A TOUR OF THE SUITE USER INTERFACE SOFTWARE

Prasun Dewan
Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907

ABSTRACT

Suite offers several advanced interactive features missing from contemporary interactive systems including a generic direct-manipulation user interface; a flexible input model offering benefits of both incremental and delayed feedback; customizable system-provided dialogue managers which relieve applications from managing their user interfaces; loose physical coupling between an application and its dialogue manager, that is, execution of these components in different address spaces, residing possibly on different hosts; interactive specification of customizable properities of user interfaces; and IS-A and IS-PART-OF inheritance to reduce the effort required to specify these properities. It complements recent work done in programming languages, databases, operating systems, and distributed systems. In this paper, we take the reader on a tour of the Suite user interface software, highlighting its distinguishing features.

1. INTRODUCTION

Suite is being developed at Purdue for reducing the effort required to create interactive applications. It offers a high-level interaction model supporting several properties of direct manipulation [17] including continuous display of objects of interest, manipulation of objects by "physical operations" (e.g. selection by a pointing device), and incremental response to operations on these objects. It supports flexible incremental response, that is, allows users to dynamically choose when a certain kind of feedback is received.

An application's interaction with the user is managed by a customizable system-provided *dialogue manager*, which corresponds to a UIMS in the Seeheim reference model [16]. The dialogue manager offers the application the abstraction of an "active value" [15], that is, a value displayed on the screen whose visual representation can be updated by the application to display results and edited by the user to control the execution of the application. Active values in Suite include not only simple values such as integers and reals but also structured values such as records and sequences. The application specifies only high-level aspects of its user interface including the structure and presentation of active values, validation routines defining constraints on these values, and update routines for reacting to user changes to them. The dialogue manager handles all other aspects of the application's interaction with the user.

Suite provides an inheritance model for reducing the effort required to specify user interface properties or *attributes* of active values. The model classifies the active values of an application into several *value groups* and allows an attribute to be defined once for all members of a group. It arranges these groups in multiple intersecting hierarchies based on the IS-A and IS-PART-OF relationships among values and allows a subgroup to inherit attributes from its supergroups. Attributes of value groups may be specified both procedurally and interactively.

Suite provides an object layer supporting distributed, shared, and persistent objects. Suite objects are similar to objects supported by other distributed object-based systems such as Eden [1] and Clouds [5] in that they execute in separate address spaces residing possibly on different hosts. However, unlike objects supported by these systems, Suite objects are compatible with conventional (UNIX-like) operating systems. In particular, they can coexist with, access, and be accessed by existing components of a conventional operating system; and are named, organized, and accessed like conventional files. The object layer facilitates loose physical coupling between application and its dialogue manager by allowing them to be be created as separate objects. Loose physical coupling between an application and its UIMS is a generalization of the widely-accepted practice of logically separating them [16, 18] and allows, for instance, the user interface of an application to execute remotely and be dynamically changed. The object layer also facilitates collaborative applications by allowing them to be created as a set of objects interacting with different users and communicating with each other to allow the users to share results in real-time.

The structure of internal, active, and persistent values of an object is described by the type declarations of a conventional programming language. Thus, Suite extends the idea of a database programming language [2] by offering an application programmer a unified programming, database specification, and user-interface specification language.

The current version of Suite has been implemented on a network of workstations executing UNIX, TCP/IP, X, and Sun NFS (Network File System). It uses X for displaying windows, and TCP/IP and NFS for communication among and naming of applications and dialogue managers.

A multiple inheritance model supporting both IS-A and IS-PART-OF of inheritance distinguishes Suite from Incense [14], Planit [11], APT [13] and other systems supporting flexible

This research was supported in part by the National Science Foundation sponsored Software Engineering Research Center at Purdue.

displays of data structures. Inheritance-based interactive specification of displays of values distinguishes it from GRINS [12], Peridot [15], DialogEditor [4], Serpent [3] and other systems supporting interactive specification of user interfaces. This feature was also supported in Suite's predecessor, Dost [8], and has been refined in Suite. The input model, loose physical coupling between an application and its UIMS, and integration with operating systems, distributed systems, and database programming languages separates it from other user interface software we know of.

In other papers and reports we have discussed in detail individual aspects of Suite including integration with programming languages [9], the inheritance model [6], the design and implementation of the object layer [7], and our experience using the object layer to create interactive applications [10]. This paper gives a high-level overview of the Suite user interface software. It uses the example of a prototype project management tool to illustrate and motivate the distinguishing features of Suite.

The remainder of the paper is organized as follows. Section 2 describes an example session with the tool, illustrating the system from the point of view of end-users. Section 3 explains how the tool is specified, illustrating the system from the point of view of application-programmers. Section 4 illustrates facilities provided by Suite for interactively specifying properties of the user interface. These can be used by application-developers to define an initial user interface and end-users to customize it. Finally, Section 5 presents conclusions and directions for future work.

2. USER INTERFACE

In this section, we illustrate the user interface of Suite using the example of the prototype project management tool we have built for managing some of the activities of the Purdue-Florida Software Engineering Research Center (SERC). The tool consists of three applications: a project object which manages a list of projects, an affiliate object which manages a list of affiliates, and a budget object which maintains budgetary information including the contribution of each affiliate and the money allocated to each project. These applications communicate with each other using remote procedure calls and with users through dialogue managers.

Assume that the budget, project, and affiliate objects have been created on the host `medusa.cs.purdue.edu` under the names `budget`, `projects`, and `affiliates`, respectively, in the directory `/u13/rxc/proj/demo`, and we wish to interact with them from the host `arthur.cs.purdue.edu`. We first create a dialogue manager on `arthur.cs.purdue.edu`. The dialogue manager displays a *startup window* (Figure 1) which allows users to name the object with which they wish to interact. Let us specify the name of the budget object in the startup window and execute the `Load` command. In response, the dialogue manager loads the active values of the budget object in the *object window* and provides a *menu window* for editing these values (Figure 2).

Currently, no projects or affiliates have been assigned to SERC, therefore the fields `affiliate_budgets` and `project_budgets` in Figure 2 are empty. These data

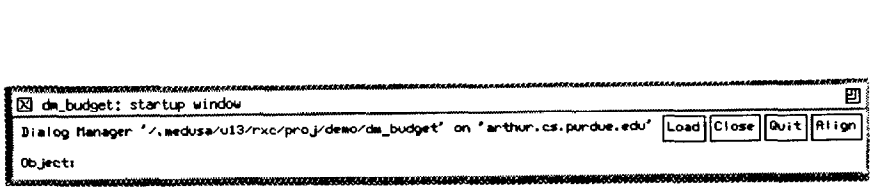

Figure 1. The Startup Window

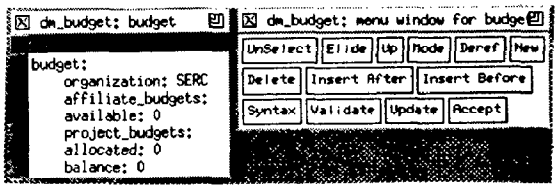

Figure 2. Object and Menu Windows

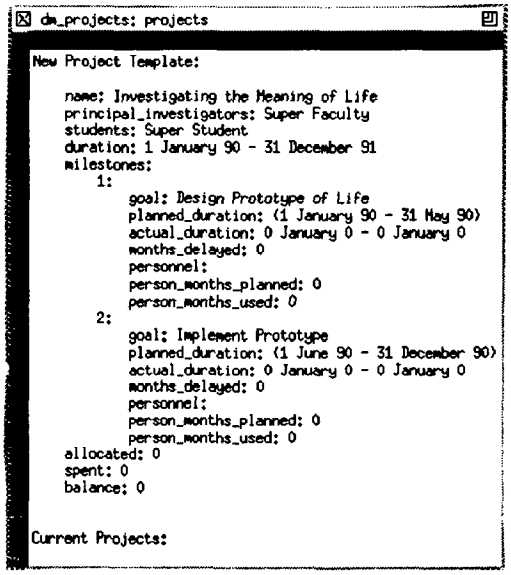

Figure 3: The Project Application

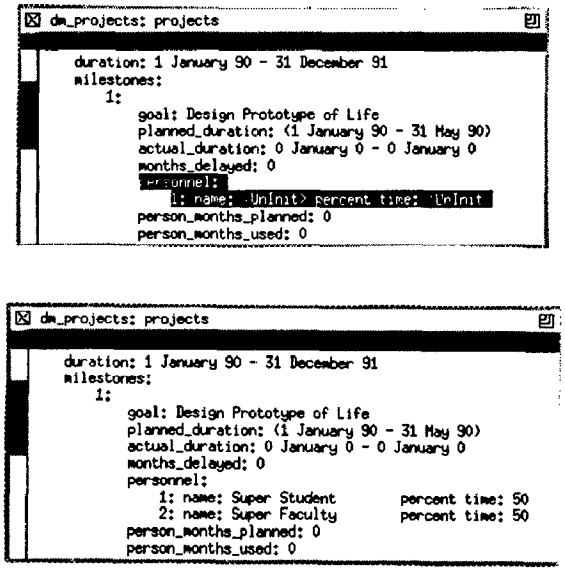

Figure 4: Structure-based Commands

structures cannot be edited directly to add new projects or affiliates, since they are managed by the project and affiliate objects, respectively. To interact with the project object, we create another dialogue manager and load the object. The object window of the project object displays two top-level data structures: a project template, which can be edited to add a new project, and a list of current projects (Figure 3).

As illustrated by Figures 2 and 3, Suite offers a form-like interface providing prompts and default values for fields. Like contemporary form editors, it offers mouse- and menu- based commands for editing these values. In addition, it supports commands that understand the structure of displayed data. For instance, it supports the `Up` command for selecting the parent of the currently selected data structure and the `Elide` command to hide the fields of a structure value. Moreover, it allows new fields to be added in an object window. For instance, in Figure 4, we have used the `Insert After` command to add new subfields in the `personnel` field of the object window of the project object.

We now execute the `Accept` command to "commit" the template. At this point several changes to the display occur (Figure 5). In the object window of the project object, the new project has been added to the list of current projects. Moreover, the "dependent" fields of the project have been automatically computed for the user. For instance, the `person_months_planned` field has been computed based on the information in the `personnel` field. Similarly, the `allocated`, `spent`, and `balance` fields have been computed for the user. In the object window of the budget object, a new entry is made in the list of current project budgets, and the total allocated amount and the balance are recomputed based on the amount allocated to the new project.

In general, a dialogue manager does the following when the user executes the `Accept` command. It first begins a *syntax phase*, when, for each value whose presentation has been changed, it checks the presentation for syntax errors based on the type of the value, parses the value if it finds no syntax error, and marks the value as *parsed*. If it finds no error in the syntax phase, it begins a *validation phase*, when it checks each of the parsed values for semantic errors by invoking the *verify procedure* associated with the value, and if it finds no error, marks the value as *validated*. Finally, if it finds no error in the validation phase, it begins the update phase, when it commits each of the validated values by invoking the *update* procedure associated with the value, which can take (possibly irreversible) semantic actions such as updating the display of other active values and sending messages to other objects. A value is considered changed if it has been directly edited by the user or it is a parent of a changed value. Values are visited in postorder in each of these phases, that is, the children of a structure value are visited before their parent is visited. In the remainder of this paper, we shall refer to the feedback given by the syntax, validation, and update phases as syntactic, semantic, and update feedback, respectively.

Figure 6 shows the response of the `Accept` command when we try to commit erroneous values. No semantic actions are taken since the update phase was not invoked.

By executing the `Accept` command, we received the highest *degree of feedback*, since the system went through all three phases: syntax, validation, and update. It is possible to receive lower degrees of feedback. The `Validate` command goes through only the syntax and validation phases, while the `Syntax` commands goes through only the syntax phase.

Finally, to complete this example, we connect the affiliate object to a dialogue manager and commit a new affiliate (Figure 7).

In this example, the project management tool was used in the "single-user mode" since the dialogue managers of all three objects were executed by a single user on the same host. In general, they can be executed by multiple users and on multiple hosts. However, currently, only one dialogue manager (and hence only one user) can interact with a particular object at any one time.

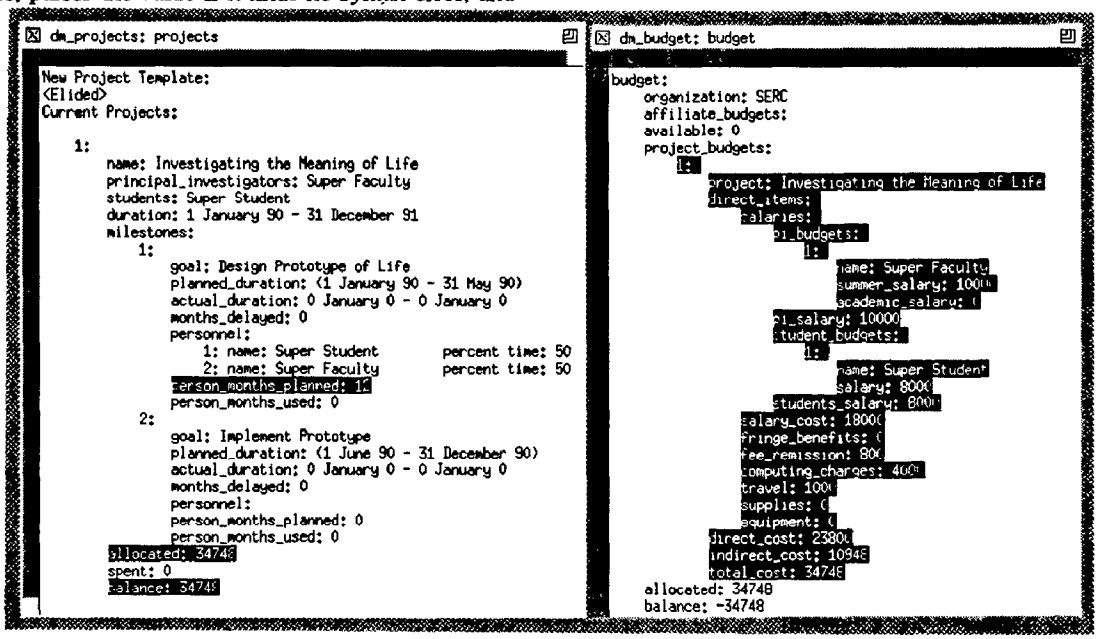

Figure 5: Accepting the Project Template

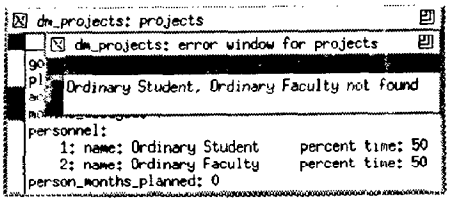

Figure 6: Errors

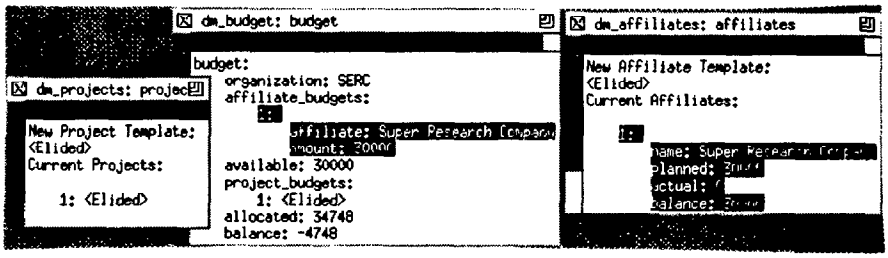

Figure 7: The Affiliate Application

3. PROCEDURAL INTERFACE

In Suite, interactive applications are created as *objects*. An object is like a traditional application in that it is described by an application program declaring data structures and the procedures that manipulate them. However, several important differences distinguish it from a traditional application: It is associated with *remote procedures* (also called *methods*) and *handlers* instead of a main procedure. Like a file, it is associated with a hierarchical file name. A client application may *open* it to get an *object descriptor* that refers to it, use the descriptor to invoke a sequence of remote procedures in it, and later *close* it. An object executes the *initialization handler* when it is created and the *termination handler* when it is removed from the system. Some of its data structures can be edited by the user to control the behavior of the object. Some of its data structures can also be declared to be persistent— these are checkpointed on persistent storage. At any time, it is in the active or passive state. In the active state, it may be in the referenced or unreferenced state. It is in the referenced state if some other object has it open for communication, and unreferenced state otherwise. When an object stays in the unreferenced state for more than a certain period of time, it is automatically passivated and its persistent data structures are saved on disk. A passive object is activated and its persistent data structures restored when it is next opened by some client. In the referenced state, an object may be in the loaded or unloaded state. In the loaded state, it has been opened by a dialogue manager, which can be used to interact with it. It can define *load* and *unload* handlers to process transitions to and from the loaded state.

Figure 8 shows an outline of a C[1]-based program describing the budget object. Like a traditional program, it defines data structures such as `budget` and procedures such as `AddAffiliate` which manipulate these data structures. It also contains special comments beginning with the words `oc` and `dmc`, called *object* and *dialogue annotations*, respectively, which are processed by the Suite *object* and *dialogue compilers*, respectively. The object annotations

Initiate with `Initiate`

tells Suite that `Initiate`, which initializes `budget`, is the initialization handler. The annotations

Method `AddProject`
Async Method `AddAffiliate`

tell Suite that `AddProject` and `AddAffiliate` are remote procedures that can be called synchronously and asynchronously, respectively. The former is invoked by the project object to inform the budget object about a new project and returns the

```
/*oc
    Initiate with Initiate
    Method AddProject
    Async Method AddAffiliate
    Eternal budget
*/
/*dmc
    Editable Budget
    Load with Load
*/
/* Declarations */
    ...
typedef struct {
    String organization;
    AffBudgets affiliate_budgets;
    int available;
    ProjectBudgets project_budgets;
    int allocated;
    int balance;
} Budget;
Budget budget;

void Initiate ()
    { ... }
Load()
    { ... }
void AddAffiliate ( aff )
    Affiliate aff;
    { ... }
int AddProject ( proj )
    Project proj;
    { ... }
void UpdateProjectBudget
    ( path, budget_ptr )
    char *path;
    ProjectBudget *budget_ptr;
    { ... }
/* Other Procedures */
    ...
```

Figure 8. The Budget Application Program

[1]Currently, the Suite implementation supports only C, but the design allows the addition of other Pascal-like languages.

amount allocated to the project, while the latter is invoked by an affiliate object to inform the budget object about a new affiliate. The affiliate object does not wait for completion of the procedure since it has been declared to be asynchronous.

The following program segment extracted from the affiliate object illustrates how a remote procedure is invoked by a client:

```
Affiliate *aff_ptr;
   ...
budget_obj = OpenObject("budget");
   ...
AddAffiliate (budget_obj, aff_ptr);
   ...
CloseObject (budget_obj);
   ...
```

The client uses the file name of the budget object to get a descriptor to the budget object and uses it to invoke the remote procedure `AddAffiliate`.

In Figure 8, the annotation

Eternal `budget`

tells Suite that the variable `budget` is persistent and should be checkpointed on secondary storage. The annotation

Editable `Budget`

tells Suite that variables of type `Budget` are editable. The annotation

Load with `Load`

tells Suite that the procedure `Load` in the object program is the load handler. The definition of `Load` follows:

```
Load()
{
    Dm_Submit( &budget, "budget",
               "Budget" );
    Dm_SetAttr( "Type: ProjectBudget",
                AttrUpdateProc,
                UpdateProjectBudget );
    Dm_Engage( "budget" );
}
```

It consists of invocations of three different remote procedures defined by the dialogue manager. The call to `Dm_Submit` submits the variable `budget` for editing to the dialogue manager. The first argument specifies the location of the value of the variable, the second argument the name of the variable, and the third argument the name of the type of the variable (which must be one of the types that has been declared to be editable). The call to `Dm_SetAttr` specifies the value of an attribute of a value group which is either a type or a submitted variable (also called a *view* in Suite). In this example, the call defines `UpdateProjectBudget` as the value of the `AttributeUpdateProc` attribute of the type `ProjectBudget`. The attribute `AttributeUpdateProc` of a type specifies the procedure to be called when a value of that type is edited by the user. Finally, the call to `Dm_Engage` asks the dialogue manager to *engage* the variable, that is, display it to the user.

Here is the update procedure `UpdateProjectBudget`, which is invoked when a new value of type `ProjectBudget` is committed by the user.

```
void UpdateProjectBudget ( path,
                           budget_ptr )
     char *path;
     ProjectBudget *budget_ptr;
{
   int  change;
   change = - budget_ptr->total_cost;
   ComputeNewBudget (budget_ptr);
   change += budget_ptr->total_cost;
   budget.allocated += change;
   budget.balance -= change;

   Dm_Update( path, "ProjectBudget",
              budget_ptr );
   Dm_Update( "budget.allocated",
              "int", &budget.allocated );
   Dm_Update( "budget.balance", "int",
              &budget.balance );
}
```

Like other update procedures, it takes two arguments: the first, `path`, indicates the *path name* of the variable that was edited by the user (Suite associates each displayed variable (including descendents of top-level variables submitted by an object) and its type with a unique path name)) while the second, `budget_ptr`, gives the address of the new value for the variable. The procedure assumes that the user changes only the independent fields of the project budget and calls `ComputeNewBudget` to calculate the new values of dependent fields such as `total_cost`. It then calls the remote procedure, `Dm_Update`, in the dialogue manager to ask it to update the display of the project budget. Changing the total cost of a project budget also affects the value of the total amount allocated to the various projects and the balance left. The update procedure computes new values of these fields and calls `Dm_Update` to update their displays.

In order to communicate data among objects, save them on persistent storage, and display them, Suite needs to know the type of the data. This presents a problem in weakly-typed languages such as C since the type of a data structure is often ambiguous. For instance, a variable declared as

```
char *ptr
```

may be a pointer to a character, character array, or string. Suite lets a programmer explicitly specify the type of such data. Moreover, it uses several rules to determine types of values in the absence of explicit programmer specification. One important rule lets it support *sequences*— variable length arrays missing from C, Pascal, and older languages. Suite lets a sequence be simulated by a record containing a length field and an array pointer. For instance, it lets a sequence of values of type `ProjectBudgets` be simulated by the record:

```
typedef struct {
    unsigned num_projects;
    ProjectBudget *pb_arr;
} ProjectBudgets;
```

When transmitting such a record as an argument to a remote procedure or saving it in persistent storage, Suite sends or saves, respectively, only as many elements of the array as are specified by the first field. Moreover, when displaying the record, it regards the record as a list and allows elements to be inserted into and deleted from the array field, updating the length field appropriately. Sequences are used extensively in all the sample

objects built by us, including the budget, project, and affiliate objects.

An object is created by invoking its program with a special argument describing its file name. Thus the command

```
budget -name budget &
```

may be used to create the object `budget` which executes the object file `budget`. The name argument is not necessary when the name of the new object is the same as the program it executes. We distinguish between the names of objects and the programs they execute since, in general, a program may be executed to create multiple concurrent objects. For instance, the budget program may be executed to create the objects `purdue_budget` and `ufl_budget` for managing the Purdue and Florida SERC budgets, respectively.

4. CUSTOMIZATION

In Suite, the user interface available to interact with an object has three main components: (1) a *generic component* shared by all user interfaces, which includes object-independent commands such as `DeleteCharacter`, names of object-dependent commands such as `InsertAfter`, and facilities such as buttons in the menu window for invoking (object-dependent/independent) commands, (2) a *default object-specific component* that includes the default implementation of object-dependent commands and is "driven" by the type declarations in the program describing the object, and finally, (3) a *customized component*.

We have so far seen mainly the first two components. We did see one example of customization—the `Dm_SetAttr` call in Figure 8, which defines the update procedure invoked by `Accept`. It is possible to customize several other parts of the user interface. Moreover, it is possible to interactively customize the user interface, thereby avoiding the edit-compile-execute cycle required by procedural customization. In this section, we illustrate the customizable aspects of a user interface and how they can be specified interactively. Procedural specification of these parameters is straightforward, and is done through `Dm_SetAttr` calls.

Interactive customization is done through an *attribute window*, which allows a user to specify attributes of value groups. This window displays four fields: `Message`, which displays a message from the dialogue manager, `Path`, which names a value group, `Attr`, which names an attribute, and `Value`, which contains a value for the attribute. It also provides seven commands: `Load`, `Set`, `Inherit`, `Delete`, `Path`, `Attr Menu`, and `Enum Menu`. The `Load` and `Set` commands are used to load and define the value of an attribute of a value group, respectively. The `Inherit` command is like `Load` except that it looks for an inherited value if the attribute is not defined in the specified value group. The `Delete` command deletes the definition of the attribute of the value group. The `Path` command lets a user select a value group instead of typing its name. It loads the path of the selected data structure in the presentation window in the `Path` field of the attribute window. The `Attr Menu` command displays a menu of attribute values from which a value can be selected to set the `Attr` field. Finally, the `Enum Menu` command shows the set of legal values for an enumeration attribute.

Let us try changing an attribute of a value. We can execute the `Attr Menu` command to display the list of attributes (Figure 10), and select one of them, say `ElideString`. We need to also specify the path of a value whose attribute we wish to change. Therefore, we select the value in the object window and execute the `Path` command in the attribute window to display the name of the value in the `path` field (Figure 11). Now we enter a valid string in the `Value` field and execute the `Set` command. Next time we elide the project, the new value of `ElideString` is displayed (Figure 11).

Let us try the same process with some other attributes. For instance, let us change the `Titled` attribute of the project to `False` (Figure 12). Notice that the `TitleString` "1: " disappears from the display. Notice also that all fields of the data structure (except the milestone fields) have become untitled. This is because Suite defines *structural inheritance*, which uses the IS-PART-OF relationship to let a child inherit attribute values from its parent. To restore the titles of the fields of the project (but not the project itself) we can set the `Titled` attribute of `View Child: ((projects) [0])` to `True`.

Suite also defines *type inheritance*, which uses the IS-A relationship to let a value inherit attribute values from its type. The reason that the milestone fields did not become untitled in the previous example is that the type `Milestones` had defined the `Titled` attribute to be `True`. We can set it to `False` to untitle values of type `Milestones` and their descendents (because of structural inheritance), as shown in Figure 13.

The structural and inheritance schemes are two independent schemes which can both influence the attributes of a value. Suite resolves the multiple inheritance problem for most attributes such as `TitleString` by giving the type inheritance higher precedence than structural inheritance. It also supports "structure-first", "type-only", "structure-only", and "no-inheritance" search schemes [6].

Not all attributes of a value determine the format of its presentation. For instance, the `Erroneous` attribute determines if the value is semantically invalid, and the `ErrorString` attribute determines the error message that is displayed when a value is found to be erroneous. Two important attributes determine how soon users get feedback from dialogue managers and objects in response to changes they make to values. To illustrate, let us change the `IncInputScheme` attribute of the of the first project budget in the budget window to the enumeration value `InputUpdate`. When we next change a descendent of the data structure such as the `summer_salary` field, the values of dependent items are recomputed on every keystroke (Figure 14). This is because the `InputUpdate` input scheme asks for implicit invocation of the update phase on every keystroke. We can set this attribute to `InputValidate`, `InputSyntax`, or `InputDelay`, to receive semantic, syntactic, or no feedback, respectively, on every keystroke.

Suite also lets a user postpone feedback until the value has been "completely" edited without requiring explicit invocation of commands. It uses movement of the insertion point away from the edited field as an implicit indication of completion of the editing of the field. We can change the value of the `IncInputScheme` attribute to `InputDelay` and the value of the `MovementInputScheme` to `IncUpdate`. Since the input scheme is `InputDelay`, the update procedure is not invoked on every keystroke. However, since the movement scheme is `InputUpdate`, it is invoked as soon as the user moves the insertion point away from the edited field. Similarly, we can set the value of this attribute to `InputValidate`, `InputSyntax`, or `InputDelay` to receive semantic, syntax, and no feedback, respectively, on movement of the insertion point. These two attributes support great flexibility in choosing the kind of feedback given by the system. For instance, a user can choose

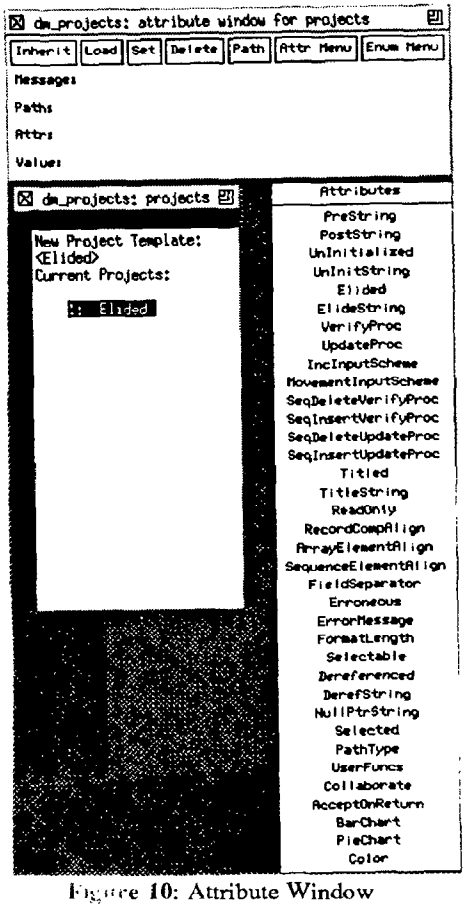

Figure 10: Attribute Window

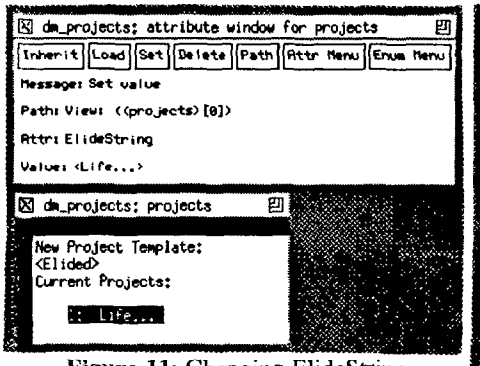

Figure 11: Changing ElideString

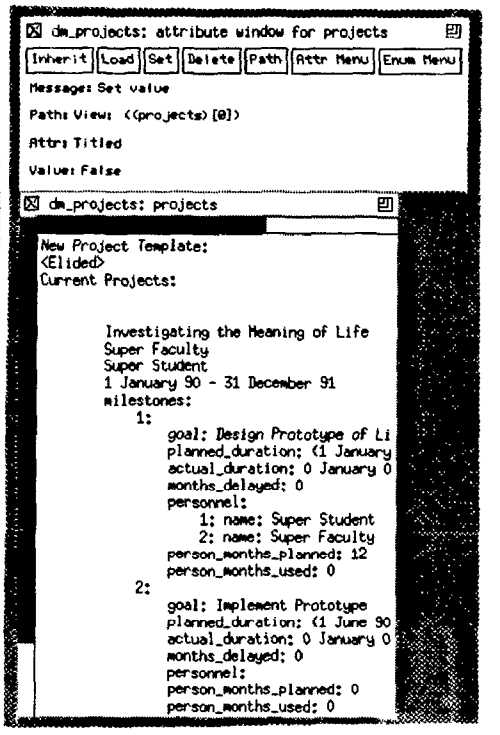

Figure 12: Structural Inheritance

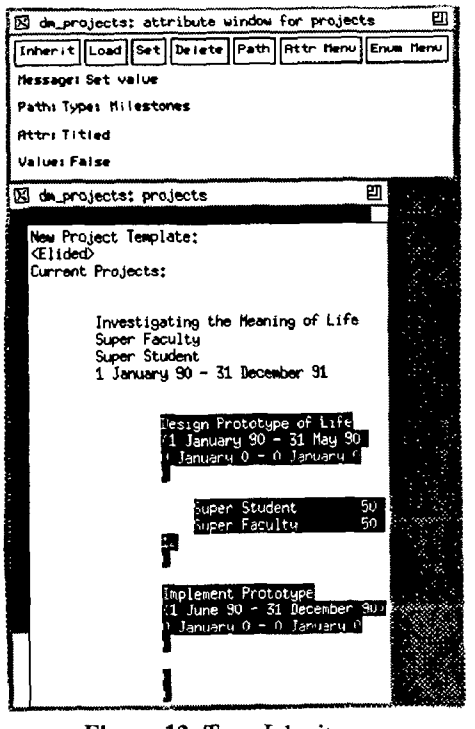

Figure 13: Type Inheritance

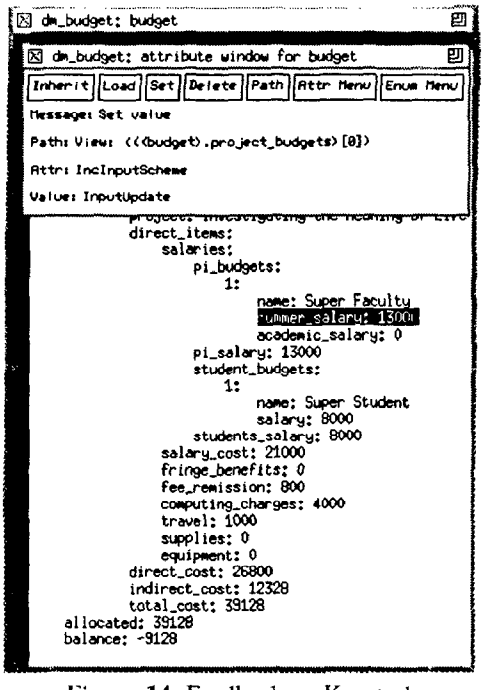

Figure 14: Feedback on Keystroke

to get syntactic feedback on every keystroke, semantic feedback on movement of the cursor point, and update feedback only on

explicit execution of the `Accept` command, by setting the `IncInputScheme` attribute to `InputSyntax`, and `MovementInputScheme` to `InputValidate`.

This section illustrates facilities provided by Suite for interactive specification of user interfaces. These features can be used by the end-users to customize the user interface defined by the application programmer. More importantly, they can also be used by the application programmer to interactively define the user interface.

5. CONCLUSIONS AND FUTURE WORK

Suite is part of a response to a general need for reducing the effort required to implement interactive applications. It extends previous work by supporting (i) an input model that provides users and programmers the flexibility of choosing when a particular kind of feedback is given in response to the modification to an active value, (ii) an inheritance model supporting both IS-A and IS-PART-OF inheritance, (iii) loose physical coupling between an application and its UIMS, and (iv) integration of user interface software with operating systems, distributed systems, and database programming languages. In this paper, we have motivated and highlighted the distinguishing features of Suite by describing in detail a simple, realistic application that uses them.

We have used our implementation for building user interfaces of several experimental applications. In addition to the project management tool, we have built a simple "process tool" which displays the list of current processes on a particular host. A user can edit the process list to delete processes from that system. Similarly, we have built a "line printer tool" which displays the current line printer queue to the user. We have also built a distributed multi-user calendar service, which allows users to enter and display appointments; a prototype student database application; and a distributed multiuser "accounting service", which displays common expenditures of a group of users.

We intend to explore how multiple dialogue managers can be attached to objects and address the associated issues of concurrency control and access control. Moreover, we plan to explore attributes for specifying flexible graphical displays of data structures.

ACKNOWLEDGEMENTS

Jeff Hostetler, Ray Humphrey, Dan Longacre, and Jyh-Jong Tsay implemented an early version of the dialogue manager. Eric Vasilik implemented most of the current version of the Suite prototype. Rajiv Choudhary is involved in extending it and measuring and improving its performance. Harry Duin, Joe Heim, and Harlene Sepulveda helped us test its implementation. Harlene Sepulveda built the line printer tool.

REFERENCES

[1] G.T. Almes, A.P Black, E.D. Lazowska, and J.D. Noe, "The Eden System: A Technical Overview," *IEEE Transactions on Software Engineering* 11:1 (January 1985), pp. 43-59.

[2] Malcolm P. Atkinson and O. Peter Buneman, "Types and Persistence in Database Programming Languages," *ACM Computer Surveys* 19:2 (June 1987).

[3] Len Bass, Erik Hardy, Reed Little, and Robert Seacord, "Incremental Development of User Interfaces," *Proceedings of IFIP TC2/WG 2.7 Working Conference on Engineering for Human-Computer Interaction, Napa Valley, August 1989*, North-Holland, 1990, pp. 155-176.

[4] Luca Cardelli, "Building User Interfaces by Direct Manipulation," *ACM SIGGRAPH Symposium on User Interface Software and Technology*, October 1988.

[5] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabeu-Auban, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh, "The Design and Implementation of the Clouds Distributed Operating System ," *Usenix Computing Systems* 3:1 (Winter 1990), pp. 11-46.

[6] Prasun Dewan, "An Inheritance Model for Specifying Flexible Displays of Data Structures," Technical Report SERC-TR-54-P, Software Engineering Research Center, Purdue University, November 1989.

[7] Prasun Dewan and Eric Vasilik, "Supporting Objects in a Conventional Operating System," *Proceedings of the San Diego Winter '89 Usenix Conference*, February 1989, pp. 273-286.

[8] Prasun Dewan and Marvin Solomon, "An Approach to Support Automatic Generation of User Interfaces," *ACM Transactions on Programming Languages and Systems* 12:3 (to appear in October 1990). Preliminary version presented at the *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices* 22:1 pp. 150-159 (January 1987).

[9] Prasun Dewan and Eric Vasilik, "An Approach to Integrating User Interface Management Systems with Programming Languages," *Proceedings of IFIP TC2/WG 2.7 Working Conference on Engineering for Human-Computer Interaction, Napa Valley, August 1989*, North-Holland, 1990, pp. 493-514 .

[10] Prasun Dewan and Rajiv Choudhary, "Experience with the Suite Distributed Object Model," *Proceedings of IEEE Workshop on Experimental Distributed Systems*, to appear in October 1990.

[11] Scott E. Hudson and Roger King, "Implementing a User Interface as a System of Attributes," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices* 22:1 (January 1987), pp. 143-149.

[12] Dan R. Olsen Jr., Elizabeth P. Dempsey, and Roy Rogge, "Input Output Linkage in a User Interface Management System," *Computer Graphics: SIGGRAPH'85 Conference Proceedings* 19:3 (July 1985), pp. 225-234.

[13] Jock Mackinlay, "Automating the Design of Graphical Presentations of Relational Information," *ACM Transactions on Graphics* 5:2 (April 1986), pp. 110-141.

[14] Brad A. Myers, "Incense: A System for Displaying Data Structures," *Computer Graphics* 17:3 (July 1983), pp. 115-125.

[15] Brad A. Myers, "Creating User Interfaces Using Programming by Example, Visual Programming, and Constraints," *ACM Transactions on Programming Languages and Systems* 12:2 (April 1990), pp. 143-177.

[16] G. Pfaff, *User Interface Management Systems*, Springer Verlag, Englewood Cliffs, NJ, 1985.

[17] Ben Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *IEEE Computer* 16:8 (Aug 1983), pp. 57-69.

[18] Michal Young, Richard N. Taylor, and Dennis B. Troup, "Software Environment Architectures and User Interface Facilities," *IEEE Transactions on Software Engineering* 14:6 (June 1988), pp. 697-708.