A user interface manager : SepView

Jean-Claude Dulac

ABSTRACT

I wrote a software package called SepView that allows control of existing processing programs with graphical user interfaces. The design of these user interfaces and their management requires no programming: an ascii descriptive file specifies for each program, the interface. Changes to the interface output style and behavior are accomplished merely by editing this file. SepView is modular: new user-specified objects can be described in this file in addition to standard interactive objects, such as menus, buttons. SepView is also an object-oriented library which presents an object-oriented model to the programmer to simplify the creation of new objects and new interactive programs. Beyond its immediate application to the SEP computing library "seplib", SepView is an interesting example of programming and structuring user interfaces for seismic processing. SepView is already used to write an editor which provides a graphical programming environment for seismic data processing.

INTRODUCTION

The SEP library is a tool-based seismic data processing system (Claerbout, 1986). A tool is a particular canonical computer program of this system. A model "seplib" process reads an input history file, writes an updated output history file by appending processing information and by overriding default parameters, and then processes the data cube. The consistency of such data description ('n1','n2','n3',...: cube parameters) and processing programs allows the "seplib" user to string several processes together end-to-end with a UNIX pipe. A task is such a chain of processing programs. The command language provides efficient access to the data-cubes structure of the system. In general, an experienced user can use the tools efficiently. A problem arises, however, as soon as a user has to deal with a task for whose he lacks details about appropriate tools. This situation does applies both to the novice and the expert. In a complex task environment, only a few tools are used frequently and

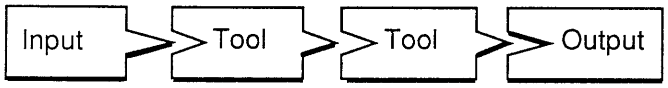


FIG. 1. **Pipe and redirection of I/O in UNIX:** graphical representation

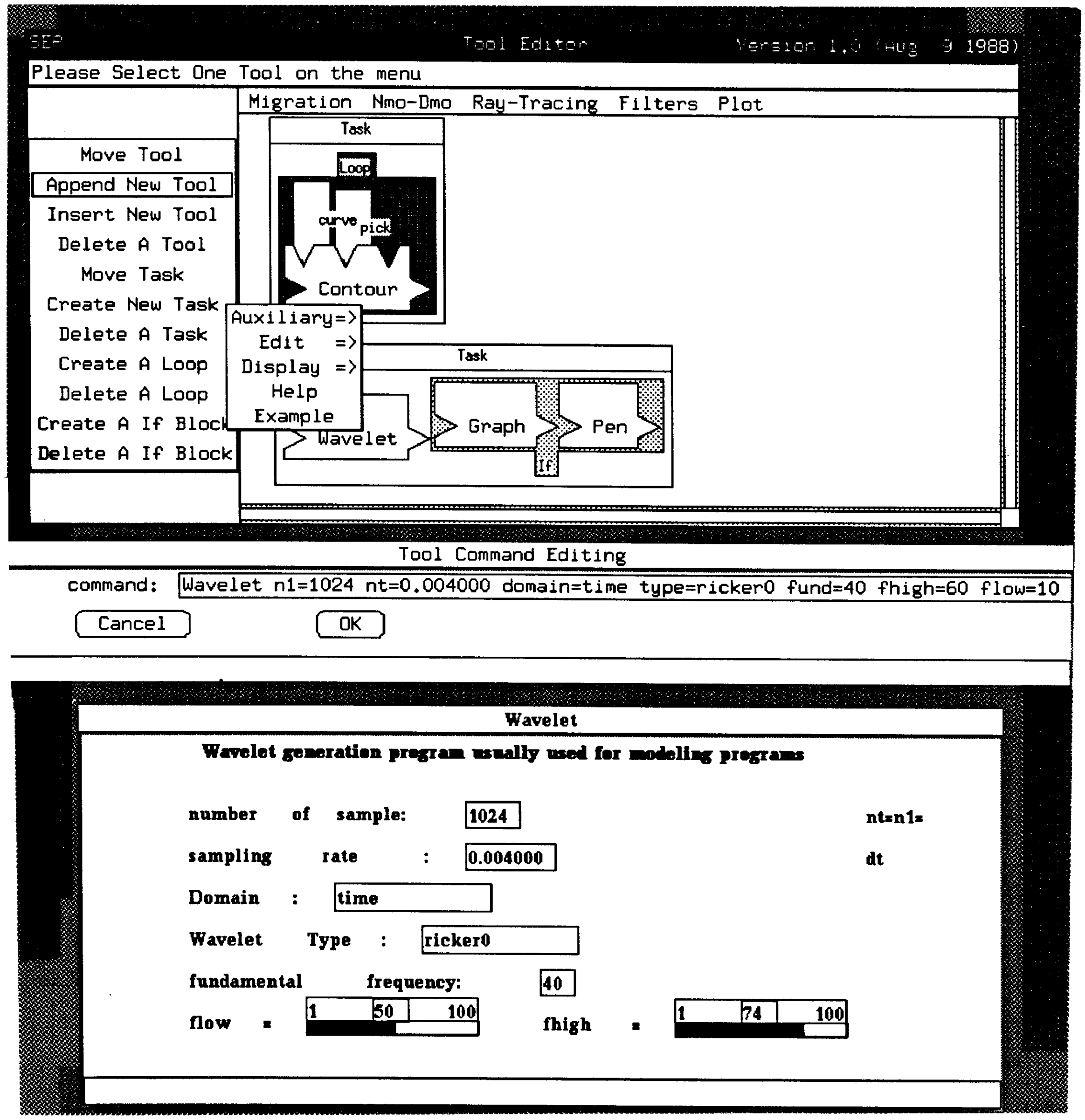
easely. Possible improvements of the dialogue interface are to make "seplib" more communicative and to enlarge the concept of tool history, as well as tool graphical control to task history and task graphical control. Believing that a graphical interface would improve the dialogue system, I began coding an advanced window system that consists of a bit-mapped display with mouse device, keyboard, menus, graphic and full screen capabilities. This window system is called SepView. In this window system, important processing features of "seplib", such as the pipe and redirection of I/O in UNIX, are visualized to achieve an useful interface. (Figures 1 and 2)

I will first present a review of user interface models, and then, introduce the main features of SepView before presenting an example and explaining some concepts I used in SepView.

BACKGROUND IN USER INTERFACES

According to Hoffman & Valder (1986), the definition of a user interface can be made in four steps (Figure 3):

1. An input/output model will describe how the user input and system output is presented on a screen.
2. A dialogue model then will describe the rules of the dialogue between the system and the user. Two aspects of the dialogue interface can be identified:
   (a) the way in which a user is guided by the system to enter commands, for example, with menu selection or system prompts.
   (b) the rules of the dialogue interface defining the amount of control the user has over the interface, i.e., how to start and stop a task or how to get on-line help.
3. A tool interface model next describes the rules which define access to underlying system tools. Possible improvement of the tool interface can be achieved by providing:
   (a) Knowledge about the task domain: strategies, methods, experience available, warning and suggestions for certain situations.
   (b) Knowledge about the objects which may be applied in a solution: information about the objects, the tools, and the task,



command: Wavelet n1=1024 nt=0.004000 domain=time type=ricker0 fund=40 fhigh=60 flow=10

Wavelet generation program usually used for modeling programs

number of sample:	1024	nt=n1=
sampling rate :	0.004000	dt
Domain :	time	
Wavelet Type :	ricker0	
fundamental frequency:	40	
flow =	1 50 100	
fhigh =	1 74 100	

FIG. 2. **Tool Editor** : SepView provides a tool editor to build a particular task graphically. Each program is associated with a particular user interface, and can be manipulated by the tool editor. This figure presents an aspect of a particular session with the tool editor. Two tasks have been built (upper figure), and a particular tool (Wavelet) from the lower task is being edited (lower figures) to define its parameters.

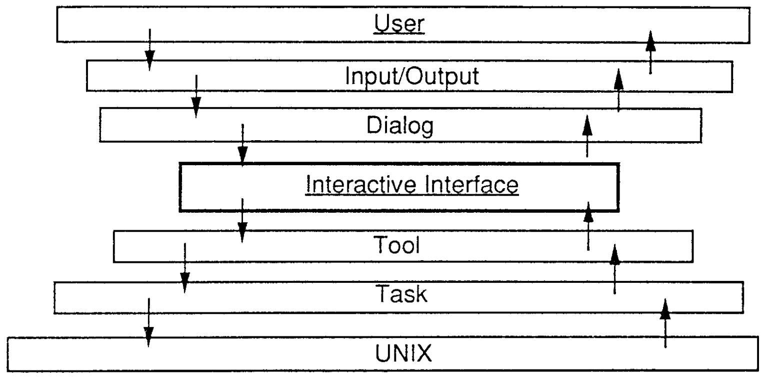


FIG. 3. **User Interface Model** according to Hoffman & Walder (1986)

(c) Knowleged about the component of the task: tools, objects, sub-task.

4. Last, an organizational interface model will comprise an attempt to describe relations between different tasks executed by different users.

According to Hendler (1987), interface design is more complicated than just putting up the windows on the screen. One technology proposed for helping designers cope with ever-increasing demands on their systems is User Interface Management Systems (UIMS). This method advocates creating tools which allow the designer to *define the interface in a declarative way, usually via some special languages features.* This form of interaction allows the designer of the interface to experiment with different forms more easily.

Many systems have already borrowed these ideas. One of these, and perhaps the best-known, is HyperCard^{TM}(Goodman, 1987), used to design applications on the Macintosh^{TM}. In HyperCard, the card or window is drawn using a special editor. The card is then inserted into a stack which represents the way the user goes through the application. The card and its elements are saved in a description file. Operations described in a script can be associated with interactive objects.

SEPVIEW

SepView is a user interface manager defined on top of the SEP library. SepView provides the components for the design and the control of a tool's interface (Figure 4). Each tool is represented by a set of interconnected windows where the user enters data. Theses control panels are described in a file. The interface is completely separated from the program, so that no recoding of the tool needs to be done to create a specific interface for this program. *Changes to the interface are quick and easy and accomplished merely by editing the data file. No recompilation is required.* SepView also provides the means to write and design any other kind

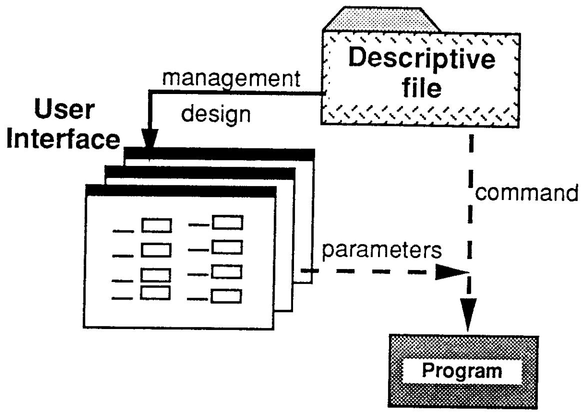


FIG. 4. **User Interface Tool model:** A tool program is associated with a graphical user interface described into a file. SepView reads this data file of interactive objects specifications and produces the user interface. The tool program does not manage the interface. After the interactive parameters acquisition, SepView executes the tool giving the defined parameters as command arguments.

of graphical user interface (Figure 5). SepView combines an application code and a user interface description to produce a complete application. The separation of interactive behavior and output aspect, as well as the separation of the interactive behavior and the abstract behavior (suiting a particular application), is enhanced by SepView and simplifies the code *development* and the code *maintenance.*

Main features of SepView

All *interactive objects* or *interactors*, which are the forms coming up on the screen, are designed in a descriptive file using a high-level language. Descriptive attributes for each object, can be specified, such as, text style or panel architecture. The "look" of the interface can be modified by substituing some attributes or some interactors into the description file. The "behavior" of the interactive objects is also described into a descriptive file. I use two concepts introduced by HyperCard to describe the dynamic part of the interactive object: *Navigation* and *Script.*

- In HyperCard, the navigation description is comparable to a `goto`. In SepView, navigation is described as a production rule (`if...then`). For example, the navigation rule allows the user to tell a particular object what is going to happen when the right mouse button is clicked on it, or when it is selected. The navigation rules allow linking the different scenes in a dialogue chain. This approach is more flexible than a programming one, and simplifies checking of the program flow and the specifications.

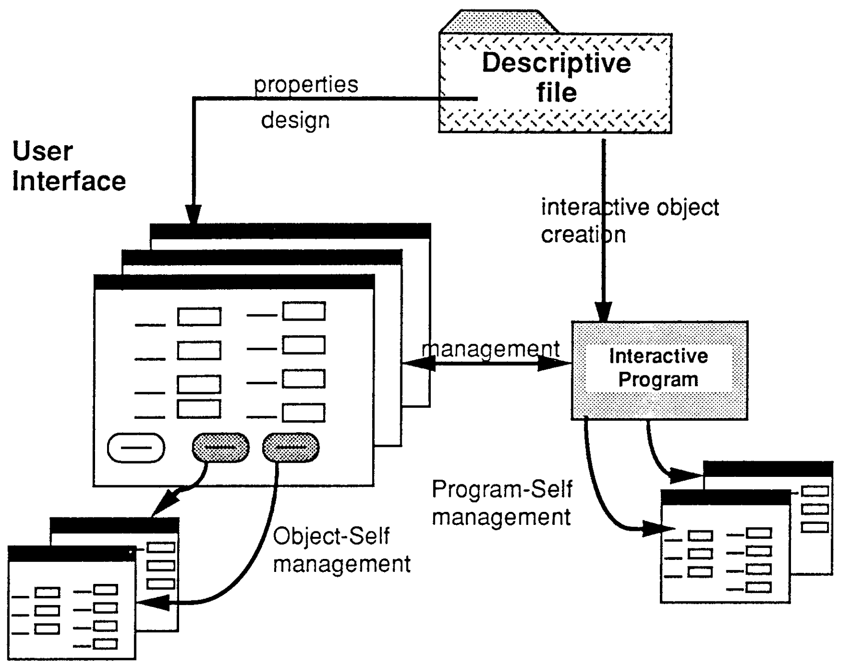


FIG. 5. **User Interface Interactive model: An interactive program can use SepView to describe its own objects and design parameters. Management of the interface is shared by SepView and the interactive program.**

- Run-time operations are described into a procedure called a script. For example, a script allows to initialize some interactors dynamically, to change at run-time the design of the screen, to compute data during an acquisition, to check input, and to communicate with the user program. The scripts are associated with all interactors. To write these script procedures, I have designed a language called C* (because it resembles C). I have added some new features to enable an easy interaction with the interface objects. (In HyperCard the script is written in a special language).

Implementation

SepView itself is written in C++ (Stroustrup, 1986), an object-oriented language. The object-oriented approach was necessary in the context of interactive objects manipulation. The interpretation of the descriptive file is made with a parser generated by "yacc" (Yet Another Compiler-Compiler)(UNIX) and an lexical analyzer generated by "lex" (A Lexical Analyzer Generator)(UNIX). For the implementation (Figure 6), I have used a C++ library called InterViews (Linton & Calder, 1987). InterViews is written on top of the X window library (Scheifler & Gettys, 1986) and provides the basic user interface package (Dulac, Nichols & Van Trier, 1988).

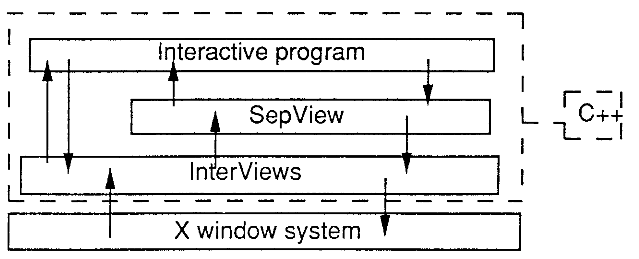


FIG. 6. **SepView Implementation:** SepView is written using InterViews which runs on top of the X window system. An interactive program can use both SepView and InterViews.

SCRUTINY OF SEPVIEW

An example of the most important feature of SepView - declaration of the interactive objects "look" and "feel" - is presented first. The next sections explain the interactive objects, script, navigation, and tool specifications used in the example descriptive file. I then describe the task concept - how its graphical representation presented in Figure 2 must be read. Finally, I explain how SepView accepts new interactive objects specifications.

Example

Figure 8 shows a complex control panel, which is defined in the ascii file presented in Figure 7. This description contains a tool declaration, a control script, and many interactor specifications.

Interactive objects

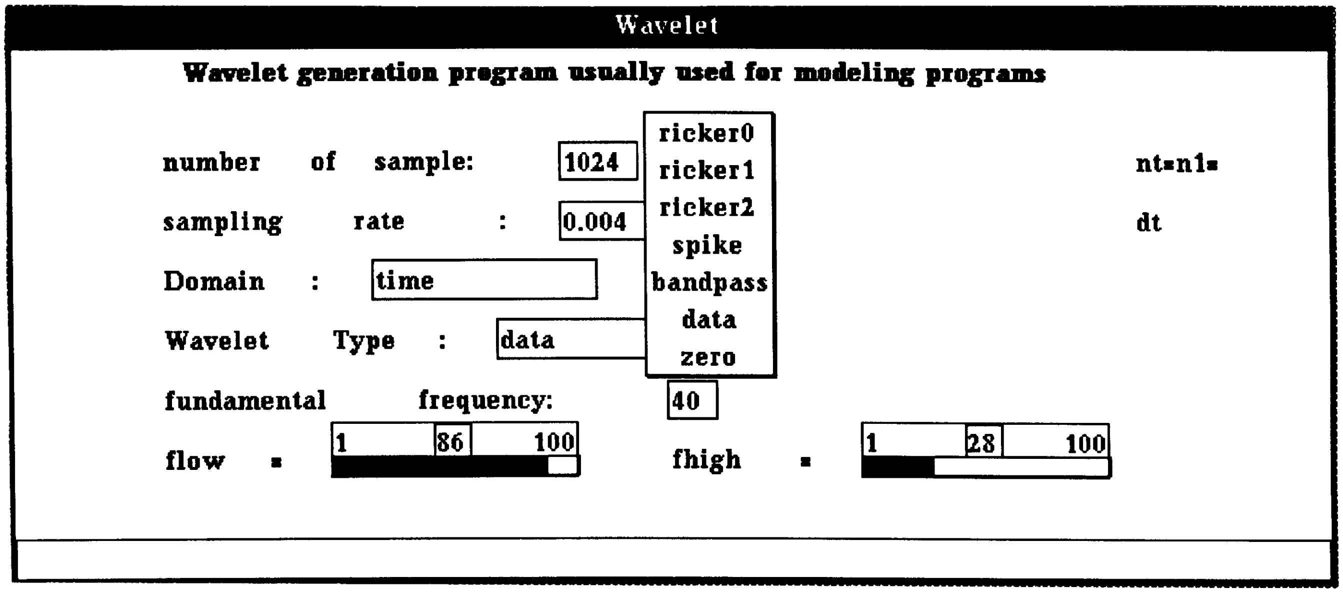
The interactive objects are divided into two parts: the simple interactors, and the composite interactors or scenes. The different interactors are derived from InterViews basic interactors. (Figure 9 and Appendix A)

The basic *interactor* class defines a general interactor object from which all other objects are derived. All interactors maintain an output state and an input value. All interactors implement a set of operations defined in the generic interactor class. These include operations for

- reading its attached description;
- setting its output state;
- setting, checking and retrieving its values;
- and evaluating itself. An evaluation of a scene is its display on the screen and its interactive manipulation.

```
% extern card Wavelet ;
% static menufield Domain, WaveletType ;
% static field samples , samprate, fund ;
% static slider flow, fhigh ;

Wavelet :
          loc   = 400,400 ;  font  = "timrom12b" ; title = "","Wavelet","" ;
          tool( name = "/usr/local/Wavelet" ;
                type = frontend ;
                synopsys = "Wavelet Generation for modeling program" ;
                parameters = "n1"     = $samples, "nt"    = $samprate,
                             "domain" = $Domain , "type"  = $WaveletType ,
                             "fund"   = $fund   , "fhigh" = $fhigh,
                             "flow"   = $flow ;
              ) ;
          endScript = { if( !$n1 ) {
                            error("n1 has to be defined") ; return(0) ;
                        } else if( $n1 < 0 ) {
                            error("n1 has to be positive") ; return(0) ;
                        }
                        return(1) ;
                      } ;
%%          /*------ ARCHITECTURE OF THE CARD -----------*/

      { \bf Wavelet generation program usually used for modeling programs }

      number of sample: [samples]            nt=n1=

      sampling rate   : [samprate]           dt

      Domain : [Domain]

      Wavelet Type : [WaveletType]

      fundamental frequency: [fund]

      flow = [flow]           fhigh = [fhigh]
%%      /*---------- OBJECTS DESCRIPTION ------------*/

samples  : = 1024 ;  // default value
           help = "Please give the number of samples" ;

samprate : = 0.004 ; help = "Please give the sample rate" ;

fund : = 40 ; help = "fundamental frequency of ricker wavelet" ;

flow : = 10 ; fvalue = 1 ; lvalue = 100 ; svalue = 1. ;
       help = "low cutoff frequency or butterworth filter" ;

fhigh : = 60 ; fvalue = 1 ; lvalue = 100 ; svalue = 1. ;
       help = "high cutoff frequency or butterworth filter" ;

Domain      : = "time" , "ctime", "frequency", "spectrum" ;
WaveletType : = "ricker0", "ricker1", "ricker2", "spike","bandpass", "data","zero";
```

FIG. 7. **Wavelet Description File:** Here is an example of a card description file. Interactors as field, slider, menufield are defined and mapped into this card according to the architecture which is the "vi" representation of the card. The interactors are surrounded by '[' ']' if they are fields, or surrounded by '<' '>' otherwise. All the interactors used in a file must be declared. An object can be declared as `static`, `extern` or `automatic`. The associated tool is also described.



Wavelet

Wavelet generation program usually used for modeling programs

number of sample: 1024 nt=n1=

sampling rate : 0.004 dt

Domain : time

Wavelet Type : data

fundamental frequency: 40

flow = 1 86 100

fhigh = 1 28 100

FIG. 8. **Wavelet Card.** This window is the result of the interpretation of the description file.

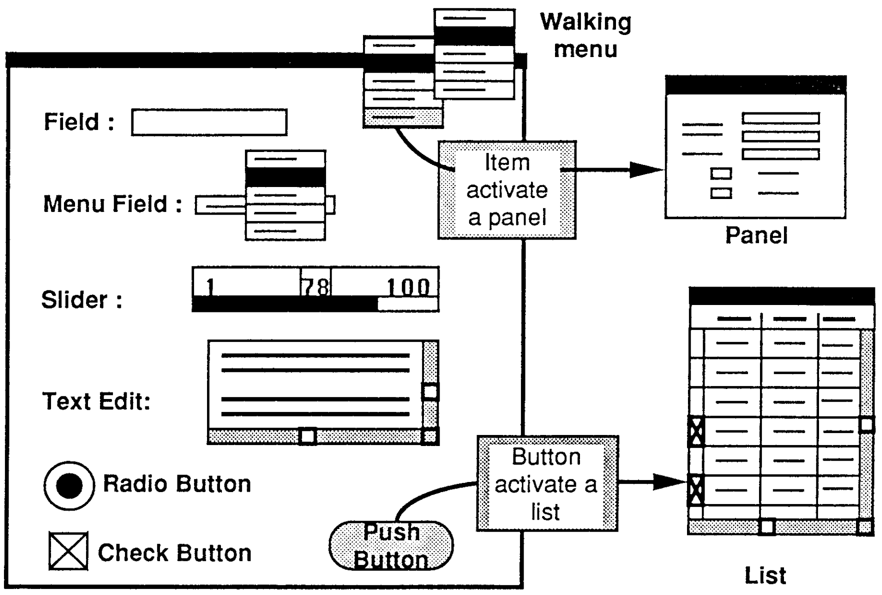


FIG. 9. **Interactors and graphical representation.** The basic interactors provided by SepView appear here: button, field, panel, list and menu.

All objects have a *text style* property which corresponds to the font in which they will be displayed. All objects have an associated *help* string. The (object-oriented) help for an interactor is a short sentence describing the object and its purpose (e.g. "apropos" command). The scene has a more sophisticated help object. This documentation gives a general information about the tool (i.e. like that provided by the UNIX "man" command). All interactors have an initial value that can be constant or run time evaluated. All have the capacity to be *visible* or *unvisible* at a particular instant. All have a *location* which is implicit for the interactors (deduced from the panel description), or explicit for the scene objects. All objects have the capacity of copying their *properties* from another element of the same type already described.

Script

The script, communication procedure between a program and its interface, allows to perform run time operations. Each interactor has three scripts: one to initialize, another to handle certain operations during the manipulation of the interactor (such as when the user is moving a slider), and finally one to perform some operations at the end of the manipulation.

- A script is surrounded by '{' and '}'. A script is written using a C-like syntax, called C*. All classical instructions (`if, for, while, switch, return,...`), classical type (`int, float, char *, ...`), classical operators (`+ /, ==, >=, ...`) and expressions are available.
- To allow the script to manipulate the interactive objects value, two features are added to C*:
  - The interactor variable value may be changed by using the **set** keyword: The following instruction set the interactor variable to a C* variable: **set** *Interactor Variable* = C*Variable.
  - The following metasequence is provided for introducing interactor variable values into the C* script: $*Interactor Variable* is replaced by the value of the variable and is used as a C* constant.

  Example :

```
% Field A ;                      // declare field A.
{ int i ;                        // declare i as an integer.
      if( $A > 0 ) { ...}        // evaluate the field A and test it's value.
      set A = i  ;               // set the field A value to i.
}                                // end of the script.
```

- To dialogue with the interactive program, in any C* block, it is possible to call **external** functions fully written in C and compiled:

- It can be built-in-functions such as `sqrt`, `printf`, ...
- but also function such as `delete(AnInteractor)`, `add(AnInteractor)`, or `error(''...'')` which are built-in-functions associated with the user interface manager.
- They can be *user precompiled functions.*

- To use or initialize data of the program, it is also possible to use *predefined user data types.*

The use of extern functions and user data types allows communication with the underlying program.

Navigation

The navigation part of an object allows the programmer to link scenes to one another dynamically. The navigation is a list of rules. A rule's syntax is the production rule syntax :

$$\boxed{\text{condition} \Rightarrow \text{consequence}},$$

meaning that if the condition is true, then execute the left part. The premise can be a script or a interactor. The consequence is a scene or a script. The premise and the consequence can be a list of premises or consequences such as P1 && P2 || P3 && !P4:

```
[ B && { return($F > 0); } ⇒ NextCard ;]
< RightMouse              ⇒ HelpMenu ;>.
```

A navigation surrounded by '[' and ']' is executed when the object is activated. The other one surrounded by '<' and '>' is activated when the object is manipulated. The right premise must be a premise describing an event. With left premises it is possible to associate an error message displayed if the left premis is false. A help message can be associated with a consequence, and will be displayed when the consequence is evaluated. Four special consequences, `Quit`, `Next`, `Prev`, and `Skip` are predefined. `Quit` implies an exit of the User Interface. `Next`, `Prev` and `Skip` allow one to manipulate the scene execution stack.

Navigation rules allow linking the different scenes in a dialogue chain. These rules may be used by the User Interface Manager to present the application in a command mode, in a stack mode, or in a tree mode, according to the user. These capacities allow one to go from a linear User Interface to a parallel one. This is one of the main ideas behind SepView. Navigation rules and script present easy ways to achieve the dialogue model.

Tool

A tool is the basic processing program that the user wants to execute. A tool description links a program tool with a set of windows. Then, a tool is described with its parameters, its auxiliary input and output files, its history input and output parameters. Figure 7 provides an example.

Tool parameters are described using two values: their names and the corresponding interactor. The default value is taken from the interface description and used to present a short version of the command (only the modified parameters) to the user, to allow him to edit quickly this command in a history substitution operation. This description allows the user interface manager to check the syntax of the command and eventually help the user type in a normal command session.

Associated with a tool, some context information can be provided, such as :

- A mail box to give the status of the tool, (information about whether the tool is running in the background or has already finished, and what time it has already consumed, etc.);
- An icon representation;
- A way to compute the time of the command (significant parameters);
- Examples of use;
- Links with other tools to propose to the user possible dialogue continuations and to find wrong use.
- Kind of tasks for which this tool is appropriate.

The tool description provides a standard scheme for describing the execution of a tool. The associated environment is one advantage of SepView.

Task

A task is a chain of at least one tool that is combined into a processing procedure. A task is itself a tool, then it has the same attributes than a tool (history files, parameters, icon, ...). The simplest task procedure combines tools using UNIX pipes. The procedure can also include loop statements or "if" statements. Graphic representation of a task is represented in Figure 10. This little program specifies flow of control and data. When a set of tools is surrounded by a loop-box, they will be executed until the loop is finished. Tools included into an "if" block will not be executed if the condition is false. To make the execution possible, intermediate history files may be generated automatically. The task execution is controlled by a description of the interdependency of the tools. This dependency can influence and can be influenced by a tool's parameter. When this happens, the user is asked for

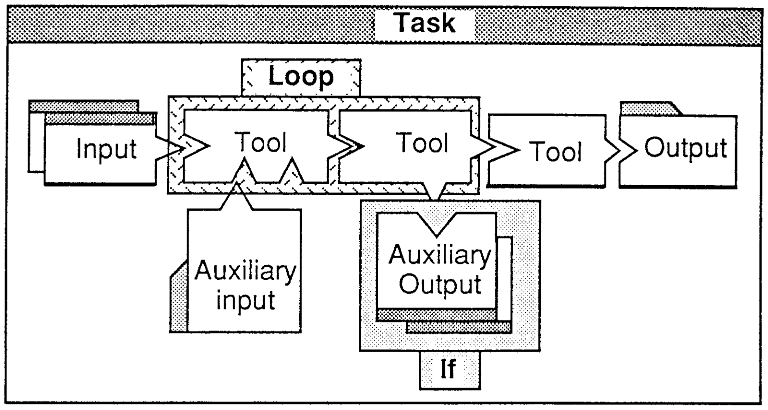


FIG. 10. **Task graphical representation:** The input itself is a task. Two tools are included into a loop. A tool auxiliary input is a file. Another tool has a task as auxiliary output. This task will be executed if the condition associated with the "if" block in which it is included is true. The output of the global task is a simple history file.

the parameters. As in UNIX's "make", tools are not executed if the standard or auxiliary input has not been modified.

A particular control panel can be associated with a task, allowing the user to define a particular interface with predefined parameters, computed parameters and interactive parameters. The user can then create his own tools using existing ones. The task is a self-contained, executable module which may be operated interactively from its front panel, or programmably by using it into an another task of higher level. The result is a construction environment in which the user can rapidly combine, interchange, and share modules with other users to build custom applications. Task is a central concept underlying SepView. SepView users may develop applications by building tasks which perform various functions and are combined to build applications.

Classes

SepView provides many pre-defined classes of interactors, but each application should be able to describe its own interactive objects. New interactor classes can be recognized by the interpreter by simply declaring their names and parameters into the description file. This class description gives **extendibility** to SepView.

Borrowing the concept of class in C++, a new user-defined type (called `class` like in C++) can be declared this way:

```
class new [Id] :  public old {
}
```

The **Id** is the class identifier used in the construction of the new class. The inheritance mechanism is also available. In the example above, class *new* is derived from a class *old.*

Definition of a user-specified type requires the specification of the data needed to represent an object of the type and a set of operations for manipulating such an object. The type can be a pre-defined type such as `int`, `float`, `string`, `boolean` or a `user-type`. A data element may have more than one type. Derived types as vector are also available. An example of the declaration of the class `iactor` and of the class `menu` derived from `scene` follows:

```
class iactor [101] {
     string help ;                                        // help string
     int, term loc[2] = userRelative, eventRelative ;     // location
     int dim[2] ;                                         // dimension
     string, iactor font ;                                // text style
     boolean visible ;                                    // visibility flag
     script  preScript, coScript, endScript ;             // operations

     Help() ;                                             // Help functionality
}

class item[121] ;                                         // pre-declaration

class menu : public scene {
     term type = popup , pulldown, menubar, choice ;      // type
     item ;                                               // item declaration
}
```

An operation specification can be set into a navigation rules in the consequence part, or into a script. During the navigation or script execution these operations can then be executed. An example of such function utilization follows:

```
% menu HelpLDataMenu ;

HelpLDataMenu : font="timrom12b" ;
               item( icon="Help"      ; [ => iactor.Help() ; ] ) ;
      // If this item is selected then the Help functionality of the current interactor is called.
               item( icon="List Data" ; [ => field.ListData() ; ] ) ;
      // If this item is selected then the ListData functionality of the current field is called.
```

At compilation type, type-checking is done. At run-time, a check is made if the function can be properly evaluated: if not, the interactor is invalidated.

The generic object from a class allows the user to predefine some attributes or some behaviors for all objects of this class. The generic is described as another object, but its associated values or behavior will be copied into new objects. Such generic declaration can be made into a root file. In the root file, standard objects can also be defined (Figure 11). This *standardization* is very helpful for the user. Only one representation of a concept needs to be memorized, and not the difference between “display”, “show”, and “list” which are different syntactic words for a same category of action.

```
              /*----- DEFAULT CONTROL MENU --------*/

% menu ExitConfirmMenu, HelpMenu, HelpLDataMenu ;

ExitConfirmMenu : font = "timrom12b" ;
                  item( icon = "Really Exit" ; [/* nothing */ => Exit ; ] );
                  item( icon = "No Exit" ;)  ;

HelpMenu : font = "timrom12b" ;
           item( icon = "Help" ; [ /* nothing */ => iactor.Help() ; ] );

               /*---- GENERIC CLASSES -----*/

generic button : type = push ;  < RightMouse => HelpMenu ; >
 // the default type button will be a push button.
 // A clik on the right mouse on the button will pop up the HelpMenu.

generic field  : type = string ; < RightMouse => HelpLDataMenu ; >
                 size = 20 ;
 // the default type field will be a string of lenght 20.
 // A clik on the right mouse on the field will pop up the HelpLDataMenu.
```

FIG. 11. **Root File:** An example of a root file that defines default control menus and defines two generics. Class declaration can be made into the root file.

STAGE OF COMPLETION

SepView converts **getpar** into control panel and pipes into graphical networks. Control panels are very hard to write in SunView and the Mac ToolBox. SepView emulates HyperCard in making it much easier to write control panels.

In its current state, SepView provides the elements to create tools user interface and interactive interface. Presently, the task editor allows one only to create a task and run it: all the history recovery and substitution has to be written. A resource editor to edit the interactive object description (like the Macintosh one) has to be written. A general improvement of the help system merging text and graphics has to be made. The current version of SepView does not implement completely the tool model. The user is not supported when he does not know the name of the right tool. I have to find how to collect and provide information about how to work on a specified task, offering the necessary tools and objects for the work. The current implementation does not provide an organizational environment, or information about the system state. The main changes will occur when the new version of C++ becomes available (Stroustrup, 1987). The current implementation will be improved and cleaned up.

Using C++ as the implementation language for SepView has had several benefits. Intrinsic to object-oriented language are facilities for data hiding and protection, extensibility and code sharing through inheritance, as well as flexibility through run time binding of operations to objects. Class inheritance and "virtual" functions simplify the structure of code and data, making the implementation easier to debug and understand. Much of the complexity is in the primitive classes, hidden from user interface designers. C++ is also very portable. A significant advantage of using C++ for SepView was that there was a good match between the language and the software. It was much easier to implement an object-oriented user interface package using an object-oriented language than it would have been with a procedural language. Classes define objects that model closely real objects and concepts the system is meant to manage. The programmer focuses on the objects that are manipulated, not on the flow of control. It is important to concentrate on the protocols for communication between objects. If these protocols are well-designed, the implementation is relatively straightforward.

REFERENCES

Claerbout, J.F., 1986, Canonical program library: SEP–**50**, 281-289.

Dulac J.C., Nichols D., and Van Trier J., 1988, An introduction to InterViews: SEP–**59**.

Goodman, D., 1987, The complete HyperCard handbook: Bantam Computer Books.

Hendler, J., and Lewis, C., 1987, Introduction: desigining interfaces for expert systems: *in* Hendler, J.A., Ed., Expert systems: the user interface: Ablex Publ. Co., 153-182.

Hoffman, C. and Valder W., 1986, Command language ergonomics: *in* Hopper, K. and Newman, I.A., Eds., Foundation for human-computer communication: Elsevier Science Publisher B.V., 218-232.

Linton, M., and Calder, P., 1987, The design and implementation of InterViews: *in* Proceedings of the USENIX C++ Workshop, Santa Fe, NM, 256-267.

Scheifler, R.W., and Gettys, J., 1986, The X window system: ACM **54**, No 2., 79-109.

Stroustrup, B., 1986, The C++ programming language: Addison-Wesley Publ. Co..

Stroustrup, B., 1987, The evolution of C++: 1985 to 1987: *in* Proceedings of the USENIX C++ Workshop, Santa Fe, NM, 1-21.

APPENDIX A : INTERACTORS

SepView provides differents type of interactors divided into scenes and simple interactors which are listed below:

Interactors: there are five types of basic interactors:

Buttons: Buttons can be of three types :

Push Button: One time selected.

Radio Button: Many times selected and unselected.

Check Box: Switch between selected and previous value.

Buttons could be configured in three different ways. They could be independent. They could share a same status, in a such way that one and only one of this list can be selected at a given time. Buttons can be build hierarchically, in such way that if the main button is not selected the other buttons are invalid.

Fields: A field allows the user to enter a value. The size will determine the maximum number of characters the user can input in the field. There are three types of field depending on the type of input value. These types are integer, real, and character. In an integer field, only digits, '-' '+' are accepted. In a real field, no letters are allowed except E or e. Help can be provided using a dynamic list of values which can fill the field. This list is set by the listdata attribute.

Display Fields: They display a value at a given location. The inside value can be set by the application but not directly by the user.

Menu Fields: Fields which can take their value only among a certain set of values. These values will be presented with a pop-up menu and the selected value will be put in the associate list field.

Multi Line Fields: Fields which allow multiline input. Scrollbar and panner will permit scanning through the text.

Sliders: Select a continuum value between two extremes using a cursor.

Scenes: Five basic types of scene exist:

Cards or Panels: A card is a scene, and has different interactors mapped according to the card's architecture. Its behavior is determined by its script or by the script of its components.

Menu: A pop-up menu displays many items and selects an item based on the position of the pointing device. Items are arranged vertically on a box. A '$\Rightarrow$' in the left of a item means that more options are available. These options are displayed when the pointing device points to the '$\Rightarrow$'. An item can be valid or not. It is always displayed in gray, and is not selectable if invalid.

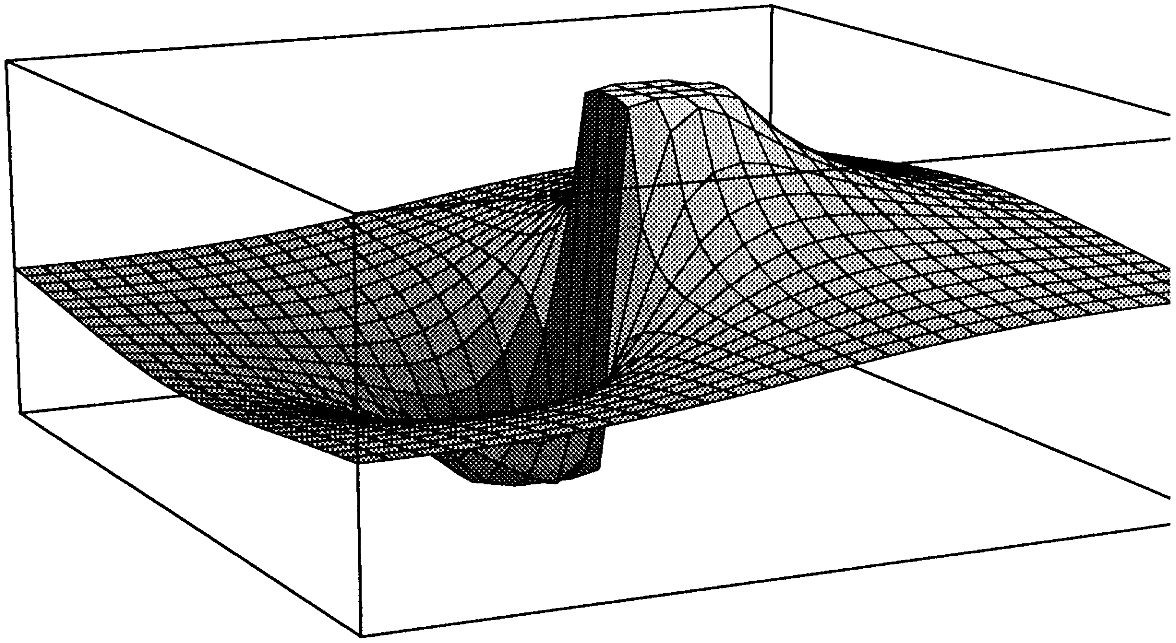
List: A list is like a menu, but if all the components of the list cannot be displayed on the screen, a scroll bar is used to provide access to all components. One element or several can be selected.

Table: a table is an array of fields in which each column represents a vector of data. Some editing operations such as delete, copy and move, can be performed on the rows of the table.

Graphic: A graphic scene is an interactor in which a graphic is displayed to be edited using user functions.

In[25]:=

```
Plot3D[
     Re[1/(x+I y)]
     ,{x,-1,1}
     ,{y,-1,1}
     ,PlotPoints->30
     ,PlotRange->{-3,3}
     ,ViewPoint->{-2,-4,1.15}
     ,BoxRatios->{2,2,1}
     ]
```



Out[25]=

```
-SurfaceGraphics-
```