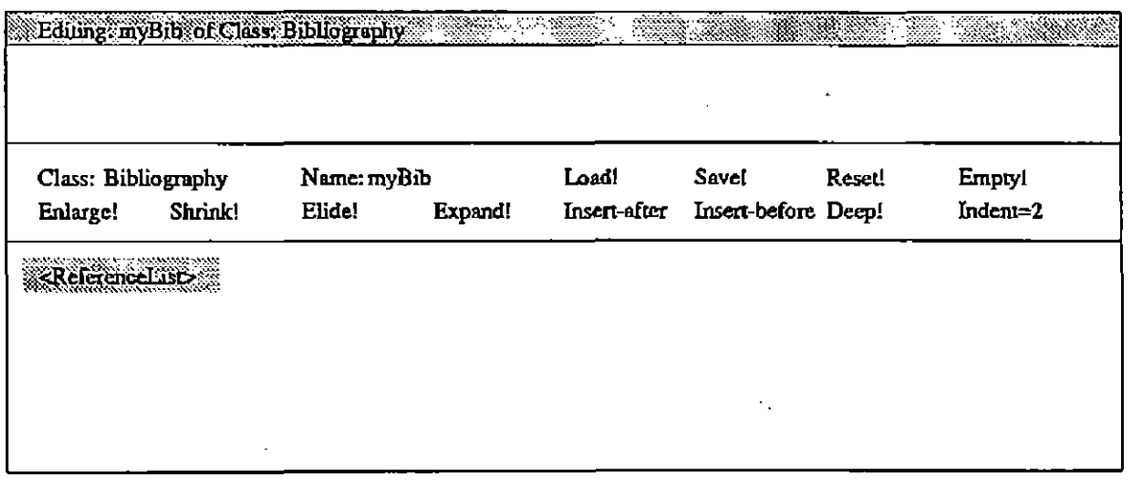


Figure 3
Creating a Bibliography

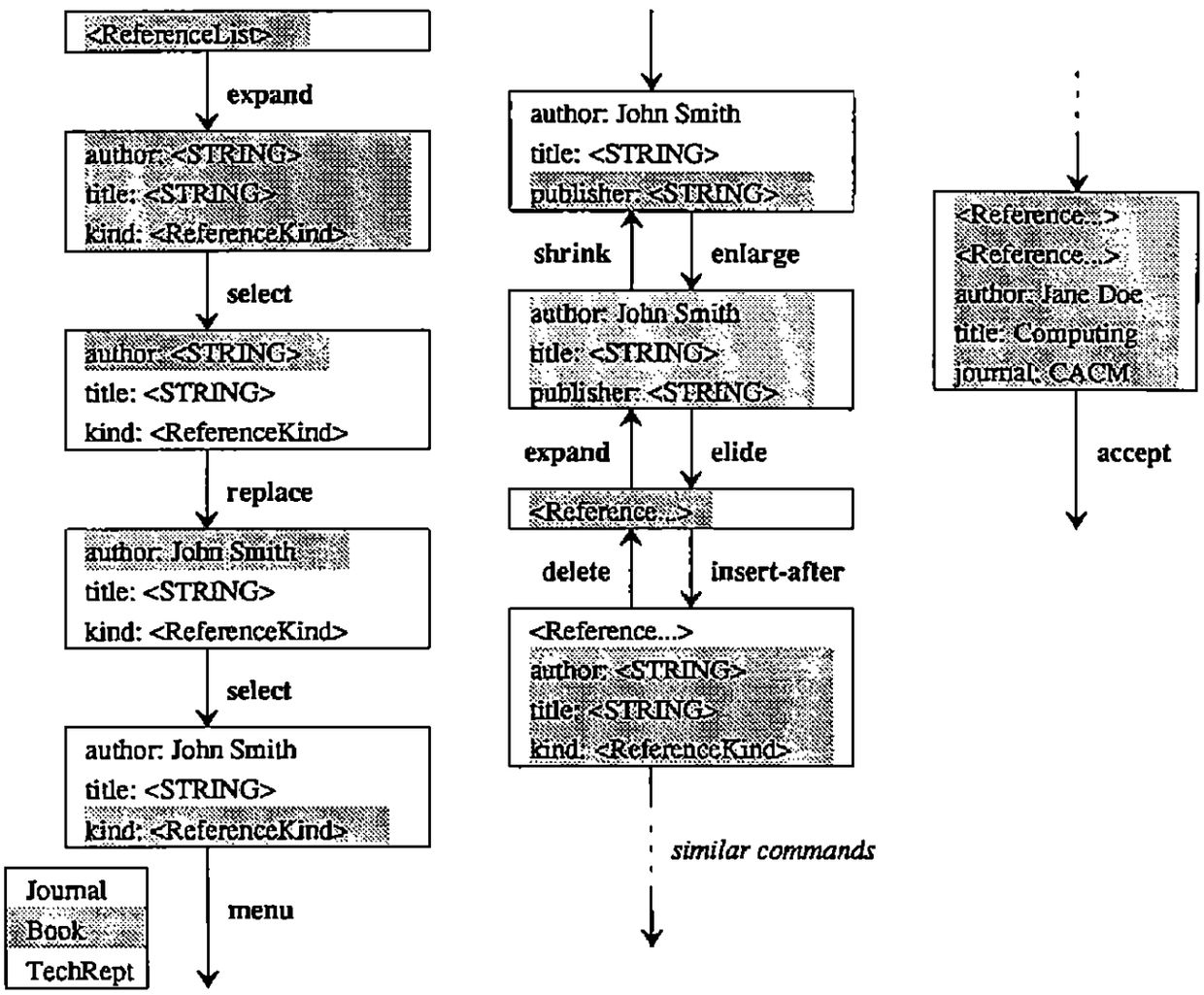


Figure 4
Editing a Bibliography

commands may be reversed by executing the **expand** and **shrink selection** commands respectively. The **insert after** command may be executed to insert a template for a new entry. (The template may be removed by executing the **delete** command.) The user may now fill this entry, and add other entries using similar commands. After specifying all the elements of the list, the user may execute the **accept** command. At this point, the new list of entries is sent to the object.

The Programmer's View

Figure 5 shows the text of the class `Bibliography`, which is written by the application programmer to support the user interface we have just seen. The class declares the variable `refList`, which stores the database of reference entries. The variable is a pointer to a record containing a Mesa *sequence* of `Reference` records. (A Mesa sequence is an array of variable size). A `Reference` is a variant record discriminated by the `reference-`

Kind field. These declarations are used by the dialogue manager as a guide for the user interface of instances of the class.

```
1     -- Class name
2     Bibliography: CLASS = {
3
4           -- Type declarations
5           ReferenceKind: TYPE = { Journal, Book, TechRept };
6           Reference: TYPE = RECORD [
7                 author, title: STRING;
8                 info: SELECT referenceKind: ReferenceKind FROM
9                       Journal: [journal: STRING],
10                      Book: [publisher: STRING],
11                      TechRept: [institution: STRING]
12                ];
13          ReferenceList: TYPE = POINTER TO RECORD [
14                list: SEQUENCE length CARDINAL OF Reference];
15
16          -- Instance variables
17          refList: ReferenceList;
18
19          -- Methods
20          EditMe: METHOD[ dm: DialogueManager ] = {
21                -- method invoked when an instance is edited
22
23                -- set default attributes for type STRING
24                dm.Alignment[ attrGrp: STRING, val: vertical ];
25                dm.Titled[ attrGrp: STRING, val: TRUE ];
26
27                -- set default attribute for field referenceKind
28                dm.Titled[ attrGrp: Reference.referenceKind, val: TRUE ];
29
30                -- declare update method for type ReferenceList
31                dm.SelfUpdate[ attrGrp: ReferenceList, val: RefListUpdated ];
32
33                -- submit a value for editing
34                dm.Edit[ var: refList ];
35          }; -- end of EditMe method
36
37          RefListUpdated : METHOD[ newVal: ReferenceList ] = {
38                -- method invoked when a value of type ReferenceList is updated
39
40                refList <- newVal
41          };
42
43          --Body of class Bibliography
44          MakeEditable[ load: EditMe ]
45    }; -- end of class Bibliography
```

Figure 5
Declaration of Class Bibliography

The body of the class (which is executed by each new object) consists of the call of the Dost function `MakeEditable` (line 44), which expresses the caller's willingness to interact with users through dialogue managers. `MakeEditable` has one parameter, `load`, which is a method defined by the programmer to be invoked when the user asks a dialogue manager to load an instance into its window. In this case, the load method is `EditMe`, which is defined on lines 20 to 33. `EditMe` has one parameter, `dm`, a reference to a dialogue manager. `EditMe` uses `dm` to send messages regarding the presentation of the object.

The first two messages sent by `EditMe` ask the dialogue manager to associate certain display properties with all variables of type `STRING` in the object. Line 24 asks the dialogue manager to align `STRING` variables vertically, while line 25 asks it to *title* them. (If a variable is "titled," its name precedes its value. For instance, because the author field is titled, it is displayed as "author: John Smith" rather than simply "John Smith.") Similarly, line 28 informs the dialogue manager that the `referenceKind` fields of `Reference` variables should also be titled.

Line 31 asks the dialogue manager to call the method `ReferenceListUpdated` (which is defined on lines 37-41), whenever a variable of type `ReferenceList` is updated by the user. Finally, line 34 asks the dialogue manager to display variable `refList` as part of the presentation of the object. The dialogue manager uses the information in the type declarations of class `Bibliography` to generate a default representation of `reflist`, customized by the messages on lines 24, 25, and 28, and displays `reflist` for the user to edit. Later, when the user executes the **accept** command, the dialogue manager calls `ReferenceListUpdated`, passing it the new list of reference entries in the `newVal` parameter. The object uses this value to update its version of the list stored in `refList`.

Figure 5 demonstrates the degree of automation provided by Dost. The class declaration contains very little code to handle interaction with the user: Only the call `MakeEditable` and the method `EditMe` are specifically related to user interaction. All other details are handled by dialogue managers for instances of the class, and these objects are generated automatically from the class declaration. On the other hand, this example also illustrates how the application programmer can override default behavior as necessary, with messages like `Alignment` and `Titled`. We will present examples below that show how more elaborate tailoring of the user interface to the application can be accommodated.

3. MAIN CONCEPTS

We now describe some of the basic concepts behind the design of Dost. These include:

- *Objects* and *Classes*, which are the basic components of the environment, replacing processes and programs in traditional systems.
- *Dialogue managers*, which replace I/O procedures provided by traditional systems and provide a much higher level of automation.
- *Presentations*, which display visual representations of data.
- *Attributes* and *attribute inheritance*, which allow an application programmer to exert control over the interfaces generated, replacing formatting constructs in traditional languages.

- Mechanisms for keeping the value of a variable consistent with its display.
- The user interface, available for interaction with all objects.

3.1. Objects and Classes

Dost objects are like a cross between processes and files in traditional systems. Like processes, objects can communicate with each other. Like files, they are part of the permanent memory of the system. They have permanent names, and can activated and passivated. A Dost class is a Mesa program, extended with constructs to support inter-object communication and permanence.

Inter-Object Communication

An object communicates with another object my sending it a *message*, which invokes a *method* in the receiver. Sending a message to an object is similar to invoking a procedure in a traditional module instance. The difference is that procedures are used to communicate between instances of modules linked together in a program, while messages are used to communicate between instances of classes, which, (like traditional programs, of which they are extensions) are separately linked.

In Dost, a method declaration is identical to a procedure declaration, except that the keyword `METHOD` replaces the keyword `PROCEDURE`. Figure 5 illustrates how methods are defined; it contains declarations of the methods `Load` and `RefListUpdated`.

Similarly, sending a message to an object is similar to invoking a procedure in a module instance, except that a message names an object instead of a module. For example, a message is sent to an instance of `Bibliography` by executing

```
I.RefListUpdated [newVal: s]
```

where `s` is an argument of type `ReferenceList`, and `I` names the instance. (Like Mesa procedure calls, messages can use either positional or keyword notation for parameters. In this paper we shall use keyword notation exclusively.)

How should an object be named in a message? Clearly a static name such as a module name does not suffice, since objects are created dynamically. Therefore objects are named by *object pointers*, whose values are bound at runtime. In the above example, `I` is an variable that has been assigned a pointer to an instance of class `Bibliography`.

Since the interactive user cannot refer to an object pointer directly, each object, like a file is given an character-string name. A user may use this name to specify an object for editing. An object may use this name to get a pointer to an object by calling a predefined procedure that 'opens' the object for communication. When the first object decides that it no longer needs to communicate with the second object, it can call a predefined procedure to 'close' the object pointer that references the object.

Constructs for Permanence

The editing model supported by Dost allows each object to represent permanent data that can be saved between editing sessions. Therefore, it is important that the environment provide support for permanence in objects.

One approach to supporting permanent objects, used in Smalltalk [14], is to divide a user's interaction with a system into several sessions. At the end of a session, the user asks for a snapshot of the state of the entire system to be saved in secondary storage, from which it is reloaded at the start of the next session. The saving and restoring of state is done by the system.

Our approach is different from Smalltalk in several respects. It is simpler to support, and allows each object to be activated and passivated individually. However, it makes an object responsible for saving and restoring its state. We have chosen this approach mainly for its simplicity.

A programmer may make a class of objects permanent by including the parameter `dataFile` (of type `STRING`) in the declaration of its parameter list. The system associates each instance of such a class with a unique file called its *data file*, which may be used by the object to store its permanent data structures. When an existing instance is activated, or a new instance is created, the `dataFile` parameter contains the name of the data file of the instance. The object can read its data structures from this file.

A passive object is activated when another object opens it for communication. When all object pointers to an executing object are closed, the system decides to passivate the object. Before it actually unloads the object from memory, it calls the *passivate method* of the object to give the object an opportunity to save its data structures in its data file. An object introduces this method to Dost by calling the system procedure `RegisterPassivate-Handler`.

An example may clarify these ideas. The following class fragment shows how `Bibliography` may be extended to support permanent instances:

```
Bibliography: CLASS[ dataFile: STRING] = {

      -- Type declarations
      ReferenceKind: TYPE = { Journal, Book, TechRept };
      Reference: TYPE = ...;
      ReferenceList: TYPE = ...;

      -- Instance variables
      refList: ReferenceList;

      -- Utility procedures
      SaveState: METHOD[] = {
            -- save refList in file dataFile
      };

      RestoreState: PROCEDURE[] = {
            -- restore refList from file dataFile
      };

      -- Methods
      EditMe: METHOD[ dm: DialogueManager ] = { ... };

      RefListUpdated : METHOD[ newVal: ReferenceList ] = { ... };

      --Body of class Bibliography
      RestoreState[];
      RegisterPassivateHandler[ passivate: SaveState ];
      MakeEditable[ load: EditMe ]
}; -- end of class Bibliography
```

When an instance of the class is created or activated, its main body is executed, which reads any saved data from the data file, and registers its passivate and load methods. The object then executes methods in response to messages from objects that have opened it for communication. When all object pointers to the executing object are closed, the `SaveState` method is invoked, which saves the state of the object in the data file.

3.2. Dialogue Manager

In a traditional system, a process calls input/output procedures to interact with a user. These procedures automate several aspects of the process' interaction with the user. They relieve a process from the task of echoing input, providing simple editing commands such as 'erase character' and 'erase line', converting between simple values such as integers and reals and their visual representation, and, in a window based system, multiplexing and demultiplexing the input/output among the different windows.

In Dost, the set of input/output procedures are replaced by a dialogue manager, which manages an object's interaction with the user and provides a much higher level of automation than the set of input/output procedures it replaces. An object performs its I/O operations by sending messages to a dialogue manager and invoking methods in response to messages from the dialogue manager.

Not all objects are connected to dialogue managers—only those that are interacting with users. A connection between an object and a dialogue manager is established and broken explicitly by a user. At any moment, one or more dialogue managers may be active in Dost. Each dialogue manager displays a *window* on the screen. Some of

these windows are busy; these correspond to dialogue managers connected to objects. Others are empty; they correspond to dialogue managers that are idle and can be connected to objects. New dialogue managers are created when the user creates empty windows. They are destroyed when the user deletes windows. To connect an idle dialogue manager to an object a user enters the class and object name in appropriate fields of the dialogue manager's window and executes the load command. The connection is broken by the unload command.

When a user executes the load command, Dost invokes the object's *load method*, which is introduced to Dost by the system procedure `MakeEditable`. An object may communicate with several dialogue managers simultaneously, each allowing a different user to interact with the object via a separate window. The load method is called each time a user loads an object in a new window. An object may also define *unload*, *save*, and *reset* methods, to be invoked when the user executes the corresponding commands. (The latter two commands have no other effect than delivery of the corresponding messages. By convention, these methods are expected to checkpoint and restore the object's permanent state.) These methods are optional and a dialogue manager attempts to invoke them only if they have been defined.

Dialogue managers are themselves objects. However, they differ from other objects in two important ways: First, they cannot themselves be edited by users. Therefore, they are not given character-string names. Second, messages sent to them need to be preprocessed, as described in § 4.

3.3. Presentations

An important concept in Dost is the idea of a *presentation* of an object, which is a visual representation of data in the object. It is composed of the presentations of one or more of the object's variables. It is similar to output in a traditional system, but it may also be edited by the user to effect a corresponding change to the data it displays. Moreover, it may also be modified by the object to show new values.

An object may have several presentations displayed simultaneously on the screen, each presentation displaying a different 'view' of the object. For instance, a spreadsheet manager might simultaneously create one presentation to display the values of the spreadsheet and another to display the expressions that define the relationships among them. The different presentations of an object are displayed by a dialogue manager in different *subwindows* of a window. The object specifies what data are displayed in each presentation. The dialogue manager uses this information to construct the presentation.

An object specifies the data to be displayed in a presentation by sending *edit messages* to a dialogue manager. Each such message names a variable to be displayed and a *subwindow number* indicating the presentation to which it is to be appended. Thus the message

```
dm.Edit [var: values, pres: 2]
```

asks dialogue manager `dm` to append the presentation of `values` to the contents of the subwindow `2`. (Many objects will have only one presentation at a time. Therefore, the presentation parameter of an edit message is optional and defaults to 0.)

Edit messages may display not only variables of predefined types such as characters and integers but also variables of programmer-defined types such as records, arrays, sequences, and variant records. For instance in Figure 5, the edit message

```
dm.Edit [var: refList]
```

asks a dialogue manager to display a sequence of variant records. The dialogue manager has information about the types declared in the class of the object, and uses this information to construct a default representation for each type. These defaults can be modified in a variety of ways as explained below.

A simple variable is displayed by displaying its value. Thus, a variable of type 'SampleEnum' described by the declaration

```
SampleEnum: TYPE = {choice1, choice2, choice3};
```

is displayed as `choice1`, `choice2`, or `choice3`, depending on its value. An array is displayed by displaying all its elements. For instance, if `SampleArray` is defined by

```
ElementIndex: TYPE = {element1, element2, element3};
SampleArray: TYPE = ARRAY ElementIndex OF SampleEnum;
```

a value of type `SampleArray` might be presented as

```
choice1
choice1
choice3
```

The presentation of a sequence is similar to the presentation of an array; the only difference is that the number of elements displayed depends on the size of the sequence.

The presentation of a record displays its fields. Thus if `SampleRecord`

s defined by

```
SampleRecord: TYPE = RECORD [
    f1: INTEGER,
    f2: SampleEnum,
    f3: STRING];
```

a value of this type might be displayed as

```
1
choice2
a sample string
```

The default presentation of a variant record is the default presentation of its current variant. Thus a variable of type

```
SampleVariant: TYPE = RECORD [
    f1: INTEGER,
    unionField: SELECT disc: SampleEnum FROM
        choice1: [f2, f3: INTEGER],
        choice2: [f4, f5: STRING],
        choice3: [f5, f6: CHARACTER]]
```

might have the presentation

```
1
another string
yet another string
```

if its `disc` field has the value `choice2`. The value of the discrimminant is not displayed in this presentation since it is implied by the fields displayed in the variant part.

Finally, the default presentation of a non-null pointer is simply the presentation of the value to which it points. The default presentation of *nil* is the null string. Thus a pointer variable of the type

```
List: TYPE = POINTER TO RECORD [
     contents: INTEGER;
     next: List]
```

might be presented as

```
2
3
5
```

An object may use the presentation of pointers to display recursive structures. Our presentation scheme for pointers would create an infinite presentation for cyclic structures. The dialogue manager uses an algorithm of 'lazy dereferencing' to prevent itself from creating such a presentation. Normally, the presentation of a pointer is 'elided' (see next section) and does not show the referant. An 'elided' presentation is dereferenced when the user invokes the **expand** command on it.

Commands that alter the 'current selection' follow the hierarchical structure of presentations. The editing commands **enlarge selection**, **shrink selection**, **next**, and **previous** move up, down, left or right in a tree-structured value. (The 'current selection' is actually stored by the dialogue manager as a pair of nodes: a leaf, which represents the current point of focus, and an ancestor of that leaf, which defines a region of text surrounding that point.)

3.4. Attributes

Defining a default presentation for every data type is a great convenience, but it is clearly much too rigid. Therefore, we associate with each displayed value a set of *attributes*, with default values determined by the type of the value, which can be modified by the object or by the interactive user. The dialogue manager uses these attributes to determine how the value should be displayed. These attributes have been chosen to allow a user or an object to specify general characteristics of the presentation of a variable, while leaving details of the presentation to the dialogue manager. Some of the more important attributes currently defined by Dost are described in the next few paragraphs.

The *value* of a variable is a special attribute that is maintained by the dialogue manager for each variable submitted to it via the edit message. This is the value displayed in the presentation of the variable, and may be different from the actual value of the variable contained in the object. (We discuss the mechanism for keeping these two consistent in § 3.5).

The *initialized* attribute determines if the presentation of a variable is *initialized* or *uninitialized*. This attribute is defined for simple variables and union fields of records. An initialized presentation of a simple variable displays its

value while an uninitialized presentation displays a *placeholder* that may be replaced to initialize the variable. Thus a presentation of an integer variable may be

```
5
```

or the placeholder

```
<INTEGER>
```

depending on its *initialized* attribute. An initialized presentation of a union field of a record displays the appropriate variant while an uninitialized presentation displays a placeholder for the discrimminant. Thus the presentation of a record of the type `SampleVariant` defined above may have the presentation

```
1
another string
yet another string
```

or

```
1
<SampleEnum>
```

depending on the *initialized* attribute of the `unionField` of the record. Displaying an uninitialized presentation of a variable serves two purposes: First, it tells the user that an input value is expected for the variable. Second, it describes the set of legal input values.

The *elided* Boolean attribute determines whether the presentation of a structure variable shows or hides the presentations of its components. Thus the presentation of a variable of the type `List` might be

```
2
3
5
```

or

```
<List...>
```

depending on the *elided* attribute of the variable.

The *titled* attribute of a value determines whether its presentation should be preceded by a 'title.' The title of a variable is the variable name, the title of a field in a record is the field selector, and the title of an array element is its subscript. For example, a `SampleRecord` might be displayed as

```
f1: 1
f2: choice2
f3: a sample string
```

or

```
1
choice1
a sample string
```

depending on whether the fields are titled. Similarly, if the elements of a `SampleArray` are all titled, the array might be displayed as

```
element1: choice1
element2: choice2
element3: choice2
```

The *alignment* attribute determines if the presentation of a variable is *horizontal*, *vertical*, or *indented* with respect to the presentation of the preceding sibling. Thus the presentation of v2 may be aligned in one of the following ways depending on its alignment attribute:

```
v1: 5  v2: 3
```

```
v1: 5
v2: 3
```

```
v1: 5
    v2: 3
```

The attributes of a displayed variable influence more than its presentation. They are a mechanism for specifying to the dialogue manager general properties of the variable. An example of such a property is whether the value of the variable is readonly or changeable by the user. We study some of the other properties later.

3.4.1. Attribute Inheritance

Requiring the programmer to specify individual attributes of individual values, while providing great flexibility, would be unbearably cumbersome. Therefore, Dost accepts messages that allow an object to change whole classes of attributes at once.

Each object is associated with a *tree* of *attribute groups*. Different trees can be defined for different presentations, but the tree always has four levels, called the *default*, *type*, *component*, and *variable* levels. As an example, consider an object that has the following declarations

```
SimpRef: TYPE = RECORD [
      author, title: STRING];
refVar: SimpRef;
```

An attribute-group tree for this object is shown in Figure 6.

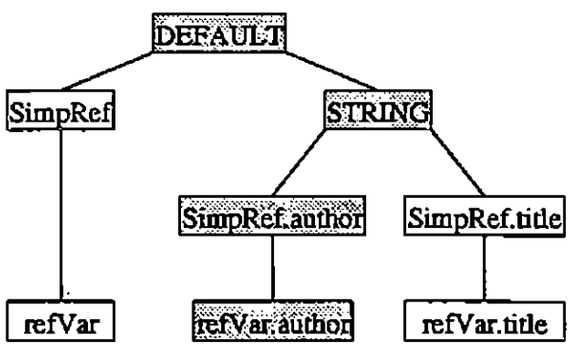


Figure 6
An Attribute-Group Tree

The root of the tree is the *default group*. Attributes in this group apply to all values that do not override them at lower levels of the tree. For example, if the default group contains the attribute binding

```
titled = TRUE
```

then all values will be titled unless specified otherwise.

Children of the default groups are the *type groups*—one for each type defined in the object. Attributes in this group apply to all values of the type. For example, if the binding

```
alignment = vertical
```

appears in the type group for the `STRING` type, then all values of type `STRING` will be stacked vertically by default.

The third level contains the *component groups*, which are associated with field selectors and array subscripts. If the component group `SimpRef.title` contains the binding

```
alignment = horizontal
```

then the `title` fields of `SimpRef` values will be displayed horizontally, overriding the *alignment* attribute in the `STRING` type attribute group.

Finally, there is a *variable attribute group* associated with each variable.

Each attribute group begins with an initial value for each attribute, which may be changed or *redefined* dynamically as a result of the object sending the dialogue manager a message or the user executing an editor command. The new value is inherited by all children (and their children, and so on) in which the attribute has not been redefined. Thus if an attribute is changed in the attribute group "DEFAULT", then the shaded nodes show a possible path along which the changed value is inherited.

The method `EditMe` in Figure 5 illustrates how attributes are specified by an object. It sets attributes of the attribute group `STRING` and `Reference.ReferenceKind`. In § 3.7, we explain how attributes can be specified by users.

Together, attributes and attribute inheritance provide a balance between *flexibility*, which allows the interface of an object to be tailored according to the specific needs of the object, and *automation*, which frees the programmer or the user from the task of implementing the interface of an object. Attributes provide the mechanism for flexibility, while attribute inheritance provide the mechanism for automation.

In the rest of this discussion we shall not distinguish between type groups, component groups, and variable groups, which are defined by a dialogue manager, and the corresponding types, components, and variables, which are defined by an object and its class. For instance, we shall continue to talk about attributes 'of variables,' though, strictly speaking, the attributes are associated with variable groups defined by a dialogue manager for these variables.

3.5. Keeping a Variable and its Display Consistent

Once a variable has been displayed as a result of an edit message, the value of the variable maintained by the object and the value displayed by the dialogue manager can become inconsistent. The value stored in the object may

change because of internal changes in the object or messages from other objects. The value displayed by the dialogue manager may be changed by the user. Dost provides several primitives that can be used to restore consistency.

3.5.1. Changes in the Variable

First consider the case when an inconsistency arises because of a change in the value stored in the object. An object can update the display by sending an *update message* to the dialogue manager informing it about the new value of the variable. The dialogue manager responds to the message by changing the *value* attribute of the variable and updating the presentation appropriately.

To illustrate update messages, let us consider the example of a record `R` of type `SampleRecord` that is displayed to the user as

```
1
choice2
a sample string
```

Assume that the object changes the value of field `f2` of of the record to `choice1`. It can inform the dialogue manager about the change by sending the message

```
Update [attrGrp: R, newVal: [f1: 1, f2: choice1, f3: "a sample string"]]
```

to the dialogue manager. The first parameter names the attribute group whose value attribute is to be updated and the second parameter gives the new value of the attribute. In the above example the value attribute of the variable group corresponding to variable `R` is to be updated. (Update messages can also be used to change the value attribute of attribute groups corresponding to types and components of structures, and the default group. The dialogue manager responds to the message by propagating the new value to the appropriate attribute groups.)

In this example, the object sends the whole record to the dialogue manager when only one field is updated. It may be preferable to send information about *incremental* changes to a structure, to decrease the amount of data transmitted, and to simplify processing in the dialogue manager. Incremental updates to a structure are easily achieved by changing the value attribute of a substructure that changed rather than the complete structure. Thus, in the above example, an object can send the message

```
Update [attrGrp: R.f2, newVal: choice1]
```

to update the presentation of the field that changed. Elements of arrays and sequences can be updated in a similar manner. Additional kinds of messages are defined to inform a dialogue manager about elements inserted into or deleted from the middle of a sequence.

3.5.2. Changes in the Display

When the user edits a presentation, the value attribute of a variable may change. The object needs to be told about the change so that it can update the variable. Continuing our previous example, suppose the object binds the following method to the `selfUpdate` attribute of variable

```
RUpdated: METHOD [newVal: SampleRecord] = { R <- newVal };
```

If the user edits the `f3` field to read

```
another string
```

the dialogue manager calls `RUpdated` with the new value of `R` and the instance variable `R` is updated accordingly.

The value of `selfUpdate` must either be a method or *nil*. In the latter case, no method is invoked on update.

In this example, the object receives the whole record whenever any field is updated. Sending *incremental* changes to a variable can cut down on the amount of data transmitted and relieve the object of the task of finding the part that changed. An object will receive incremental changes to a structured variable if it has bound update methods to components of the variable. In our example, the object could set the *selfUpdate* attribute of the attribute group `R.f2` to the method

```
RDotF2Updated: METHOD [newVal: SampleEnum] = {
     R.f2 <- newVal};
```

if it wished to be informed of changes to that field.

Although the same technique can be used to trap updates to elements of arrays, it would be rather clumsy to bind a separate method to each component of a 1000-element array. Therefore, arrays also have an *elementUpdate* attribute which may be bound to a method that is to be called with the subscript and new value of a changed element. For example, the array `A` could have its *elementUpdate* attributed bound to the method

```
AElementUpdated: METHOD [index: ElementIndex, newVal: SampleEnum] = {
        A[index] <- newVal};
```

For sequences, *insertUpdate* and *deleteUpdate* attributes are defined, in addition to *elementUpdate*.

One more attribute controls the triggering of update methods. If the Boolean attribute *incFeedback* is `TRUE`, then any change to the corresponding value causes update methods (if any are defined) to be invoked immediately. If *incFeedback* is `FALSE`, the update methods will only be invoked when the user issues an accept command. Setting *incFeedback* is useful if frequent changes requiring feedback (such as validation) are expected or if it is desired to cut down on keystrokes. On the other hand, if users are expected to change several values "at once," it may be desirable to disable `incFeedback` to prevent spurious error messages while values are inconsistent.

A single change can trigger several update methods. In such cases, the update methods are all called in order, from most specific to most general. For example, if *selfUpdate* methods are defined for a record as well as one of its fields and the field is changed, the method for the field is called immediately followed by the method for the whole record. Similarly, if a sequence has *elementUpdate*, *insertUpadate*, and *selfUpdate* methods, they will all be called (in that order), when a new element is added. These routines can then process the new element, insert it into data structures, and check the entire sequence for validity, respectively.

3.6. User Interface

We briefly presented a typical interactive session in § 2. In this section we discuss the user interface in more detail.

At any instant, the screen is composed of one or more Dost windows, examples of which are shown in Figures 2, 3, and 8. A Dost window is like an XDE text window except that it is managed by a dialogue manager instead of a

text editor. As a result an object of an arbitrary class can be edited in such a window. A window has one or more *presentation subwindows* as well as a *message subwindow* for error messages, a *command window* for entering certain commands, and an *error subwindow* for error messages.

The user interacts with an object by loading its presentations into the window and issuing editing commands. The most common commands are bound to menu items or dedicated buttons on the keyboard. Commands are divided into six categories:

(1) *window editing* commands,

(2) *object editing* commands,

(3) *text editing* commands,

(4) *structure editing* commands,

(5) *attribute editing* commands, and

(6) the *accept* command.

Most window editing commands are supported directly by the XDE window manager. Dost provides the commands to create and delete windows, since they result in activation and deactivation of dialogue managers.

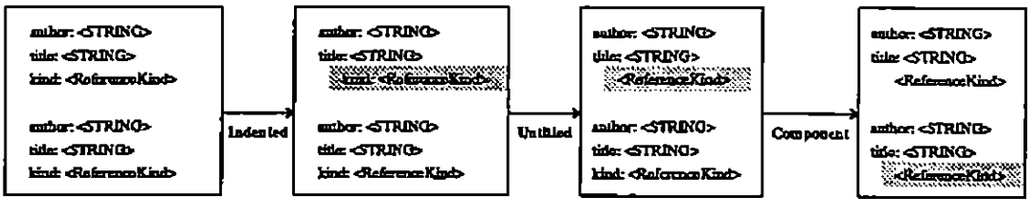
The object editing commands are used to load and empty the presentations of an object from the window, and to save and reset user changes to the presentations of the object. They replace similar commands provided by the XDE text editor to load and empty the contents of a text file into a window, and to save and reset user changes to the file.

The text editing commands are supported by the XDE text editor, and allow a user to manipulate a presentation of an object as text. They include commands for selecting text, modifying the presentations of simple variables, searching for patterns, scrolling, and so on.

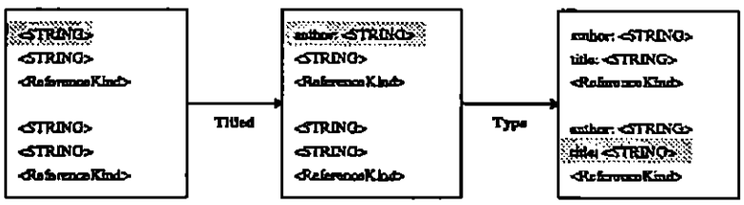
The structure editing commands allow a user to manipulate a presentation as structured text. They include such commands as **enlarge selection**, **shrink selection**, **elide**, and **expand**.

The attribute editing commands allow a user to change attributes interactively. Although most attributes would normally be set by the object rather than the interactive user, these commands are occasionally useful to allow the user to customize the interface according to personal preferences. More importantly, we have found them very useful for the application designer who can quickly check the readability of various combinations of attributes. Attributes in variable groups are changed by selecting the presentation of the appropriate variable and executing a command. Attributes in component, type, or default groups are changed by changing the attributes of a variable of the appropriate kind and then invoking the **component**, **type**, or **default** command.

Figure 7 illustrates the use of the **component** and **type** commands. In Figure 7 (a) the user sets the *alignment* attribute of a `kind` field of a `Reference` record to `indented`, and the `titled` attribute to `FALSE`, and executes the component command. The appropriate values are propagated to all `kind` fields of all `Reference` records. Similarly, in Figure 7 (b) the user sets the *titled* attribute of a variable of type `STRING` to `TRUE`, and then



(a) Setting Attributes of Components



(b) Setting Attributes of Types

Figure 7
Changing Presentation Attributes

uses the type command to propagate the change to all `STRING` variables.

The accept command is used to send the new value of a variable to an object. The dialogue manager reacts to the command by invoking appropriate update methods in the object, as discussed in § 3.7.

In the design of the user interface of Dost, we have made a distinction between the *abstract command* and the *mechanism* provided to invoke it. For instance the abstract command `replace`, which changes the value of a simple variable, is invoked by selecting appropriate characters in the presentation of the variable with the aid of the mouse, deleting them using the *delete* key, and inserting the new characters. The `elide` command is invoked by selecting a variable with the aid of the mouse, and then clicking the mouse at the *command item* `Elide` in the command window. In general a user invokes a command with the aid of the mouse, keyboard, and menus.

3.6.1. Object-Specific Editor Commands

We have so far discussed the set of *default* commands provided by Dost. A dialogue manager implements these commands, and decides which commands are available to edit a presentation of a variable. Delegating these tasks to dialogue manager is a great convenience, but may be too rigid for some applications, Therefore, Dost allows an object to tailor its user interface by restricting the use of current commands, changing their implementation, or defining new commands.

For each command, a variable is associated with a Boolean *command enable* attribute that determines if the command may be applied on the presentation of the variable. For instance, the *expandEnable* attribute of a variable determines if an elided presentation of a variable may be expanded. An object may change the values of command enable attributes to restrict the set of commands that may be applied to the presentation of a variable.

Each command also associates an *override* attribute with each variable, which may be assigned a method that overrides the default implementation of the command. The value of an override attribute must either be a method or *nil*. In the latter case, the default implementation of the command is used.

An object may provide a new command by sending a message to the dialogue manager that names the new command, the attribute group (§3.4.1) to which the command applies, and a method that implements the command. For example, an instance of `Bibliography` may send a dialogue manager the message

```
AddCommand [attrGrp: ReferenceList, name: "sort", cmdMethod: SortRefs];
```

to define a command that sorts variables of type `ReferenceList`.

An object-defined implementation of an editor command is often most conveniently implemented using the default or current implementation of another command. Therefore Dost allows an object to send messages to the dialogue manager asking for editor commands to be invoked. Each such message indicates the command to be invoked, the attribute group to which the command applies, and whether the default or current implementation is to be used. The following example illustrates the use of these messages.

Consider application of the **insert after** command to an element of a sequence. The default implementation inserts a new element after the current element. Assume that an object wishes to add a new command called **append**, which, when applied to an element of a sequence, appends a new element to the list. The command method supplied for the new command can invoke the *current implementation* of **insert after** command on the last node of the sequence.

4. IMPLEMENTATION

We have implemented and tested the major components of Dost. We have built a *dialogue manager program*, which can be executed to run dialogue managers. We have also built an *object manager*, which supports objects. Finally, we have designed a *precompiler* which translates Dost classes into Mesa programs. It performs the translation by inserting procedures that allow an object to communicate with the dialogue and object managers. Since the precompiler has not been implemented, we have tested the system by hand-translating Dost classes into the equivalent Mesa programs.

We discuss below the salient parts of the three components.

4.1. Precompiler

The precompiler takes a Dost class and translates it into a Mesa Program, which is called its *class program*. The class program contains code that handles the constructs in the class not available in Mesa programs.

The methods in the class are translated into procedures. The class is associated with a *method record type*, which defines fields that can point at these procedures. An instance of the class is associated with a variable of the type, called its *method record*. A pointer to the method record of an object is stored by the object manager. Other objects can query the object manager for this pointer by making a call that opens the object for communication.

The declaration of an object pointer in a class is translated into the declaration of a pointer to the method record of the object. A message that names the pointer is converted into a call to the appropriate field of the method record.

A message to a dialogue manager is handled differently. Typically, it requires special processing of its parameters, as illustrated by the following example. Consider the message:

```
SelfUpdate [attrGrp: ReferenceList, value: RefListUpdated]
```

sent by an instance of `Bibliography` of Figure 5. It asks the dialogue manager to call the procedure `RefListUpdated` when a variable of type `ReferenceList` is updated. This message cannot be simply converted to a Mesa procedure call in the dialogue manager, for two reasons: First, the parameter `attrGrp` may be the name of a variable, type, component of a structure, or the default group. No Mesa type describes such a parameter. The precompiler needs to convert it into a parameter that encodes information about the attribute group. Second, the dialogue manager calls the method `RefListUpdated` with the value of a variable of type `ReferenceList`. Therefore, the precompiler needs to check that the parameter `newVal` of the method `RefListUpdated` is of type `ReferenceList`.

4.2. Object Manager

The object manager performs several functions. It activates and passivates objects and maintains the list of all the objects that are created. For each object, it maintains the class name, the data file, the name of the class program, a pointer to the method record, a pointer to the load method, and other information about the object. The information is manipulated by `RegisterPassivateHandler`, `MakeEditable`, and other system procedures called by an object.

4.3. Dialogue Manager Program

The dialogue manager program is a Mesa program that can be executed to create a dialogue manager. It maintains two main data structures to allow editing of an object. The first is a table of attribute groups associated with the presentation of an object. The second is the symbol table produced by the Mesa compiler after compiling the class program of the object. These two data tables are used by the dialogue manager to process editor commands executed by the user and messages sent by the object.

5. EXPERIENCE

We have used Dost to define several interactive applications including the class `Form`, which defines a form with a variable number of fields, the class `Spreadsheet`, which defines a simple spreadsheet, the class `Directory`, which defines an editable directory, the class `References` which is an extension of `Bibliography` (it defines more realistic fields for a reference entry), and the the class `ASPLE` which checks the static semantics of a toy programming language called ASPLE [22].

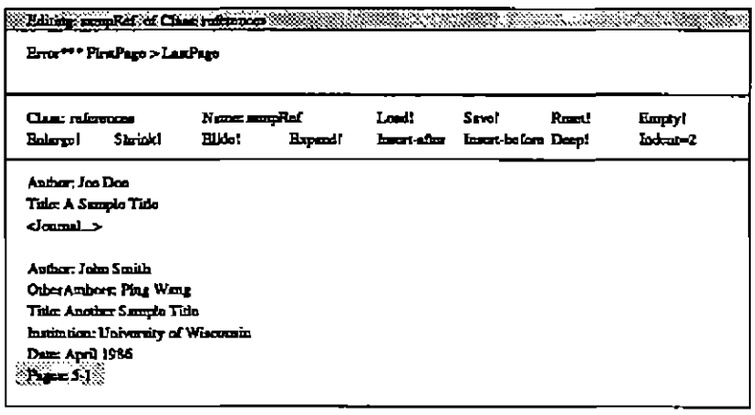
Table 1 shows the sizes of the *interaction code* of these classes. For the purposes of these measurements, we define the "interaction code" of a class to be the code required to drive the dialogue managers of an instance. It

Form	8
Spreadsheet*	14
Directory*	39
References	16
ASPLE	31

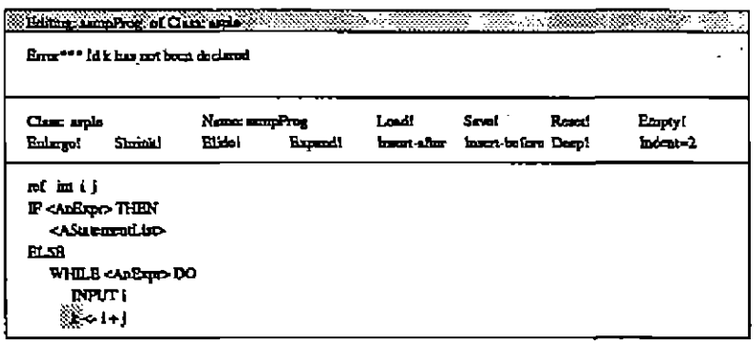
Table 1
Size (in source lines) of the Interaction Code in Different Classes

includes any code in the class that interacts with a user via a dialogue manager. Thus, it includes the code required to specify the attributes of attribute groups, submit variables for editing, and specify the load method. It does not include the code required to define the data encapsulated by an instance, or the code required to manipulate them.

"Form", "ASPLE", and "References" were actually implemented. "Directory", and "Spreadsheet" could not be implemented because some parts of the dialogue manager design were not implemented. Figure 8 displays sample presentations of instances of "References" and "ASPLE".



(a) A Bibliographic Database



(b) A Simple Programming Language

Figure 8
Examples of Dost Windows

The sizes of the interaction codes of these classes illustrate the automation in the environment. The 8-39 lines of interaction code in these classes replace thousands of lines of code that would be required to implement the interfaces manually. The dialogue manager program, which implements the interface available to all objects, is about 7000 lines of code.

6. RELATED WORK

The idea of generating user interfaces is not new. Previous approaches range in scope from form development systems to environments that support the editing model of interaction. In this section we compare our work with these approaches.

6.1. Form Development Systems

Several systems support a model of interaction based on *forms* [20, 31-33]. An application interacts with a user by displaying one or more forms to the user. Each form consists of one or more *items*, which a user fills with appropriate values.

There is an important similarity between the form model of interaction and editing model supported by Dost: A user can fill the items in any order. Thus interaction is not constrained to be sequential.

However, the form model is not as general as the editing model. It imposes three main restrictions:

(1) The number of items in a form is *static* and not a function of input.

(2) The values of items are restricted to simple types. Thus none of the structure editing commands are available to interact with forms.

(3) An application does not receive *incremental updates* to the items on the screen. It receives the values of all items together when the user executes the equivalent of the accept command in Dost.

As a consequence of these restrictions, the model is not sufficient to interact with a large number of applications. For instance, it cannot be used to define user interfaces for any of the applications mentioned in § 5. Typically, the use of forms is restricted to entering tuples in databases or setting initial parameters of an application.

6.2. EZ

Fraser and Hanson have built a software system called EZ [10, 11] based on a high-level string-processing language derived from SNOBOL4, SL5 [16], and Icon [15]. The language supports four basic types of values: numerics, strings, procedures, and SNOBOL4-like tables. The system has 'persistent' memory much like an APL workspace in which values exist until changed. As a result it integrates the traditionally distinct facilities of programming languages and operating systems into a single system. Files are represented as strings, and directories are provided as tables.

EZ provides a screen editor that edits all EZ values using the same interface. As a result it can be used to edit text files, directories, and relational data bases represented as tables. Moreover, procedure activations in EZ are just EZ tables. Therefore the editor is automatically a debugger as well.

There are two striking similarities between EZ and Dost. First, EZ's 'persistent' data corresponds to Dost's 'persistent' objects. Second, the EZ editor is similar to a Dost dialogue manager: Both are capable of editing data structures defined by a programming language.

There are, however, several important differences between Dost and EZ, arising because of the differences in the goals of the two projects. Dost illustrates an approach to support generation of user interfaces in environments based on Pascal-like languages, while EZ is an integrated operating system-debugger-editor based on a SNOBOL-like language. Thus while the EZ editor manages only two types: strings, and tables (procedures and numerics are automatically converted to strings by the language), a Dost dialogue manager allows editing of numerics, strings, enumerations, arrays, records, variant records and other data structures defined by Mesa types.

The editing commands presented by the EZ editor and a dialogue manager are also different, mainly because of the differences in the data structures managed. The EZ editor provides only text editing commands. Thus it does not provide equivalents of the structure editing and attribute editing commands provided by a dialogue manager. On the other hand, it provides an enter command, which may be applied to a line displaying a key of a table. The command recursively invokes the editor on the value associated with the key. The value may be another table, so the editor can be used to 'walk' tables. An equivalent command cannot be straightforwardly supported in Dost.

Finally, EZ does not provide an application programmer facilities to format data or check data for semantic consistency. Thus it does not support customization of the editing interface.

6.3. Descartes

Descartes [35,36] is a framework for building user interfaces having several characteristics in common with Dost. It allows an application to input and output values of programmer-defined types. Moreover, an application's interaction with the user is managed by an application-specific module called a *compositor*.

A compositor in Descartes corresponds to a dialogue manager in Dost. However, there is an important difference between the two entities: A dialogue manager is provided automatically for each Dost object and is 'driven' by a small amount of application-specific code. On the other hand, a compositor has to be developed manually for each Descartes application using 'utility code' shared by all interfaces and provided by the system. Thus, Descartes provides less automation in generating a user interface, but gives an application programmer more flexibility in specifying an interface.

The models of interaction offered by Descartes and Dost are also different. Descartes supports only sequential interaction. An application asks for values it needs in a particular order and the user is constrained to supply each value as it is requested. Dost, on the other hand, offers a more sophisticated model of interaction that allows the variables displayed in a presentation to be edited in any order. Introduction of the editing model is planned in Descartes.

6.4. Smalltalk

Dost borrows two elements of the object-oriented paradigm supported in Smalltalk: permanent objects, and interobject communication. Both properties are essential to our approach. Permanent objects are necessary to support

the editing model of interaction, which allows each application to act as an editor of permanent data that can saved between editing sessions. Interobject communication is necessary for an object to interact with the user via the dialogue manager. It is also useful for keeping data in related objects consistent.

We left certain other object-oriented features out of Dost, however, to allow our appoach to be applied to more conventional programming environments. Thus while Smalltalk treats every entity as an object, objects in Dost are special entities that coexist with 'smaller' entities such as integers, reals, and other data described by Pascal-like type declarations. Smalltalk classes share code through the mechanism of inheritance while Dost classes share code by using Mesa constructs for importing and exporting declarations. The destination of a message is determined at execution time in Smalltalk, but is bound at compile time in Dost. Finally, unlike Smalltalk, Dost does not support metaclasses.

An important difference between Dost and Smalltalk is in the way user interfaces are defined in the two systems. The user interface of a Smalltalk object has to be defined manually by the application programmer, while the user interface of a Dost object is implemented automatically by its dialogue managers. The job of manually implementing the user interface of an object is alleviated to some extent in Smalltalk by the mechanism of inheritance: The methods of existing classes can be used to implement the user interface of a new object. However, an application programmer is still concerned with implementing the details of those aspects of the user interace that are specific to the object. In particular, he is concerned with converting between the internal and visual representations of instance variables defined by the class of the object, and implementing editing of these variables. The Dost dialogue manager is able to automate most of this work by using the type declarations presentation in the Mesa code for the object to select default representations and editing commands.

Dost does not automate all aspects of the user interface. An object must specify the attributes of displayed variables and implement object-specific editor commands, but our experience suggests that these tasks are not code-intensive. Attributes provide a high-level language for specifying display properties of variables, and attribute inheritance provides reasonable default values. Moreover, an object needs to provide very few object-specific commands, and they are often easily defined in terms of other existing commands. Nonetheless, it would be useful if class inheritance could be used to reuse methods and attribute values defined in other classes.

6.5. Language-Oriented Editor Generators

Dost is closely related to the Synthesizer Generator [30], POE [6], ALOE [23,25], sds [8], PECAN [28], PSG [2] and other language-oriented editor (LOE) generators. An LOE generator provides a *specification language*, which may be used to define the syntax and semantics of a *target language*. The definition of a target language is used by the LOE generator to create an editor (LOE) for the language. Traditionally, LOEs have been used to edit programs written in conventional programming languages. However, they have also been used to edit other structures such as documents, a desk calculator, and the specification language itself.

There are several similarities between our approach and LOE generators. The target language description used by an LOE generator corresponds to a Dost class, the syntax tree maintained by an LOE corresponds to a Dost object,

and the LOE generator corresponds to a dialogue manager. The main differences arise from differing goals: LOE generators are designed to compose programs, while Dost is designed to interact with them. For example, LOE-based environments tend to be mixtures of standard programs that manipulate unstructured text and structure editors that manipulate syntax trees. Our approach, on the other hand, supports an environment in which all programs offer editing interfaces.

Moreover, our approach uses the type declarations and procedures of a Pascal-like language to describe the structure and semantics of edited data. As a result, the editor description language is an extension of a conventional general-purpose programming language. LOE specification languages, in contrast, are BNF grammar descriptions embellished with constructs for describing semantics such as action routines [23], attributes [30], attributes and action equations [18], and denotational definitions [2,27,38].

LOE generators based on attribute grammars [6,17,29,30] allow a programmer to specify the semantics of user interaction *declaratively*. This feature is useful for specifying the static semantics of a target language, since it relieves a programmer from the task of explicitly calling procedures that check related values for semantic consistency. It may also be easier to verify certain properties of a specification that is declarative rather than procedural.

On the other hand, since editor descriptions under our approach are extensions of conventional programs, they automatically include all features of the base language that aid the programming task. For instance, since a Dost class an extension of a Mesa program, it is strongly typed, can be composed of several modules, can be divided into interface and specification parts, can share code with other classes, and can define a large variety of data structures. LOE generators have not evolved to the stage where they provide equivalent facilities in the specification languages they support.

The architectures of Dost and LOE generators also differ in significant ways. Dost supports the separation of the syntax and semantics of an editor into an object and a dialogue manager, which communicate with each other through messages. Thus an object and a dialogue manager can reside on different machines, such as a mainframe host and a workstation. Moreover, an object can be connected simultaneously to several dialogue managers, allowing several users to view and edit the object at the same time, with each instantly seeing changes made by the others.

Finally, Dost objects can exchange messages to maintain consistency among related objects. An equivalent facility allowing sharing of information between different syntax trees is not currently provided by LOE generators.

6.6. AGAVE

Recently, Notkin [24,26] has proposed an environment called AGAVE that replaces standard programs with editor modules written in a language based on the ALOE specification language. He has augmented the ALOE specification language with primitives that allow sharing of code between different modules. He has also proposed capability-based addressing to allow sharing of data between different syntax trees.

Dost and AGAVE are both environments in which all interaction is through structure editor interfaces. However, there are two significant differences between the two systems, which stem from the fact that AGAVE is derived from ALOE. First, AGAVE offers a grammar-based specification language for describing editors, while Dost offers a

Mesa-based programming language.

Second, AGAVE replaces a traditional operating system kernel with an editor kernel, which implements the editing interfaces. The kernel is (dynamically) linked to all the editor modules in the system and acts as the controlling module of a monolithic program. Dost, on the other hand distributes control over all the objects and dialogue managers in the system. As a result, AGAVE can edit only one syntax tree at a time, while Dost allows multiple objects to be edited simultaneously. Moreover, AGAVE cannot offer the advantages of object-dialogue manager separation described in the previous section. On the other hand, the AGAVE kernel is responsible for activating and passivating the syntax trees in the system. Dost supports the simpler approach of making each object responsible for activating and passivating its data.

6.7. Voodoo

Voodoo [34] is a framework that supports the generation of editing interfaces in an object-oriented system. It is perhaps the closest in spirit of any of these systems to Dost. Both Dost and Voodoo (which were developed independently and contemporaneously) support editor-oriented interaction.

Voodoo divides the objects in the system into *emenands*, *images*, and *editors*. Each emenand is associated with with one or more images and one or more editors. An image consists of an abstract syntax tree, which describes the *external* structure of the emenand. The image is used by an editor to allow the user to interact with the emenand. A dialogue manager in Dost corresponds to an editor in Voodoo.

There are two main differences between the two systems: First, a dialogue manager in Dost is created automatically, while an editor in Voodoo is created manually using the primitives for inheritance offered by the host object-oriented system. Thus, while Dost offers more automation, Voodoo offers more flexibility, including support for graphical presentations.

Second, an emenand in Voodoo is associated with both an internal structure and an external structure. As a result, an emenand's internal structure can be changed without affecting the user's view of the object. However, the implementor of a new application has to be concerned with creating two structures and keeping them consistent. In Dost, a single set of type declarations in an object's class defines both the structure and representation of the object.

7. FUTURE WORK

We now discuss some useful features missing from Dost and outline possible ways to add them.

7.1. More Attributes

Attributes are used by an object to specify various characteristics of a displayed variable: the format, the commands available to edit its presentation, the update methods, and so on. A dialogue manager uses the "high level" description provided by these attributes to handle the "low level" details of user interaction.

One problem with using attributes is that a dialogue manager supports only a limited number of them, thus limiting flexibility in specifying user interfaces. For instance, Dost currently lacks a way to specify fonts and sizes of

characters, spacing between lines on the screen, or justification of text. Our hope is that a finite but large set of attributes will provide sufficient flexibility for most applications. Further research is needed to determine this set. An alternative would be to add a facility for defining new attributes, but it is not yet clear what such a facility should look like.

7.2. More Commands

The generality of the Dost interaction model stems from the large number (currently 30) of editing commands, including commands for editing windows, objects, text, structures, and attributes. Experience has shown that a typical object can use most of these commands and needs to provide very few object-specific commands.

However, the current set of commands is by no means exhaustive. Two important commands missing in Dost are **undo** and **redo**. Currently, an object that wishes to provide an "undo" facility must implement it itself. Further research, perhaps using the ideas presented in [6,21,42], is needed to define a general-purpose undo/redo facility for Dost.

Further work is also needed to determine other default commands that may be provided by a dialogue manager.

7.3. Specification of Attributes

Dost provides two ways to specify attributes: one for the applications programmer defining a class of objects, and another for a user interacting with a specific object. The application programmer specifies attributes with procedural code that sends attirbute-update messages to a dialogue manager. The user specifies attributes interactively with attribute-editing commands.

It would be useful if an application programmer could also specify attributes of attribute groups interactively. The programmer could create a "dummy" instance of a class, use attribute editing commands as described in §3.6 to experiment with different formatting attributes, and finally execute an **accept** command to "freeze" these attributes, thereby creating a class description with appropriate initial defaults.

It would also be useful if Dost allowed attributes to be specified declaratively, thus supporting definition of initial or constant values for attributes. For instance, to specify that the `author` fields of `references` should be horizontally aligned, the programmer might write

```
Reference:  RECORD [
    author: STRING ATTRIBUTES alignment = horizontal END,
    ...];
```

Further research is needed to determine how procedural, declarative, and interactive specification of attributes may be integrated.

7.4. Graphical Presentations

Dost supports only textual presentations of data structures. In many situations, it is useful to allow editing of graphical presentations of variables. For example, it might be nice to display variables of type

```
Time: TYPE = RECORD [
     hr: 0..23,
     min, sec: 0..59]
```

as the face of a clock, and let the user change the time by moving its hands. Another example is a *scroll bar*, a popular way of displaying and changing a scalar value.

This example also illustrates the problems associated with supporting graphical interfaces. How does a dialogue manager know that this data structure should be displayed as a clock? One approach would be to make the application programmer explicitly specify the graphical presentation as a bit map. This approach, while adequate for simple *icons*, would not allow the dialogue manager to support editing, since it would not know the structural composition of the presentation.

A better approach would be to define high-level graphical attributes that describe the structure of the image to the dialogue manager. For instance a number could be associated with attributes that determine if it is to be described by the length of a line segment, the angle between two line segments, the radius of a circle, and so on. Further work is necessary to determine if it is possible to define a general set of graphical attributes that captures the rich variety in graphical images.

7.5. Support of Pipe-Like Connections

The stream-based I/O of traditional systems allows programs to be joined together in *pipelines*. This idea has been used extremely fruitfully in to build new applications out of existing building blocks. The idea of pipe-like connections, however, seems inconsistent with the input/output model of our approach. A Dost application does not read and write information sequentially from input and output streams, instead it displays information in one or more presentations, and receives updates to changes in these presentation. Thus one-way communication between an application and an input or output stream, essential to defining pipes, is replaced with two-way communication between an application and its dialogue manager.

It would be possible to redirect messages meant for a dialogue manager to be delivered instead to another object, which would then respond like a dialogue manager. But interaction with a human user seems to call for a very different style of programming than exchanging messages with a program. Moreover, under our current approach such redirection is not possible since messages to a dialogue manager are preprocessed (for reasons described in § 4.1). Further research is necessary to accommodate pipe-like connections in our approach.

7.6. Applicability to Other Programming Languages

Our approach is tailored to Pascal-like languages, and although some components support programming paradigms common to most modern programming languages, others are more specifically tied to features of statically typed, procedural languages like Pascal or Mesa. We discuss below both kinds of components, and outline ways to extend our approach to a wider variety of languages.

The idea of associating formatting attributes with variables is fairly language-independent, as is support for extending input/output from predefined types to programmer-defined types. Automatic input/output of values,

however, depends on language facilities for describing types. Any value whose type can be declaratively described by the language can be formatted and parsed automatically by the dialogue manager, but data structures that must be implemented procedurally require more application-specific code to support editing. For example, FORTRAN has no support for defining record types, so a FORTRAN programmer might represent an array of records by a set of parallel arrays, one for each field. If the program wished to display one "record" it would have to display each field individually. Similarly, a Dost-like environment for LISP might a provide default presentation for lists, whereas lists implemented as linked lists in Pascal would require more specific code to support their display and editing. This problem becomes partiularly severe for a typeless language such as BLISS [44].

The lack of a rich set of data structuring primitives in a language may be overcome to some extent by supporting attributes that determine structural properties of displayed variables. For instance, a BLISS variable may be associated with the attribute *treatAsArray* to treat its l-value (address) as a pointer to an array whose dimension is determined by the attribute *dimension*. Variables whose *treatAsArray* attributes are `true` are input/output as arrays. Other similar attributes may be added to augment the set of data structures parsed/unparsed by a dialogue manager. Naturally, these attributes would be poor substitutes for a richer set of type declarations.

A related problem occurs in *dynamically typed* languages such as SNOBOL, APL, or Smalltalk. Although such languages are strongly typed, types are associated only with values, not variables. A dialogue manager for such a language would have no trouble displaying values (indeed, a major attraction of such languages is the ease of displaying values). However, if the program were to display an uninitialized variable, with the intention that the user fill in an appropriate value, the dialogue manager would have no information to guide the parsing of the value entered. For simple built-in types, the syntax of value entered could guide the type determination (for example, if a decimal point is included, the value is assumed to be real), but this approach does not extend naturally to programmer-defined types. More importantly, the type of value expected is often an important visual cue to the user. In dynamically-typed lanuages, information about the type expected is only implicit in the program flow.

One approach to accommodate such languages is to associate each variable with the attribute *type*, which determines the type of the variable for input purposes. This attribute may be set by the object to restrict the set of values input by the user. It may also be set by the user to declare his intention to input a value of a certain type. The dialogue manager can then use this attribute to provide "type-directed editing" of the value.

Our approach currently can only be used to edit data structures defined by Pascal-like types. We have not considered SNOBOL tables, abstract data types, polymorphic types, and other more complex data-structuring methods. However, we see no conceptual difficulties with extending the set of type constructors supported.

Finally, the editing paradigm offered by our approach supports the concept of repeated user modifications of variables. This concept appears to be at odds with with functional languages such as pure LISP or FP [1] that prohibit side effects such as the modification of the value of a variable. This difficulty in dealing with functional languages is not unique to our approach; the whole issue of input/output in functional languages is still an open problem being studied by researchers [43]

Perhaps no single approach is suitable to support all programming language paradigms. An alternative is a set of approaches, each tailored to a particular programming language paradigm. For instance, an approach supporting functional languages may consider editing of a presentation as a "macro" change of state instead of "micro" changes to individual variables. Further research is needed to study the augmentation or adaptation of our approach to support different language paradigms.

8. SUMMARY

This paper presents a new approach to automatic generation of user interfaces. The approach organizes information into objects, which communicate with each other through messages and with the user through editor-oriented interfaces. Between each user and an object is a dialogue manager, which provides the user with a default interface to edit the variables of the object. The default interface may be overridden by the object, using the mechanisms of attributes and attribute inheritance.

We have tested our approach by building the major parts of Dost. Preliminary experience with the system shows that the user interface code in an application is a very small part of the total code, and replaces thousands of lines of code that would be required to implement the interface manually.

In comparison to previous approaches, our approach

- automates both input and output of values of programmer-defined types,
- supports the editing model of interaction,
- is based on conventional programming languages,
- allows a user to interact with several applications at the same time,
- offers the advantages of object/dialogue manager separation, and
- allows an implementor to use a single description of data structures for display, entry, modification, and semantic processing.

Further research is needed to study general sets of attributes and default editing commands, interactive and declarative specification of attributes, graphical presentations, support of pipelines in an editing environment, and applicability of the basic elements of the approach to a diverse range of programming languages.

REFERENCES

1. John Backus, "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs.," *CACM* 21(8)(August 1978).
2. Rolf Bahlke and Gregor Snelting, "The PSG - Programming System Generator," *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Languages*, pp. 28-33 (June 1985).
3. Prasun Dewan and Marvin Solomon, "An Approach to Generalized Editing," *Proceedings of the IEEE Ist International Conference on Computer Workstations*, pp. 52-60 (November 1985).
4. Prasun Dewan, "Automatic Generation of User Interfaces," Ph.D. Thesis and Computer Sciences Technical Report #666, University of Wisconsin-Madison (September 1986).
5. Prasun Dewan and Marvin Solomon, "Dost: An Environment to Support Automatic Generation of User Interfaces," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software*

Development Environments, SIGPLAN Notices 22(1) pp. 150-159 (January 1987).

6. C. N. Fischer, Gregory F. Johnson, Jon Mauney, Anil Pal, and Daniel L. Stock, "The POE Language-Based Editor Project," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 21-29 (April 1984).
7. Christopher W. Fraser, "A Generalized Text Editor," *CACM* 23(3) pp. 154-158 (March 1980).
8. Christopher W. Fraser, "Syntax Directed Editing of General Data Structures," *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* 16(6)(June 1981).
9. Christopher W. Fraser and A. A. Lopez, "Editing Data Structures," *ACM Transactions on Programming Languages and Systems* 3(2) pp. 115-125 (April 1981).
10. C.W. Fraser and D.R. Hanson, "A High-Level Programming and Command Language," *Sigplan Notices : Proc. of the Sigplan '83 Symp. on Prog. Lang. Issues in Software Systems* 18(6) pp. 212-219 (June 1983).
11. C.W. Fraser and D.R. Hanson, "High-Level Language Facilities for Low-Level Services," *Conference Record of POPL*, pp. 217-224 (1984).
12. David B. Garlan and Philip L. Miller, "GNOME: An Introductory Programming Environment Based on a Family of Structure Editors," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 65-72 (April 1984).
13. Adele Goldberg and David Robinson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass. (1983).
14. Adele Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Mass. (1984).
15. R.E. Griswold and M.T. Griswold, *The Icon Programming Language*, Prentice Hall, Englewood Cliffs, NJ (1983).
16. D.R. Hanson and R.E. Griswold, "The SL5 Procedure Mechanism," *Comm. ACM*, pp. 392-400 (May 1978).
17. Gregory F. Johnson, "An Approach to Incremental Semantics," Ph.D. Thesis, University of Wisconsin - Madison (August 1983).
18. Gail E. Kaiser, "Generation of Run-Time Environments," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 51-57 (June 1986).
19. G. D. Kimura, "A Structure Editor for Abstract Document Objects," *IEEE Transactions on Software Engineering* 12(3) pp. 417-436 (March 1986).
20. J.M. Lafuente and D. Gries, "Language Facilities for Programming User-Computer Dialogues," *IBM J. Res. Develop.* 22(2) pp. 145-158 (March 1978).
21. George B. Leeman, Jr., "A Formal Approach to Undo Operations in Programming Languages," *ACM Transactions on Programming Languages and Systems* 8(1) pp. 50-87 (January 1986).
22. Michael Marcotty, Henry F. Ledgard, and Gregor V. Bochmann, "A Sampler of Formal Definitions," *Computing Surveys* 8(2) pp. 194-275 (June 1976).
23. Raul Medina-Mora, "Syntax-Directed Editing: Towards Integrated Programming Environments," PhD Thesis, Department of Computer Science, Carnegie-Mellon University (March 1982).
24. David Notkin, "Interactive Structure-Oriented Computing," PhD Thesis and Technical Report, CMU-CS-84-103, Department of Computer Science, Carnegie-Mellon University (February 1984).
25. David Notkin, "The Gandalf Project," *The Journal of Systems and Software* 5(2)(April 1985).
26. David Notkin, "Sharing and Modularization in Structure Editing Environments," *Proceedings of the 19th Hawaii International Conference on Systems Sciences*, (January 1986).
27. Anil Pal, "Generating Execution Facilities for Integrated Programming Languages," Ph.D. Thesis and Technical Report #676, University of Wisconsin-Madison (December 1986).
28. S. P. Reiss, "PECAN: Program Development Systems that Support Multiple Views," *IEEE Transactions on Software Engineering* SE-11(3)(March 1985).
29. Thomas Reps, Tim Teitelbaum, and Alan Demers, "Incremental Context-Dependent Analysis for Language-Based Editors," *ACM Transactions on Programming Languages and Systems* 5(3) pp. 440-477 (July 1983).

30. Thomas Reps and Tim Teitelbaum, "The Synthesizer Generator," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 42-48 (April 1984).

31. Lawrence A. Rowe and Kurt A. Shoens, "A Form Application Development System," *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, pp. 28-38 (1982).

32. L. A. Rowe and K. A. Shoens, "Programming Language Constructs for Screen Definition," *IEEE Transactions on Software Engineering* SE-9(1) pp. 31-39 (January 1983).

33. Lawrence A. Rowe, "'Fill-in-the-Form' Programming," *Proceedings of VLDB*, pp. 394-404 (1985).

34. Jeffrey Scofield, "Editing as a Paradigm for User Interaction," Ph.D. Thesis and Technical Report No. 85-08-10, University of Washington, Department of Computer Science (August 1985).

35. M. Shaw, E. Borison, M. Horowitz, T. Lane, D. Nichols, and R. Pausch, "Descartes: A Programming-Language Approach to Interactive Display Interfaces," *Sigplan Notices : Proc. of the Sigplan '83 Symp. on Prog. Lang. Issues in Software Systems* 18(6) pp. 100-111 (June 1983).

36. M. Shaw, "An Input-Output Model for Interactive Systems," *CHI'86 Proceedings*, pp. 261-273 (April 1986).

37. David Canfield Smith, Charles Irby, Ralph Kimball, Bill Verplank, and Eric Halsem, "Designing the Star User Interface," *BYTE* 7(4)(April 1982).

38. Gregor Snelting, "Unification in Many-Sorted Algebras as a Device for Incremental Semantic Analysis," *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 229-235 (January 1986).

39. J. Sutton and R. Sprague, "A Study of Display Generation and Management in Interactive Business Applications," Tech. Rept. RJ2392(#31804), IBM San Jose Research Laboratory (November 1978).

40. Richard E. Sweet, "The Mesa Programming Environment," *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments*, pp. 216-229 (June 1985).

41. Tim Teitelbaum, Thomas Reps, and Susan Horwitz, "The Why and Wherefore of the Cornell Program Synthesizer," *Sigplan Notices* 16(6)(August 1981).

42. Jeffrey Scott Vitter, "USER: A New Framework for Redoing," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 168-176 (April 1984).

43. John H. Williams and Edward L. Wimmers, "Sacrificing Simplicity for Convenience: Where Do You Draw the Line?," *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 169-179 (January 1988).

44. W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, New York, N. Y. (1975).

45. Marvin V. Zelkowitz, "A Small Contribution to Editing with a Syntax Directed Editor," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 1-6 (April 1984).

46. Marvin V. Zelkowitz, Jennifer Elgot, David Itkin, Bonnie Kowalchack, and Michael Maggio, "The Engineering of an Environment on Small Machines," *Proceedings of the IEEE Ist International Conference on Computer Workstations*, pp. 61-69 (November 1985).