ITS: A Tool for Rapidly Developing Interactive Applications

CHARLES WIECHA, WILLIAM BENNETT, STEPHEN BOIES, JOHN GOULD, and SHARON GREENE
IBM T.J. Watson Research Center

The ITS architecture separates applications into four layers. The action layer implements back-end application functions. The dialog layer defines the content of the user interface, independent of its style. Content specifies the objects included in each frame of the interface, the flow of control among frames, and what actions are associated with each object. The style rule layer defines the presentation and behavior of a family of interaction techniques. Finally, the style program layer implements primitive toolkit objects that are composed by the rule layer into complete interaction techniques. This paper describes the architecture in detail, compares it with previous User Interface Management Systems and toolkits, and describes how ITS is being used to implement the visitor information system for EXPO '92.

Categories and Subject Descriptors. D.2.2 [**Software Engineering**]: Tools and Techniques—*software libraries, user interfaces*; H.1.2 [**Models and Principles**]: User/Machine Systems—*human factors*; H.4 [**Information Systems Applications**]: General; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*device independence, ergonomics, interaction techniques, languages*; K.6.1 [**Management of Computing and Information Systems**]: Project and People Management—*systems analysis and design, systems development*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software development, software maintenance.*

General Terms: Design, Human Factors, Languages, Management, Standardization

Additional Key Words and Phrases: Management systems, user interface

1. INTRODUCTION

Application developers today face a problem of great complexity. Because of the diversity of users and of their equipment, applications must run in many different configurations. They must support displays of varying size, resolution, and color depth. Different types of input devices are required, from keyboards to touch screens. Applications must run in different countries and be able to reformat messages in varying lengths in each language. Messages should be available in large font sizes for vision-impaired users. Interface style should be consistent

Portions of this paper have previously appeared in the proceedings of the 1989 and 1990 ACM Symposia on User Interface Software and Technology.
Authors' address: IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598. email: wiecha@ibm.com

with other applications running on similar hardware. Style should at the same time conform to guidelines being developed by many organizations for presentation and interaction behaviors.

Historically, two types of layered architectures have been developed in attempts to provide the required flexibility in applications: User Interface Management Systems (UIMS) and toolkits. UIMS separate the application from its interface. Back-end computations are separated from dialog control and style. Style, however, is often treated in a single interface layer. Toolkits separate style from the application. Dialog control remains in the back end while the implementation of interaction techniques is hidden in a code library.

ITS is a four-layer architecture with separate tools for back-end computation, dialog control, definition of interface style, and implementation of graphic primitives used in the style. We begin by reviewing previous work done in UIMS and toolkits. Then we show how the layers of the ITS architecture combine the benefits of both UIMS and toolkits. After describing each layer of the architecture we show how ITS is being used to implement the visitor information system for EXPO '92 in Seville.

2. RELATED WORK

2.1 User Interface Management Systems

User Interface Management Systems, by analogy to Data Base Management Systems, divide applications into two layers. An application layer implements the underlying computations, while an interface layer implements the details of interface presentation and interaction. According to the traditional Seeheim [18] architecture, the interface layer defines the connection to the application, dialog flow of control, and presentation style. The application layer is structured as a set of callback routines executed in response to user actions or external interrupts.

The goals of separating the application from its interface are well documented [9, 10], and include reusing a given application in multiple interfaces, consistency among a family of applications that run using a common interface, and independent tools for application and interface developers. We believe previous UIMS have had limited success in meeting these goals because of two problems: They have not sufficiently decomposed the interface layer, and they have provided inadequate tools for interface designers.

The Seeheim model calls for separate dialog control and presentation components in the interface layer. Much of the previous work on UIMS has focused on the dialog component. Less attention has been paid to presentation. Previous systems sometimes represented presentation and dialog in a single layer. Serpent [20], for example, allows for control over style by composing primitive graphic objects into compound interaction techniques. However, this composition is specified together with dialog control flow.

In other systems, work has only recently begun to refine the presentation layer. One example is UIDE [6]. UIDE refines dialog into several layers for application data objects, commands, and control. Recent work focuses on new layers for intelligent control over presentation style [7]. The UofA* UIMS [22] separates dialog control from interface style. Interface style is generated automatically.

The rules that control style, however, are not exposed in a layer of their own. Rather, the interface designer tunes the interface once it has been automatically generated.

For those systems that do allow designer control over style, two approaches have been taken: manual editing with WYSIWYG or constraint-based editors, and automatic generation using executable style rules. An early example of the editing approach is the WYSIWYG form editor in Cousin [10]. Using such a tool, a designer may position command buttons or menu items and link them to application actions by direct manipulation.

Early editors supported only static screen designs. They specified *what* objects should appear and unconditionally *where* they should be located. These editors are inadequate for developing applications that must meet criteria of portability across hardware, end-user language, and type of user. An application that runs even in just two languages such as English and Spanish has different layouts driven by the translation of its messages. If it must run on displays of different resolutions or under a window system with resizable windows, its layout must change even more dramatically.

Work on interface workbenches has begun to respond to these challenges. More recent systems [3, 11] are based on layout constraints. Constraints describe *how* to reflow objects when window size or other variables change. However, constraints still do not capture *when* and *why* objects should appear, that is, the specifications in a style guide or the designer's knowledge that led to the given layout. Such knowledge is required if each screen is not to be designed individually or if the design is to be altered automatically under changing conditions.

Examples of systems that automatically generate interfaces include APEX [5], APT [15], Jade [25], and ITS [2, 26]. These systems all have the goal of encoding interface style in a set of executable rules. One dimension that helps distinguish among them is the intended role of their rules relative to an interface designer using the system. We can distinguish between design formulation and execution. Formulation involves making decisions such as what gridding to use in screen layout or what color values are appropriate for representing concepts such as errors, values, or prompts. Execution involves being able to apply these decisions automatically by executing style rules. APEX, for example, stresses the formulations of style. It consists of metarules for constructing style. In contrast, a graphic artist or human factors engineer formulates the design in ITS. Style rules are the language for representing and executing that design. The usability of the style language for designers is therefore of much greater concern in ITS than in previous work.

2.2 Toolkits

Toolkits differ from UIMS in the architectural separation that they make between application and interface. Toolkits are code libraries of common interface components such as standard windows, dialog boxes, menus, and dials. Applications developed using toolkits do not separate code for user interaction from the rest of the application. As in conventional applications, toolkit applications control internally the selection of interaction techniques and the sequencing of the user's dialog with them. To use a menu rather than a text entry field, for example, the programmer must change application code. To allow the user to interact nonmodally with a set of menus rather than step sequentially through them also requires changes to application code. In most UIMS both the selection and control of interaction techniques are external to the application.

Tookits do remove the composition and implementation of interaction techniques from applications. Thus a window may be composed of a hierarchy of objects including title bar, resize icon, and scrollbar. To the application programmer the window is created and manipulated as a single object. The implementation of each of these objects is also hidden in code reused from the toolkit. Thus the useful abstraction that toolkits offer application programmers is in the implementation of interaction techniques, not in their selection or design.

Historically, toolkits evolved out of the window system community in a variety of environments. The best known examples are MacApp [19] and the several X toolkits, Xtk [27], Atk [17], and Interviews [14]. Some of these toolkits gained quick commercial success because of their reduction of the complexity of programming interfaces.

The second factor driving their success is the growing interest in interface consistency and standards. By hiding the composition of interaction techniques from application programmers, toolkits control the implementation of interface style. An application that requests a window will by default get the toolkit standard definition of that window automatically.

Toolkits have two major problems in supporting the growing interest in consistency and standards. First, new interaction techniques can usually be composed only by programming. To create an analog clock object, for example, one must write a new toolkit object that instantiates, sizes, and positions a set of more primitive objects for clock hands and numbers. Nonprogramming graphic artists and human factors engineers therefore still lack appropriate tools to create styles themselves.

Toolkits are also poor tools for supporting consistency since they do not capture the rules governing when to use each interaction technique. These rules, an essential part of style, are contained only in hardcopy style guides. Such guides have now been published for a variety of styles, including the MacIntosh [1], Open Look [23], OSF/Motif [16], and Common User Access [12].

Hardcopy style guidelines are specifications for the presentation and interaction behavior of user interfaces. They may include general design philosophy, such as "adopt an object-action style" or "keep the user in control of the application at all times." Guidelines may also include more specific rules for implementing the interface, such as "the first item in a pull-down bar must appear below and one character to the right of the action bar item that leads to it." Based on informal observations of experience at IBM and elsewhere, guidelines are difficult to interpret by programmers and omit important details necessary for the complete definition of a style.

3. THE ITS ARCHITECTURE

3.1 Overview

The ITS architecture has four layers as shown in Figure 1. Actions read and write data values without concern for dialog control (when or in what sequence they will be called). They do not change depending on whether, or how many,

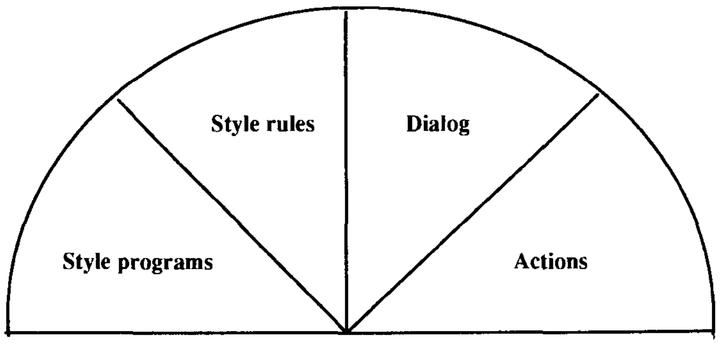


Fig. 1. Architecture of ITS applications.

views are attached to the data values they compute. The style of those views may change independently as well, for example, from tables to graphs. Actions are written in C.

Dialog is specified in the next layer of the system. Dialog includes the content of logical *frames* and the control flow among them. Separating dialog from actions allows actions to be reused in different applications. An airline reservation system intended for public access, for example, might not include some functions that are available to reservations agents. The dialog written for the reservations agent would have *frames* for those additional functions and the control flow necessary to access them. Two versions of the application can be created by writing different dialogs without having to change the actions themselves.

So far the application is not committed to a style, screen layout, language, or window system. The rule layer composes style programs into interaction techniques for each object in the dialog. An example of one type of style decision is whether to represent a *choice* the user must make as a radio button menu or action bar. These decisions are made automatically by a set of executable style rules. Style rules are written, and are open to modification, by interface designers. Style can be changed at any time by the appropriate selection of the rule base to be used in a given application.

Style rules are executed at compile time. Run-time changes in the dialog are managed by the style program layer. Example style programs include routines for text formatting, tree layout, row and column table layout, circular menus, raster images, scalable border decorations, and selectable hot spots on maps or images. Style programs can be added to the toolkit as required.

Style programs commit the application to screen layout, including the size and line wrapping of individual items. Layout is initially decided as each frame is drawn at run-time. Layout can later change when, for example, the user resizes a window.

3.2 Combining UIMS and Toolkit Layering

The split between actions and dialog in Figure 1 corresponds to the familiar separation between the application and its interface in previous UIMS. The remaining three layers further divide the interface into style-independent dialog

```
:frame id=main
  :choice purpose=overview, message="Select an item to"
    :ci message="View flights for today", activate=check_today
    :ci message="View future flights", activate=check_by_date
    :ci message="Make a reservation", activate=reserve
  :echoice
:eframe
```

Fig. 2. Dialog for opening frame of example airline reservations kiosk.

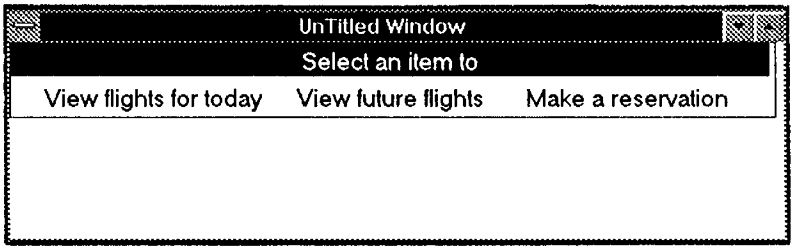


Fig. 3. Opening frame of the airlines reservation example shown in one possible style.

control, definition of style, and implementation of the toolkit objects used in creating styles. To understand the benefits of this layering consider a simple example, which might appear in an airlines reservation kiosk. Figure 2 shows the opening *frame* of dialog for the kiosk. The *frame* appears in one style as shown in Figure 3. *Choice* and *choice item* (*ci*) statements are keywords in the dialog language. *Messages* provide descriptive text that can be used variously as titles, prompts, or defaults under control of style rules. The dialog branches to the *frame* named by the *activate* attribute when the user *executes* an item.

To appear as shown in Figure 3, one or more style rules must fire and transform the *choice* into the tree shown in Figure 4. The rules introduce additional nodes for components of the menu not coded in the dialog. The style rules add a *title* and an object that arranges children horizontally (in this case the choice items). Style rules also decorate each node with attributes controlling presentation (such as color, font, margins, horizontal and vertical layout) and interaction behavior.

Splitting the interface into separate layers for style-independent dialog, rule base, and toolkit has three benefits. First, the dialog remains independent of style. Dialog can be mapped into any style simply by firing the appropriate rule. Second, interface designers now control style rather than application programmers. Application designers build dialogs only from style-independent objects such as *choice*. The rule layer represents the selection criteria for all interaction techniques. As we have seen, this knowledge is not captured in executable form in toolkit applications, only in hardcopy style guides.

The third consequence of the ITS architecture is that new interaction techniques can be composed by style rules from existing toolkit objects. In conventional toolkits, new interaction techniques are also composed from existing objects, but usually by programming. The example menu here was constructed

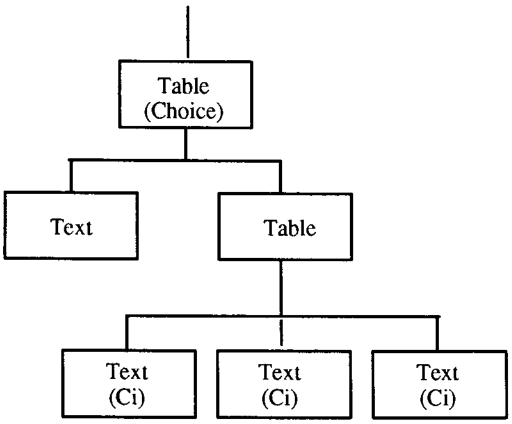


Fig. 4. Tree of units for an action bar menu. Types of nodes originally coded in the dialog are given in parentheses.

from only three toolkit objects: text, raster icons, and tabular layout. The style decisions on how to compose them into the final menu are made entirely by rules.

Since the menu is created by rules, a "menu" object is never added to the toolkit. The menu exists only as the coordinated behavior of the primitive objects. Since the toolkit need contain only primitive objects, programmers themselves have less need to implement new objects as style changes. As a result, interface designers enjoy control over style independent both of application and of toolkit programmers.

New objects are added to the toolkit whenever an interaction technique cannot be built simply by composing existing objects. Of course, these new objects are then available for reuse in composing other interaction techniques that may wish to build upon them.

3.3 Usability of Design Tools by Nonprogrammers

If application and style designers are to be truely independent from programmers, their dialog and rule languages must be highly usable. A financial analyst should be able to create a tax application, or an airline route planner a crew-scheduling application, with minimal application or style programming support. We want to give application and style experts executable tools to engage them as first class participants in application design. Currently their role is often restricted to providing input to systems analysts or programmers implementing widgets in a toolkit. Without appropriate software tools, admonitions toward user-driven design will remain only that.

Each layer in the ITS architecture corresponds to one of four roles in application development: application programmer, application expert, style expert, and style programmer. Of course, these roles may be played by one or several people as appropriate to the particular application.

An application expert is familiar with some problem domain, such as the airline industry, and wants to implement an application in that domain. The application expert typically is neither trained in programming, nor part of an information systems department. Traditionally the application expert interacts with a systems analyst to produce a design specification. Later, the specification is implemented by application programmers. Application designs produced by the analyst are not executable, but usually are represented as natural language or pseudocode algorithms. These incomplete specifications create opportunities for costly design errors. In ITS the application expert is the dialog author.

The style expert has had perhaps the poorest tool support in traditional application development. A style expert may be a graphic artist or human factors engineer. Style experts in the past have worked by drafting design guidelines or evaluating prototypes. They have had no separable work product, such as an executable style definition. Since they have had to influence designs indirectly through software developers, they have had difficulty having significant impacts. Rules give them direct control over style in ITS.

4. THE ACTION LAYER

A dialog depends on actions to implement the semantics of application functions. Actions are separated from dialog to create a library of reusable functions, to reduce the complexity of the dialog itself, and to allow changes in back-end computations independent of the interface. To communicate, the dialog and actions must share both data and control.

4.1 Data Sharing between Actions and Interface

Actions read values from data tables and return results to the same or other fields in tables. All communication between the application and interface occurs by passing values stored in data tables (see Figure 5). The collection of all data tables active in an application is called the data pool.

A particular instance of a data table is stored only once in the data pool. As in the Andrew Toolkit or Smalltalk Model-View-Controllers [13], any number of dialog objects may refer to a data table. Thus the same data may be shown in both tabular and graphical form if it is viewed simultaneously by two dialog objects. Whenever a value is changed by an action or by the user, all dialog objects representing it will be notified automatically. Each object then updates the display in whatever way is appropriate. The actions themselves have no knowledge of the number or type of dialog objects viewing the tables they manipulate in the data pool.

The update mechanism that sends style program messages about changed values is a hybrid of program-initiated and automatic notification [24]. Actions take the initiative in setting a bit in the data pool to indicate they have changed a field. This change flag is then propagated to style programs automatically by the dialog manager. Further, only those views of the changed field that are currently visible on the screen receive update notices. Others redraw only when they become visible.

4.2 Calling Actions

Control lies with the dialog manager until a user event triggers a call to an action. These events include *frame initialization* and *termination, selection, keyboard entry*, and *execution*. When called, the action is passed the event type and current dialog object. The events and parameters define the ITS architecture for actions. An action must be able to respond to one or more events. Otherwise actions are

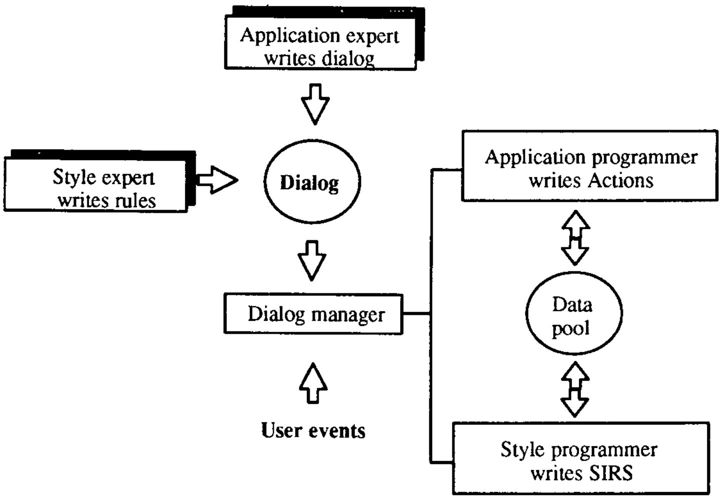

Application expert writes dialog
Style expert writes rules
Dialog
Dialog manager
User events
Application programmer writes Actions
Data pool
Style programmer writes SIRS

Fig. 5. The dialog manager calls actions in response to user events. Actions read and write values in the data pool. Change flags in the data pool trigger update notices to all style programs currently displaying the changed values.

conventional C library functions. In the OS/2 version of ITS they are typically stored in dynamically linked libraries and demand loaded.

This "external" model of control [10] can be interrupted asynchronously by an action that independently notices an error or other condition requiring user action. The action causes an interrupt by marking an arbitrary field in the data pool as changed. The resulting update notices may cause the dialog manager to branch to an error frame or to call other actions to correct the error.

5. THE DIALOG LAYER

The application expert in ITS is responsible for defining the *content* of the interface in the dialog layer. *Content* consists of two parts: tables that describe the data to be stored in the data pool and frames that define the application dialog itself.

5.1 Data Tables

Application actions receive input from the data pool and write output back to the data pool. ITS organizes these values using two data structures: *forms* and *lists*. A single set of *fields* is called a *form*; a set of *forms* is called a *list*. Both structures can be drawn, of course, in any number of styles not necessarily similar to physical forms and lists.

The application expert defines the structure of the data pool by writing table definitions. Tables are templates for the *fields* in each *list* and *form*. One or more instances of a table can be created in the pool under dialog control. Each instance is called a *generation* of the data object. Multiple *generations* are particularly useful when *frames* are called recursively and need private data spaces.

```
:form table=flight_info
  :fi field=destination, rangename=cities, size=20
  :fi field=departure_time, size=10
  :fi field=departure_date, size=20
  :fi field=airline, rangename=airlines, size=20
  :fi field=number_stops, size=5
:eform

:list listname=flights, numrecords=10
  :li field=destination, rangename=cities, size=20
  :li field=departure_time, size=10
  :li field=departure_date, size=20
  :li field=airline, rangename=airlines, size=20
  :li field=number_stops, size=5
:elist
```

Fig. 6. Data definition file. A form is a named collection of fields. A list is an array of forms with "numrecords" elements.

Tables for the airlines reservation example are shown in Figure 6. Each *field* has a name and an optional set of descriptive attributes. The *datatype* may be used by style rules in selecting appropriate interaction techniques for the *field.* The *size* attribute can be used in estimating the screen space required for the *field.* A *message* can be used as a title or prompt if no *message* is given in the dialog file.

Tables describe the data in a family of applications. Attributes given in the tables are inherited by a dialog when it refers to each *field.* Thus a family of dialogs may be built using the same table definitions and reuse attributes coded only once in the table.

5.2 Dialog

The dialog layer defines the content of a set of *frames* and the flow of control among them. Traditionally, application programs have fully controlled the sequencing of user dialog. Input and output statements were intermixed with those performing application computations. The application decided when and in what order interaction would take place. Nearly all UIMS remove control of dialog sequencing from the application and place it instead either in a central dialog manager [10] or in a distributed set of interaction objects that jointly regulate user interactions [21].

As we saw in Figure 2, control branches to the frame named by the *activate* attribute when the user executes a dialog statement. *Activated* frames are put on a run-time stack and popped from the stack as the user returns from them. Since the user can follow multiple dialog paths simultaneously, perhaps by interacting in different windows in parallel, the stack is really a tree of *frame* activations.

The *attachto* attribute controls where a *frame* is added to the execution stack. According to its intended lifetime in the dialog a *frame* may be attached near the root or near the leaves of the tree. For example, *frames* that are attached at the root of the stack exist until the user explicitly terminates them or leaves the application. *Frames* that are attached to lower branches of the stack terminate when any *frame* above them terminates. Since generations of *forms* and *lists* in the data pool are also linked to nodes in the stack, a *frame* may be bound to different data depending on where it appears in the stack.

Name	Description
Session	Indicates the start of a tag file or complete tag set.
Frame	Indicates the start of a group of conceptually related dialog blocks.
List	Indicates the start of a set of related items, e.g., a set of employees in an organization.
Li	List item. Specifies a field that should be included in a list built by an enclosing list tag.
Form	The start of a form, e.g., a related collection of fields.
Fi	Form item. Names a field to be included in a form built by an enclosing form tag.
Choice	Indicates the start of a set of alternatives to be chosen by the user.
Ci	Choice item. Specifies one of the choice items in the list enclosed by a Choice tag.
Info	Information block. A static panel of help information.
Ii	Information item. An item in a help panel.

Fig. 7. Objects used to specify dialog content.

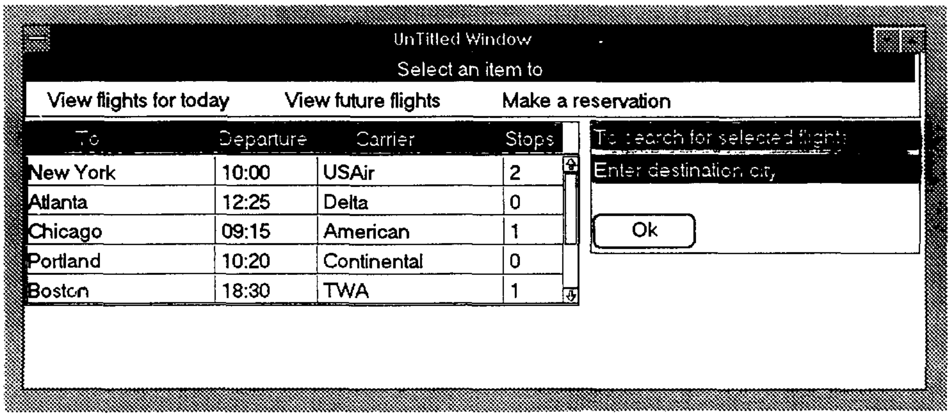
Name	Values	Definition
Action	⟨action name⟩	Name of the action to call on frame entry, item selection, or execution.
Activate	⟨frame id⟩	Transfers control to frame "id" when an item is selected.
Purpose	Elaboration Example Feedback Group Help Instructional Overview Useraction Warning	Specifies the intention of the application expert in creating the frame. Can be used by style to set colors, titles, or controls appropriately.
Structure	Continuum Scale Order Set Points Label Disjoint	Description of the underlying nature of the data contained in the field. Used to choose appropriate interaction techniques.
UserCan	Delete Filter Hide Modify Search Select Show Sort	Specifies the actions the user can perform on the specified field.

Fig. 8. Selected dialog attributes.

Next, the application expert defines the content of each *frame* in the dialog. To simplify the application expert's job, we chose a small fixed set of dialog objects. These objects are listed in Figure 7. Each statement may include a number of attributes, some of which are listed in Figure 8. Attributes are used to specify the *actions* executed, *values* set, and *messages* displayed by each dialog object.

```
:frame id=check_today, action=getlist,
      listname=flights, value=flights.dta
  :list listname=flights, number=5
    :li field=destination, message="To"
    :li field=departure_time, message="Departure"
    :li field=airline, message="Carrier"
    :li field=number_stops, message="Stops"
  :elist
  :frame message="To search for selected flights"
    form table=flight_info
      :fi field=destination, message="Enter destination city"
    :eform
    :choice purpose=useraction
      :ci message="Ok", action=select_flights,
        table=flight_info, listname=flights
    :echoice
  :eframe
:eframe
```

Fig. 9. Frame for viewing and selecting flights in example airline reservation kiosk.



To	Departure	Carrier	Stops
New York	10:00	USAir	2
Atlanta	12:25	Delta	0
Chicago	09:15	American	1
Portland	10:20	Continental	0
Boston	18:30	TWA	1

Fig. 10. Flight list and controls for selecting particular flights in the example airline reservation kiosk.

Figure 9 shows how these objects have been used to display a list of flights in the airline reservation example. The *frame* is shown in one possible style in Figure 10.

The *frame* contains a *list* showing four of the five *fields* defined for "flights" in the data definition file. Each *field* is coded on a list item (*li*) statement. The list items define the *fields* displayed in each *record* of the list. Five *records* will be presented to the user at a time. *Lists* and *forms* defined in the dialog file are views onto data values themselves, which are stored in the data pool. All views of "flights" will be synchronized by update messages from the dialog manager whenever any of them modify the data pool.

The *subframe* in Figure 9 groups a *form* and *choice* into a control for selecting particular flights. "Select_flights" will be called when the user *executes* "Ok." The action uses all nonnull fields of the table "flight_info" as search keys into the “flights” list. It replaces “flights” with the set of *records* satisfying the search. Dialog manager update messages then cause all views of the list to be redrawn.

6. THE RULE LAYER

What is style? One definition of style is a coordinated set of decisions on the appearance and behavior of the interaction techniques used in a family of applications. Three dimensions of this definition are important: (1) it refers to both the output and the input characteristics of interaction techniques, (2) it considers both style in the small (individual techniques) and in the large (a coordinated set of decisions), and (3) it applies to more than one application.

For a family of applications, both consistency and identity of style are important. Consistency means that similar functions in an interface, such as choosing a single item from a list of mutually exclusive options, are represented and interact with the user similarly wherever found in an interface. Consistency does not mean that such a function is identical throughout the interface, just that where appropriate the technique is similar and where differences exist, those differences have meaning.

Consider Grudin’s example of a potentially destructive menu operation [8]. In his example, when a menu appears it defaults to the last item executed by the user. This helps the user repeat nondestructive operations such as copy or select. If, however, the last item executed was delete, then the menu defaults to another item. Grudin argues that this is an example of desirable, though inconsistent, behavior. We believe, rather, that this is consistent behavior. Consistency can mean a hierarchy of default rules and subrules for special cases. In our view, users will in fact perceive the consistency in behavior of dangerous operations even if that behavior differs from the default rule.

ITS represents the range of general to special cases as such a hierarchy of rules. In ITS, the different types of *choices* are represented with one or more rules that describe how they are similar, together with subrules that give the particular characteristics of each one of them.

What we call identity is perhaps as important as consistency. Identity is the customer’s requirement of being able to create styles that users associate with their applications. Interface style is now an area of competitive advantage in itself. The identity of excellent styles is becoming strategically important to the success of our customers.

ITS style rules support each of these requirements. They control both presentation and interaction in style. They support style in the large by representing not just individual interaction techniques but the *conditions* under which each of them should be used. And they are an open architecture that allows customers to modify standard styles and create entirely new styles of their own.

6.1 The Rule Mechanism

Each rule has a *conditions* part and a *results* part. The *conditions* determine when a rule will be executed. *Conditions* may test the type of dialog statement (e.g., *choice, list, form*). Any dialog attribute can also be tested so that, for example, a rule will fire only for those *choices* that have *kind* = *1_and_only_1*. Other *conditions* may test for particular numbers or kinds of children. Thus a rule for

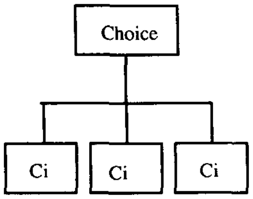


Fig. 11. The choice before style matching.

forms may fire only when a *form* has fewer than ten *fields*. Another rule that uses less screen space might fire when there are more than ten *fields*.

All of these *conditions* can be applied to a dialog object's children. A rule for *frames* that contain images can test for the presence of an image and can refer to the child that matches that test in the *results* part of the rule. That child might be positioned specially, or it might be assigned attributes to draw or interact with it differently than the other children.

All of a rule's tests must succeed for the rule to be considered for execution. Thus the current rule language supports only conjunctive *conditions*. Our experience to date does not suggest the need for more sophisticated conditionality including both conjunctive and disjunctive *conditions* within a single rule. Cases where the same *results* should occur for two different *conditions* can be handled by two rules with similar *results*. The language does support a null *condition* to catch all cases not otherwise coded. Thus if the rule base handles only the *1_and_only_1 kind* of *choice*, all others can be matched by a single rule with null *conditions*.

Five types of *result* statements can be nested with a rule:

—*Units*: to create and modify the dialog structure,

—*Set or Conditions*: to control the execution of nested rules,

—*Kids*: to collect or match on characteristics of the children of the dialog block being matched,

—*Attributes*: to set default values for style attributes, and

—*Content*: to generate dialog under style control.

6.2 Creating and Modifying Dialog Structure

Let us begin with structure. We have seen that the application expert codes a *choice* as shown in Figure 2. The set of allowed *choice items* are coded on separate *ci* statements. Before it has been matched by a rule, only the abstract *choice* exists as shown in Figure 11. The job of the style rule is now to refine this structure into one such as shown in Figure 12 by giving a template to control how each dialog node will be represented.

As style experts, assume we want all *choices* to be menus (rather than, say, entry fields). Structurally, a menu is simply a set of *choice items* arranged vertically or horizontally, perhaps under a *title*. Additional nodes are required beyond those shown in Figure 11 for the *title* and to group the *choice items* together.

Figure 13 shows a rule for creating simple menus. The *conditions* statement matches only on the type of dialog object, in this case all *choices*. The nested set

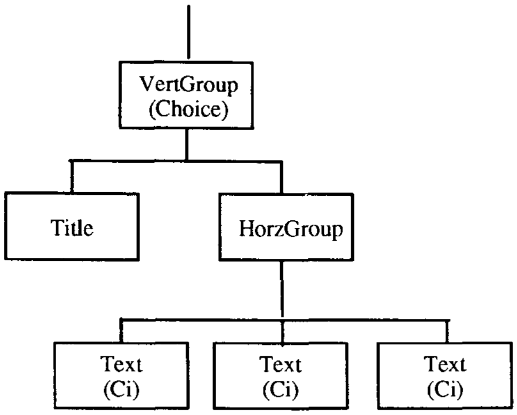


Fig. 12. Tree of units describing a simple menu in ITS. Title and HorzGroup units are added automatically by the style rule.

```
:conditions source=choice
    fire for all choices
  :unit type=VertGroup
    the block arranges the title vertically
    above the set of items
   :unit type=Title
   :eunit
   :unit type=HorzGroup
    the item is replicated for all children of
    the choice, i.e. all ci statements
     :unit type=Message, replicate=all
     :eunit
   :eunit
  :eunit

:econditions
```

Fig. 13. Style rule creating a simple menu with title.

of *units* describes the structure of the *choice* for this style. Thus the menu has a parent node that controls the layout of the *title* relative to the *choice items*. The child *unit* is *replicated* for all of the *ci* statements in Figure 11.

Unit structure is created by taking the rule as a template. Several *units* in the rule may derive from a given dialog object. For example, the single *choice* node in Figure 2 is redefined into a *vertgroup*, *title*, and *horzgroup unit* in Figure 13. Dialog attributes from the *choice* are assigned to all three *units* that derive from it. In this way style programs have ready access to all dialog attributes.

This is very helpful when the dialog node is bound to an object in the data pool. Then all *units* derived from it are bound to the same object. They are simply multiple views of the same data. Each *unit* will get update messages about changes to the data, but can decide to respond to them in very different ways.

Each parent *unit* handles the layout of its immediate children. Thus the *vertgroup unit* aligns the *title* above the set of *choice items*, and the nested *horzgroup* aligns the individual *items* horizontally. Vertical and horizontal directions themselves are defined by the *fillorder* attribute. *Fillorder* equals *colsfirst* in a *vertgroup* type and *rowsfirst* in a *horzgroup* type.

```
:conditions source=choice

    still match on all choices

  :unit type=VertGroup
    :unit type=Title
    :eunit

    pick the best one of the following subrules

    :set apply=best

    when only one choice item can be selected at a time
    (1_and_only_1), refine each item into separate dingbat
    icon and text for the radio buttons

    :conditions kind=1_and_only_1
      :unit type=VertGroup
        :unit type=HorzGroup, replicate=all
          :unit type=Dingbat
          :eunit
          :unit type=Message
          :eunit
        :eunit
      :eunit
    :econditions

    in the default case, each choice item is simply a text label
    as before

    :conditions
      :unit type=HorzGroup
        :unit type=Message, replicate=all
        :eunit
      :eunit
    :econditions
    :eset
  :eunit

:econditions
```

Fig. 14. Style rule creating a radio button menu or a simple menu, according to the kind attribute.

Now override this default rule to draw single-option choices as radio buttons. These *choices* should still have *titles*, but we now want the *items* to be stacked vertically. Each *choice item* should now include an *icon* for the button (called a *dingbat* by typesetters) and a *message*. Figure 14 shows the revised rule. The outer *conditions* statement still matches on all *choices*. The top *vertgroup* and *title units* are the same for both types of menu. *Choices* coded with *kind* equal to *1_and_only_1* are refined with separate *units* for the *icon* and *message*. Separate *units* are required since they will be decorated with different attributes. The *message* string, unlike the *icon*, redraws itself in reverse video when the user tracks into an *item*.

6.3 Matching on Characteristics of Children

In mapping from dialog to style, rules can change the order of children. The style expert may want to group all *items* together that *activate* other *frames*. Say, for instance, all *items* that transfer control to another *frame* should come first and have an urgent *background* color. The revised rule to do this is shown in Figure 15.

```
:conditions source=choice

    Name the set of kids that transfer control "go_kids".
    These are just the ci statements that have activate attrib-
    utes

  :kids name=go_kids, present=activate
  :unit type=VertGroup
    :unit type=Title
    :eunit
    :unit type=VertGroup

    each ci with an activate attribute is drawn as a text label
    with arrow icon to its right

      :unit type=HorzGroup, replicate=go_kids
        :unit type=Message
        :eunit
        :unit type=icon, shape=down_arrow
        :eunit
      :eunit

    other ci's are simply drawn as text items

      :unit type=Message, replicate=rest
      :eunit
    :eunit
  :eunit

:econditions
```

Fig. 15. Style rule reordering kids so that choice items that transfer control (i.e., have activate attributes) appear before others. In addition, they are drawn with an icon emphasizing that they are branches.

The *kids* statement collects groups of children that meet its *conditions.* In Figure 15 the rule again matches all *choices.* Any *choice items* that have *activate* attributes (*present=activate*) are included in the "go_kids" group. The name is simply an internal label used to refer to the set later in the rule.

Replicate names a set of children that should be refined using a given subtree of *units.* So, in Figure 15, the rule first *replicates* over the members of the "go_kids" group and then over the remaining children. *Rest* is a keyword that means all children not named in any group of *kids.* The *items* that transfer control are refined into three *units*: a *message*, an *icon* reminding the user that the *item* transfers control, and a parent horizontal group. Other *items* are drawn simply as *messages.*

6.4 Setting Default Values for Style Attributes

Style attributes control the appearance and behavior of *units.* Some of the attributes supported in the current system are listed in Table I. They control space and layout negotiations between parent and child *units*, presentation including font and colors, and interaction including the mapping between *physical* and *logical* events. Attributes are parameters to the style programs that render *units* on the screen. The style programs themselves are named on each *unit* by its *techid* attribute.

Table I. Selected Style Attributes Currently Supported in ITS

Name	Values	Definition
AnswerWidth	Children SelfCalc SelfFixed PopAdjust	Compute the width (depth) of this unit by summing the requests of the child units, by a nonstandard method, by using fixed estimates, or by combining the child requests and then adjusting them.
Layout	Count Space	How units should be fit into available space. By Count fits a given number of units, e.g., 3 × 4, into the space. By Space flows as many units as will fit into the available space, then wraps to the next vertical or horizontal free space.
TopMargin	⟨integer⟩	Number of screen units in top (bottom, left, or right) margin.
TechId	⟨program⟩	Name of the Style Program (P) that will draw this unit.
NomWidth	⟨integer⟩	Estimate of the Width (Depth) of this unit in screen units.

Attributes are assigned to *units* by labeling the *unit* with a *type*. Some of the style *types* we have used so far include *title*, *dingbat*, *message*, *horzgroup*, and *vertgroup*. Each *type* defines a set of attribute *values*. The set of attributes is the same for all *types*; only their *values* change among the *types*. Thus *prompt* and *title* may have a different font, horizontal justification, and color values. The *icon type* may be drawn with a raster style program, rather than the text formatter used by *prompt* and *title*. Justification and color attributes, however, can still be coded.

Style experts can create default attribute *values* and then override them later in subrules. The parent/child hierarchy of *units* defines the scope over which a *type's values* are defined. Most *types* are defined at the root of the dialog and are global. Those that are overridden at lower levels are changed only within the subtree where they are overridden.

Assume that several rules have refined the flight reservation frame in Figure 9 into the *unit* tree shown in Figure 16. Figure 17 shows how the *session* rule assigns defaults at the root node. Another rule has overridden the *font* for *title* below the *choice*. Note that the override occurs just for that individual *choice*. Other *choices* that may not have triggered the rule making this override will draw their *titles* as given in the root definition.

Two features of this design are significant. First, attributes inherit within each *type* independently of whether any *units* are actually assigned the *type*. Thus a style expert can write rules that define and override *title* attributes without concern for which interaction techniques may or may not actually use *title*. Attributes and *units* are defined separately.

Second, the *unit* hierarchy allows each parent to control *types* for its subtree. This can be done no matter how many levels exist between the *unit* that defines the *type* and the *unit* that uses it. For example, we may want the color of *titles* in a dialog box to be different from other *titles* on the screen. Pop-up dialog boxes, new windows, or areas of an existing window are all ways of representing *frames*. One of a set of alternative *frame* rules will be selected as the best match

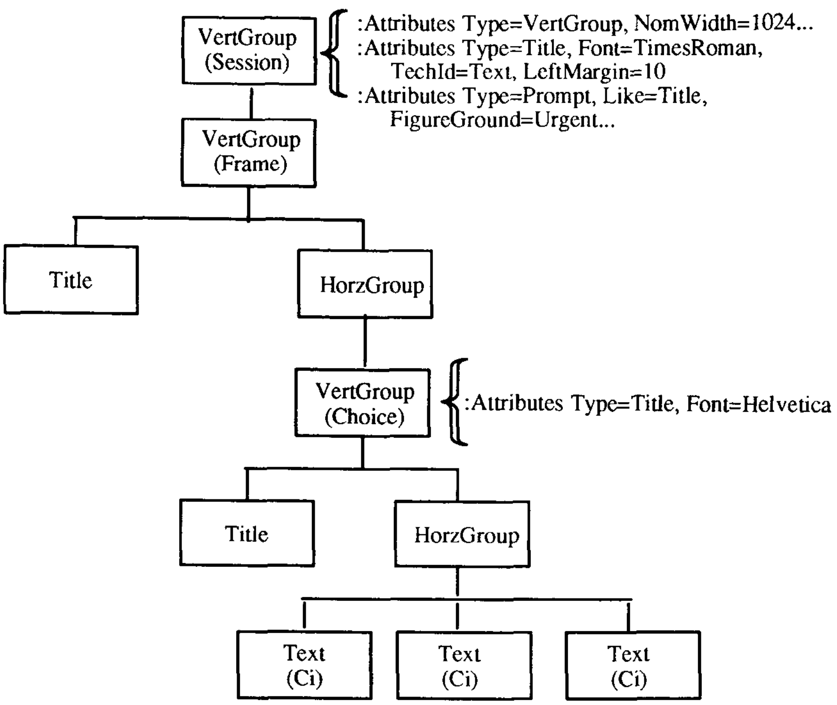

VertGroup (Session)
:Attributes Type=VertGroup, NomWidth=1024...
:Attributes Type=Title, Font=TimesRoman, TechId=Text, LeftMargin=10
:Attributes Type=Prompt, Like=Title, FigureGround=Urgent...
VertGroup (Frame)
Title
HorzGroup
VertGroup (Choice)
:Attributes Type=Title, Font=Helvetica
Title
HorzGroup
Text (Ci)
Text (Ci)
Text (Ci)

Fig. 16. The dialog tree after frame, nested subframe, and choice rules have fired. The font of titles has been overridden in this subframe.

```
:conditions source=session

    Start by defining default attributes for types used in this style. These
    defaults will be attached to the unit at the session, i.e. root, level.
    They will then inherit to lower units according to the type of each
    unit.

  :Attributes type=VertGroup, NomWidth=1024...
  :Attributes type=Title, Font=TimesRoman, TechId=text, LeftMargin=10, ...
  :Attributes type=Prompt, Like=Title, FigureGround=Urgent, ...

    Then define the unit structure for the session. We don't want titles
    here, so just create a single unit.

  :unit type=VertGroup
  :eunit
:econditions
```

Fig. 17. Any rule may assign default attributes. Those defaults are saved at the node that matches the rule. Defaults at the session level, as here, are global to the style.

for each *frame* statement in the dialog. This winning rule can override *title* colors set at higher levels of the dialog tree. The new color then is inherited by *titles* within the *frame*. Therefore, *titles* created by the same *choice* rule within different *frames* can take on different colors.

```
:conditions source=frame
  :unit type=VertGroup
    frames have titles above the set of frame contents
    :unit type=Title
    :eunit
    :unit type=HorzGroup
    Frame contents are arranged horizontally. The type isn't
    assigned here, but in other rules that match later on the spe-
    cific frame contents.
      :unit replicate=all
      :eunit
    :eunit
    each frame has Help and Quit buttons added automatically
    by the rule. The choice will be matched itself later as any
    other block.
    :content
      :choice purpose=useraction
        :ci message=Help, activate=AppHelp
        :ci message=Quit, action=shutdown
      :echoice
    :econtent
  :eunit
:econditions
```

Fig. 18. This rule creates Help and Quit buttons automatically in each frame.

6.5 Generating Dialog within Style

Finally, style rules can themselves generate dialog and delay committment to its style. A rule, for example, can add standard command buttons to each *frame* automatically. The rule in Figure 18 codes the same *choice* statements that could be used by an application expert to create Help and Quit buttons. Thus the *frame* rule does not have to say how they should appear, just that they should appear in each *frame*. The appropriate *choice* rule will fire later to create the button style. This technique has been used to generate "Yes" and "No" *choice items* for fields with Boolean data types. Rules have also automatically generated pull down menus for text entry *fields* where a fixed set of values is allowed for the *field*.

7. THE STYLE PROGRAM LAYER

7.1 Space Negotiations

Style programs, like actions, must respond to a standard set of messages from the dialog manager. The most important messages are sent to request screen space estimates, to paint to the screen, and to inform the style program when the user indicates *activation, selection,* or *execution*. The style expert can set attributes that control whether style programs receive each message or take the default response.

The problem in space negotiation is to request and then allocate screen space for each *unit* based on its requirements and the requirements of its children.

Space negotiation is a three-step process that balances performance with the need for high-quality layout. The first step is to determine the space requirements of each *unit*. The second, and optional, step is for the parent to adjust the space required for children. The third step assigns a specific location to each object that will be displayed.

Space requirements are determined by a depth first, left-to-right traversal of a *frame*. The first step occurs when moving downward. During this phase each *unit* is asked to respond with its own space requirements. Many *units* defer this question until their children have responded to it first. The second step of space allocation occurs during the upward search (looking for the next right-hand sibling) of each subtree. During this step each *unit* is asked to integrate and possibly adjust the space requirements of its children. An example would be a *unit* asking for enough space to give each of its children the same depth as that of the deepest child.

Space assignment is done by a second tree traversal. During this phase each *unit* receives its screen assignment, both location and size. This process is top down. From its allocation, each *unit* reserves what it requires and allocates the remaining space to its children. Each *unit* can, of course, have a different policy for allocating space.

7.2 Other Operations

Each of the above processes place their results only in documented attributes on the *unit* tree. This means that these results are publicly visible and hence are easily available for reuse. One such use is to be queried by children needing to make their own space or drawing decisions. For example, a text object that wants to set its font size relative to a parent can respond to a repaint message by setting its size as some increment on the size attributes in the parent block.

Mouse tracking or touch-paint location also benefits from standardized attribute definitions. The run-time manager can provide a default tree traversal for correlating mouse position with a displayed object. In ITS this default mapping is determined by the smallest display rectangle surrounding the mouse position. As such, mouse event mapping is similar to that in the X toolkit. However, alternative mappings are often required, as when a parent wants to redirect a mouse event to a child different than the one directly under the mouse (as in the Andrew toolkit). An attribute on each *unit* determines whether the *unit* wants to assume responsibility for pointer correlation within its borders. ITS thus combines a strong default method and the flexibility to override this method with alternative ones.

8. THE EXPO 92 VISITOR INFORMATION SYSTEM

To test the ITS application development tools, we are using them to build a visitor information system for the 1992 EXPO in Seville, Spain. ITS allows us to build this system iteratively as a series of progressively more complete prototypes. These prototypes are functioning applications and test each layer of ITS: actions, dialog, style rules, and style programs.

8.1 What Is the EXPO Visitor Information System?

The EXPO visitor information system will offer information and services to an estimated 20 million visitors from April through October, 1992. Information to be provided includes maps and directions for navigating around the two square kilometer site, up-to-the-minute schedules of events, and background information on countries and organizations participating in EXPO. In addition, the system will provide visitors with services beyond those traditionally found in information kiosks. These include person-to-person and group electronic messaging, automated restaurant reservations, public opinion polling, and locators for lost family members. Since we also believe the kiosk should be fun to use, it will provide a variety of standalone and networked games for use during offpeak hours.

8.2 What Are the Requirements for the EXPO Visitor Information System?

Developing a style for EXPO is a good test for ITS since both the application and the developers pose difficult challenges. The application must support several languages, including several dialects of Spanish, as well as English, French, and German. Most visitors will be extremely inexperienced computer users. Since we expect them to spend only a very few minutes at the information system, it must be operable with little or no training. To be visually attractive it must be of world-class graphic design quality and operate with multiple types of media: high-quality text, graphics, and images.

The developers, too, pose challenges the ITS tools must help meet. Our development team is distributed between two locations in Seville and New York. ITS must help integrate application functions developed in both locations. In addition, we maintain working prototypes in other locations for demonstration, including Madrid and visitor centers on the EXPO construction site. These prototypes must be attractive enough even in early versions to promote the desired image of a quality product in the works.

8.3 Dialog and Actions

The starting *frame* of the application is shown in Figure 19. Its appearance in the EXPO style is shown in Figure 20. The user can take an electronic walk through EXPO by selecting hotspots in maps and images at the left of the screen, take an opinion poll, draw a picture using a finger-painting program (Figure 21), send or receive electronic mail (Figure 22), or make a restaurant reservation.

In Figure 19, the "startexpo" and "stuff_caption" actions are called when the *frame* initially appears on the screen. "Startexpo" reads several data files to initialize the image network and opinion poll. "Stuff_caption," when called with the *frame initialize* message, sets up the caption of the current image in the image title area. Later, when "stuff_caption" is called with *select* messages, it replaces the caption with short titles describing each hotspot as they are selected on the image. Since no actions are coded on the messaging or opinion poll choice items, when *executed* they simply transfer control to the *frames* given by their *activate* attributes.

The image network panel is built from a *form* and two *lists* grouped together as a *subframe*. *Version = fit* is recognized by style rules to fit the *subframe* into

```
:frame id=basic_frame, level=a, action=startexpo, listname=nav_list,
      table=eachframe
  :choice message="Explore picture or", purpose=navigation, version=icon
    :ci message=Poll, value="opinion.bmp", activate=opinion_poll
    :ci message=Paint, value="paint.bmp", activate=finger_painting
    :ci message=Meals, value="meals.bmp", activate=reservations
    :ci message=Read,  value="read.bmp", activate=read_message
    :ci message=Send, value="send.bmp", activate=send_message
    :ci message=Vote, value="vote.bmp", activate=voting
  :echoice

  :frame version=fit, purpose=group
    :form version=almost
      :fi table=eachframe, field=ndx_title, purpose=overview
      :fi table=imagetext, field=label, purpose=feedback
    :eform

    :list listname=hotlist, datatype=image, number=11, table=eachframe,
      field=pic_file, message=the picture, action=stuff_caption
      :li field=index, action=newimage
    :elist

    :list listname=piclist,number=5, datatype=set_of_image
      :li field=small_pic_file,action=poptoimage
    :elist
  :eframe

  :frame purpose=navigation
    :choice structure=disjoint, purpose=useraction
      :ci message='Empezar en espanol', action=lang_and_size,value=spn
      :ci message="Let's Start in English", action=lang_and_size,value=eng
    :echoice
  :eframe
:eframe
```

Fig. 19. Dialog statements of the beginning EXPO frame as coded by the application expert. A choice block allows the user to select an EXPO service. Value attributes on choice items name the icon to be displayed with each choice. A form gives title and caption information for the image and its list of hotspots. A second list displays the route the user has traversed through the tree of images about EXPO.

the left-hand two-thirds of the screen and to be sure it appears there regardless of the order in which it is coded in the dialog statements. *Purpose* = *group* indicates to this style that all contents of the *subframe* are related, and hence they are drawn together on a gray background with no separating gaps or lines. *Structure* = *disjoint* is coded, in contrast, on the *choice* between Spanish and English. The EXPO style therefore separates these *choice items* visually on the screen.

Within the image *subframe*, the *form* contains two *form items* for the image title and caption. *Version* and *purpose* attributes are used here to guide style in choosing appropriate fonts, margins, and separating lines between the title and caption. The *list* with *datatype* = image contains the image and its set of hotspots. The image file name is found in the *field* "pic_file" in the *table* "eachframe." An image's hotspots are stored as *records* of the *list*. Each *record* contains *fields* for

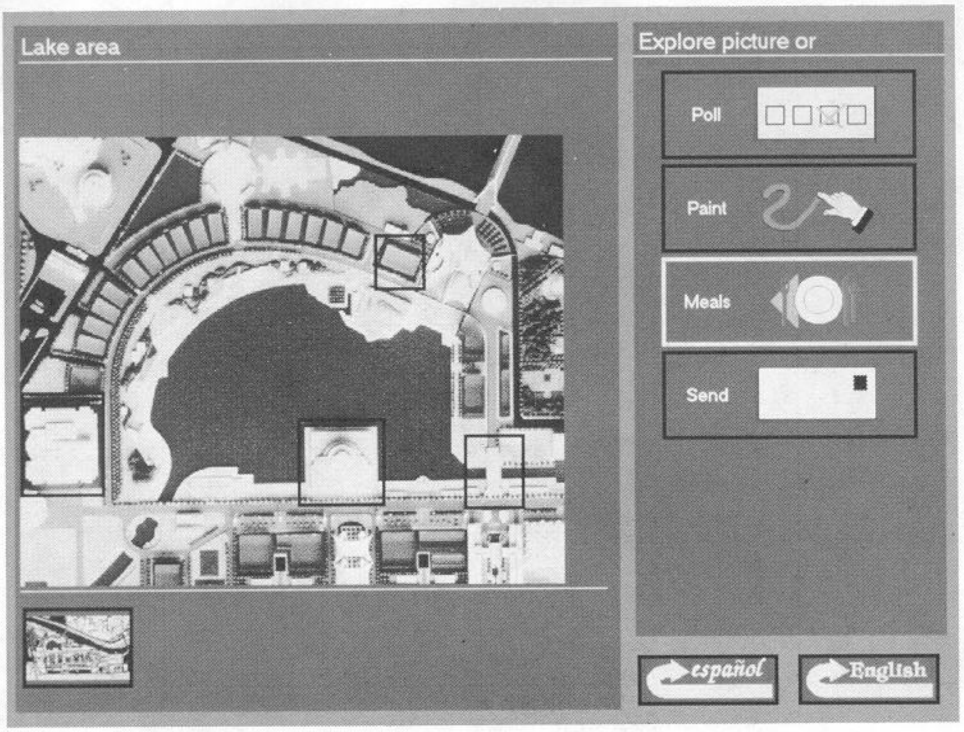


Fig. 20. Maps and images with selectable hotspots provide an electronic walk through EXPO.

the position and size of the hotspot. These *fields* are used by a style program to size and position the hotspot on the image.

To help users navigate, miniature pictures show the path of images taken from the root of the network. In Figure 19, the list with *datatype* = set-of-image stores this breadcrumb trail. This *list* is simply a view of the internal picture stack maintained by the "newimage" action. Each time "newimage" is called by *executing* a hotspot, it pushes a small picture file name onto the stack. "PopToImage," the action coded on *items* in the breadcrumb *list,* pops the stack to return directly to any image on the path to the root.

Note that all navigation through the image network is coded within a single *frame* in the dialog. No *activate* attributes transfer control outside that *frame* when users pick hotspots or pop the image stack. Rather, actions simply modify the contents of the data pool to point to new images and hotspots. Update messages propagated by the dialog manager to style programs cause repaints of only those parts of the screen that need to change to show the new image. All of the borders, title bars, and menu controls remain unchanged by these repaints. The effect minimizes disruption for the user.

8.4 Style Programs Needed to Implement EXPO

Eight different style programs are needed to control the screen in Figure 20. A text formatter draws titles, menus, command buttons, and explanatory text.

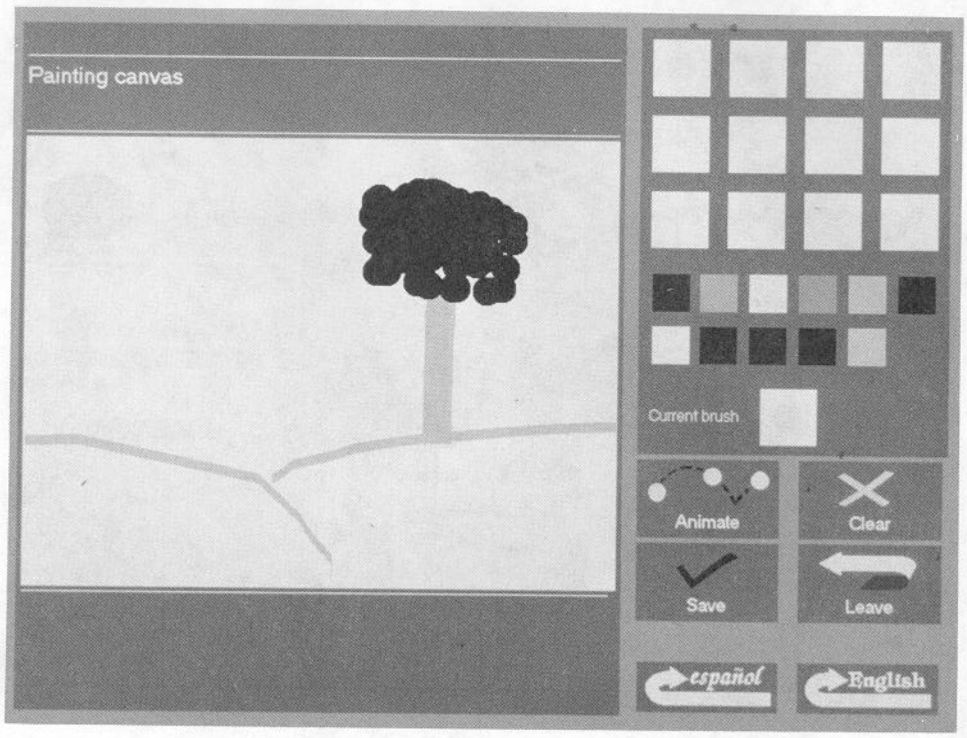


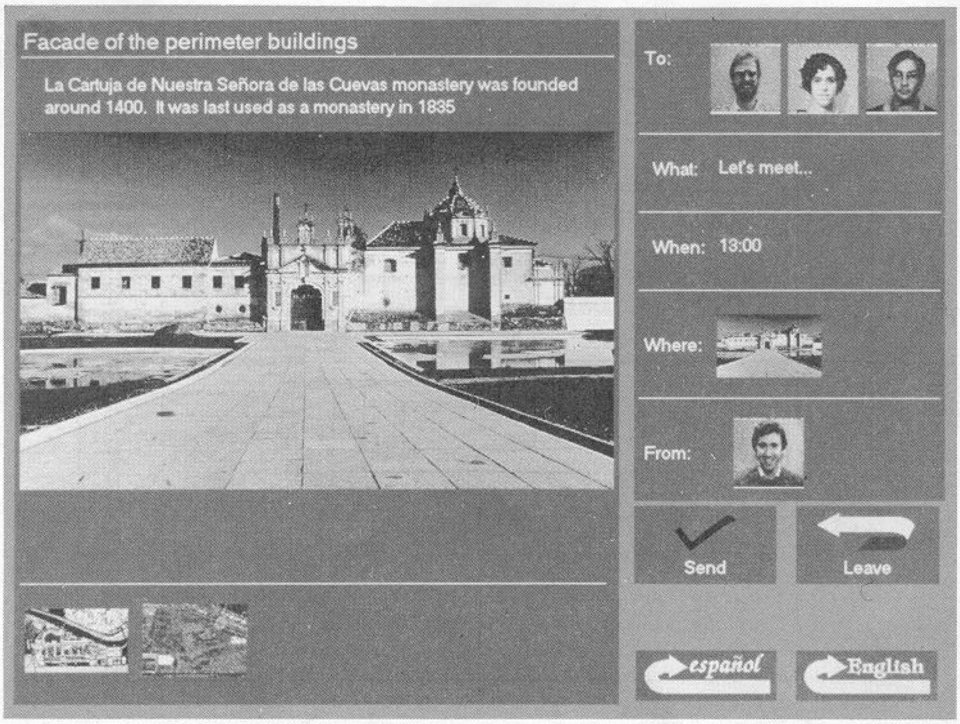
Fig. 21. Finger painting using a touch screen.

Pictures are displayed from true color images mapped offline to a 256-entry color table. Drop shadows and other decorative borders are computed from scalable bitmap definitions so they can wrap around units of any size. Selectable hotspots are computed from position data and superimposed over images. The same style program that highlights each hotspot is wrapped around the small images below the picture and around each selectable menu item. To cause all items to highlight when the user touches anywhere on the screen, a style program at the root of the screen's *unit* tree propagates finger down and up events to all interested *units* below it. Finally, rectangular layout is controlled by a style program using a gridded design [4]. In gridding, style experts specify the number of columns in a *frame. Units* within that *frame* are sized as multiples of the column width. A gridded *frame*, by having only a small number of *unit* sizes, typically looks more balanced than one with arbitrary sizes.

The text, gridded frame layout, picture, and drop shadow programs were reused from the toolkit. The programs for positioning hotspots, drawing their borders, and propagating touch events were added during the implementation of EXPO.

8.5 EXPO 92 Rules

The structure of the map *frame* is shown schematically in Figure 23. *Units* shown in bold correspond to the original content coded by the application expert in Figure 19. Other *units* are added automatically by style to control layout, to


Facade of the perimeter buildings

La Cartuja de Nuestra Señora de las Cuevas monastery was founded around 1400. It was last used as a monastery in 1835

Fig. 22. Sending a message to family members. The "where" field of the message at right tracks the currently displayed image. "When" and "what" fields are set by graphical dialogs appearing in the image area.

refine original content nodes into more elaborate structure, and to create content not originally coded by the application expert.

Unit 1 in Figure 24 is the *frame coded* by the application expert. This root *unit* controls the entire group of hotspots contained within the *frame.* Hotspots are controlled from the root of the *frame* so they are all drawn and erased together wherever the touch occurs.

Units 2, 3, and 4 are created automatically by the rule in Figure 24 to control the placement of the image and language buttons within the *frame. Unit* 2 creates a horizontal group across the entire screen. *Unit* 3 at the left contains the image *subframe (unit* 5) and its three children, *units* 6 through 8. *Unit* 4 is a vertical group of the remaining *frame* contents. The buttons for changing languages are forced to the bottom of this right-hand column. Each of these objects are further refined by the firing of appropriate rules elsewhere in the style.

9. DISCUSSION

9.1 Layering not Separation

User Interface Management Systems traditionally have been used to build generic interface controls such as menus, buttons, and scroll bars. Large "client areas" have been left as terra incognita, into which the application has been allowed to

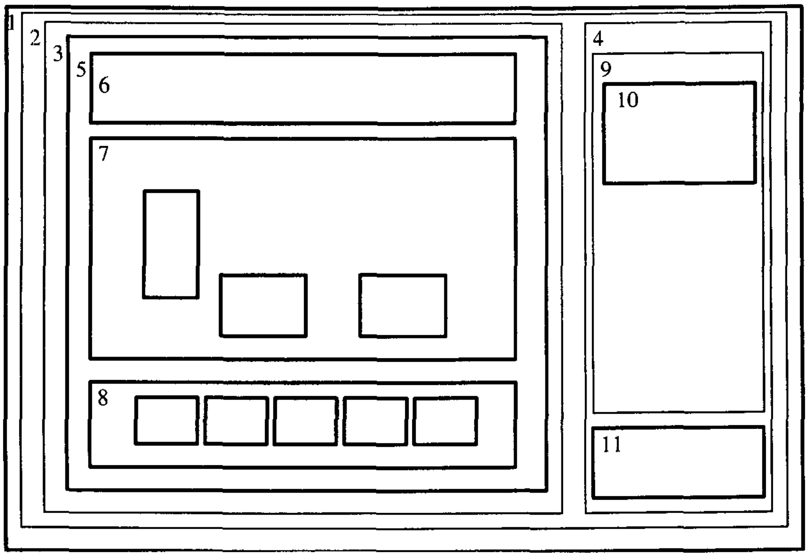


Fig. 23. Structure of map frame. The rule moves the image to the left of the screen as unit 7. Selectable hotspots are positioned by unit 7 according to locations provided by the application. Other frame contents are arranged vertically as children of unit 9. Only units in bold are coded by the application expert; others are added automatically by style rules to control layout and behavior.

write at will to handle more application-specific interaction techniques. Thus, the modularity gained by defining an interface externally from the application is lost for large segments of the interface. This is true because the application defines the style of interaction within the client area. Changes to the style of that part of the interface require changes to the application.

The alternative is to generate all of the interface within the UIMS and to eliminate the client area. This leads to the view that information about the application, in particular about its data types, is not hidden from style rules. Rather than separating application from style, ITS applications delay commitment to style. Delayed commitment means that application layers are not allowed to talk about style. Once we have committed to a particular style, however, application information is freely available to style rules and programs.

As an example, in the EXPO visitor information system users can select the time of day for a restaurant reservation. Time is a data type defined within the EXPO application as a pair of integer ranges for hours and minutes. By recognizing the use of the time data type, a style rule draws the hours and minutes in the correct sequence and automatically inserts a colon between them as is common in digital clocks. If the dialog further indicates that a given usage of time is for display only (by coding *usercan* = *show*), the rule reduces the separation between the *fields*, since space is no longer required for selectable hotspots around them.

```
:conditions source=frame
     Start by naming the contents of the frame that we want to handle spe-
     cially, these include the buttons to change languages (purpose=naviga-
     tion)...
:kids name=bottom, source=frame, purpose=navigation
     ...and the image panel that should fit into the left side of the screen.
:kids name=fits, version=fit
     Then define the unit tree for the frame.
  :unit type=Group
     Type=Group creates a unit to control the group of all hotspots on the
     screen. Draw messages are sent to each one from here, at the root of the
     frame, so they all appear and disappear together.
    :unit type=HorzGroup
     The horizontal group arranges an image panel at the left of the screen...
      :unit type=VertGroup
        :unit replicate=fits
        :eunit
      :eunit
      :unit type=VertGroup
     ...then rest of the frame contents are stacked vertically at the right side of
     the screen.
        :unit type=vertgroup, extradepth=accept
     This extra vertical group accepts extra depth, while the language buttons
     below reject it. That forces them to the bottom of the screen.
          :unit replicate=rest
          :eunit
        :eunit
     Buttons for changing languages come at the bottom of the right hand col-
     umn.
        :unit replicate=bottom, extradepth=reject
        :eunit
      :eunit
  :eunit
:econditions
```

Fig. 24. Style rule for location finder frame in EXPO. Other rules will fire for the individual contents of the frame, including the list for hotspots and the choices for personal services and languages.

9.2 Comparison with General-Purpose Production Systems

Style rules are not simple to write and understand in ITS. They are, however, much simpler to write and understand than general-purpose expert system languages such as OPS-5. ITS rules are specialized for generating interfaces in their syntax for the structure of interaction techniques, in the simple depth-first traversal used to match blocks in an application, and in the strategies supported for resolving conflicts among multiple rules that may match a given block.

The nonprocedural syntax is the most notable feature of how rules specify *units* in an interaction technique. Rules do not say how to transform dialog into specific styles. Instead, rules require the style expert to say only what the final structure should look like. The job of the ITS system is to transform the application expert's abstract tree into that style.

To be usable, the process of executing rules must also be simple. General-purpose expert systems execute rules until no new elements are added to working memory and all rules that match working memory elements have been fired. To simplify the process ITS considers each *unit* only once during a depth-first traversal of the dialog. The rule cycle consists of three steps: (1) select the next *unit* in the depth-first traversal, (2) fire one or more rules that match the selected *unit* according to a conflict resolution strategy described below, and (3) replace the *unit* with the refined subtree created by the rules just fired. Since each *unit* is replaced in the tree with a subtree refined according to the current style, rules often add new *units* that are visited for matching themselves later on. In this way rules control children but not siblings or parents of a selected node.

A typical style file contains different rules for each type of dialog statement. In addition, each statement (such as a *choice*) typically has several variant styles depending on the attributes coded on it by the application expert. *Choices*, for example, may appear differently depending on their *kind*, *purpose*, or *emphasis* attributes. How are conflicts resolved and the appropriate rule selected?

Two kinds of conflict resolution have been implemented in ITS: Execute the best rule that matches the *unit* under consideration, and execute all rules that match it in the order from most general to most specific. These two possibilities allow both for selection among competing alternatives (execute the best rule) and execution of a series of rules that progressively override defaults or make independent decisions (execute all matching rules in sequence).

The scope over which conflict resolution takes place is also more constrained in ITS than in general expert systems. ITS rules are organized into a hierarchy of sets. A set of alternative rules for *choices* might be nested inside a single set. This set itself would be contained in a set of sets for other attributes that rules might match on such as the *emphasis* or *state* of a data value. The organization of the two sets is shown graphically in Figure 25. Consider how a dialog *choice* will be processed by these rules. *Apply = each* tells the top-level set to fire each the two sets nested beneath it. As many rules in the left-hand set as match should be executed, since they make decisions on orthogonal attributes. Only one of the rules in the right-hand set should fire, since each applies to a different type of object.

9.3 Other Application Experience

In addition to EXPO '92 we have focused our application work on workbenches for the application expert. Two dialog workbenches are currently being built. One presents the dialog as a spreadsheet whose rows are dialog statements and columns contain attributes. The dialog is edited by typing into cells of the matrix or by selecting from ranges of values automatically maintained about the dialog being built.

The second workbench is based on two abstractions of the dialog: the control graph of *activations* among *frames* and the dialog tree within each *frame*. A *frame* can be selected from the control graph and its tree drawn in another window.

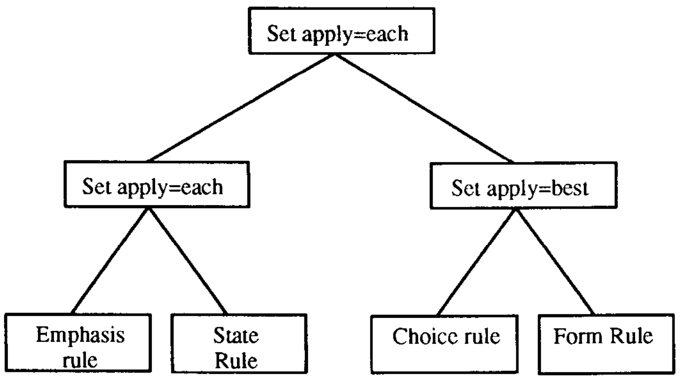


Fig. 25. Tree of sets of style rules.

Details of each object in the *frame* are shown in attached panels as the mouse tracks into nodes in the tree.

The two workbenches are complementary in that the first is a good tool for editing "in-the-small;" the second helps to view the overall structure of the dialog "in-the-large." We hope to combine them soon into an integrated tool.

9.4 Feedback into Improved Rules

We have described the benefits of generating interfaces automatically. Executable style rules greatly reduce the time required to build an interface. Given an excellent style, they also can improve the quality of an interface over that produced by untrained designers. But it is often argued that automatic style will not beat the quality of interfaces individually handcrafted by excellent designers. Is it not a good idea to allow designers to hand-tune automatically generated designs to bring them up to snuff?

While it is tempting to do so, it is a bad idea for two reasons. First, styles need to be tuned when they are missing knowledge that could have caused them to generate a better design in the first place. By manually tuning the output of the rule-base we would miss an opportunity to add the missing knowledge. No one else would benefit from our work.

Second, we ourselves would benefit from our hand-tuning only temporarily. The next time we modified the application, and regenerated its interface, we would have the opportunity to hand-tune the style again. Since maintenance is a fact of life, we need instead to write the new rules that let us use the improved style as often as needed.

Nonetheless, style workbenches do have a role in aiding interface design. Understanding and modifying the behavior of styles can be complex problems. A workbench could help by searching for rules whose behavior might be altered by style changes. The workbench could also provide visual examples of the results produced by firing rules. Direct manipulation changes to these displays, for example, to change the font of a title could be used together with prompts for entering the appropriate conditions of the changed rule.

9.5 Using Applications in Different Styles

We used to talk about writing styles that were truly separate from applications. These generic styles would work with all applications. We now believe that, while possible, this is not always desirable. This shift of view is captured by the shift from speaking about a separation between application and style in favor of a delayed commitment to style.

Styles with rules that match only on the fixed set of dialog statements and attributes (Figure 7 and Figure 8) work equally well when used with any application. These statements and attributes are standard components of all applications and so styles can be secure in matching on them.

Styles with rules that match also on application data types will not work equally well when used with any application. They have specialized knowledge about how to interact with particular types of data. On the other hand, they need not work any less well with other applications than do the generic styles. For an application with data types not matched by the style with application-specific rules, the generic rules will still fire. Rules that select for particular types of application data only improve a style.

9.6 Behavioral Experience

To understand the usability of our tools outside of our immediate group we have concentrated on two activities: teaching classes on interface design using ITS and transferring ITS to external customers. This experience has focused so far on the action and dialog layers, rather than the rule and style program layers. To date, we have taught ITS to roughly 25 people from within and outside IBM and from widely varying backgrounds. These classes are week-long, hands-on working sessions where students arrive with their own design problems and leave with working interfaces.

ITS has also been transferred to one customer external to IBM. A major insurance company has sent several of its programmers and style experts to our course. Subsequently, these programmers, working independently, successfully coded all of the actions and dialog for several prototype applications at their home locations.

CONCLUSIONS

We have described the four layers of the ITS architecture. The action layer implements back-end application functions. The dialog layer defines the content of the user interface, independent of its style. Content specifies the objects included in each frame of the interface, the flow of control among frames, and what actions are associated with each object. The style rule layer defines the presentation and behavior of a family of interaction techniques. Finally, the style program layer implements an extensible toolkit of objects that are composed by the rule layer into complete interaction techniques. Example style programs include routines to format text, render raster images, and arrange units in gridded rectangular layouts.

Our experience has shown that ITS is capable of implementing highly interactive applications. Highly interactive applications respond intimately to user actions based on the current application state. They require a tighter coupling between the application and interface than was provided by early UIMS. ITS provides this coupling by allowing actions to update the data pool on any user event and by reflecting these updates to all views of the changed data automatically.

Our experience has also suggested a number of extensions to both the dialog and rule layers. Many of the rules in the EXPO style, for example, match on the type of *list* or *form* being rendered. Examples include the time data type discussed above, as well as "labeled images" (an image together with descriptive text), "messages" (fields in an email message), and restaurant reservations.

The dialog layer will be extended with a more powerful type language. A type hierarchy will allow new types to be created as extensions of existing ones. The message type, for example, might be defined as a specialization of a flat collection of fields. Structured types will also be supported to help group fields in complex data such as messages and reservations.

The style rule layer will be extended at the same time to support the new type mechanism. The matching algorithm currently recognizes a type only by exact match. To avoid a proliferation of rules specific to single data types, the algorithm will be extended to recognize generalizations of a type if no rules are otherwise found. An instance of the message type, for example, might be matched by a more general rule for a flat collection of fields if no rule is given specifically for messages.

ACKNOWLEDGMENTS

The ITS System has been developed by the combined efforts of our entire group, including Chris Cesar, Tom Cofino, Lauretta Jones, Joe Kesselman, Rich Mushlin, Susan Spraragen, and Jacob Ukelson.

The EXPO '92 visitor information system is being developed jointly with Paco Curbera, Angel Llopis, Javier Martinez, Juan Rojas, and Luis Sopena of IBM's Madrid Scientific Center.

REFERENCES

1. APPLE COMPUTER, INC. *Human Interface Guidelines: The Apple Desktop Interface.* Apple Programmers and Developer's Association. Renton, Wash, 1986.
2. BENNETT, W., BOIES, S., GOULD, J., GREENE, S., AND WIECHA, C. Transformations on a dialog tree: Rule-based mapping of content to style. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (Williamsburg, Va., Nov. 13–15). ACM, New York, 1989, pp. 67–75.
3. CARDELLI, L. Building user interfaces by direct manipulation. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software* (Banff, Alberta, Canada, Oct. 17–19). ACM, New York, 1988, pp. 152–166.
4. FEINER, S. A grid-based approach to automating display layout. In *Proceedings of Graphics Interface '88* (Edmonton, Canada, June). Morgan Kaufmann, Palo Alto, Calif., 1988, pp. 192–197.
5. FEINER, S. An experiment in the automated creation of pictorial explanations. *IEEE Comput. Graph. Appl. 5*, 11 (1985). 29–37.
6. FOLEY, J., GIBBS, C., KIM, W., KOVACEVIC, S. A knowledge-based user interface management system. In *Proceedings of CHI 1988* (Washington, D.C., May 15–19). ACM, New York, 1988, pp. 67–72.
7. FOLEY, J. Personal communication, 1989.

8. GRUDIN, J. The case against user interface consistency. *Commun. ACM 32,* 10 (Oct. 1989), 1164–1173.
9. HARTSON, H. R., AND HIX, D. Human-computer interface development: Concepts and systems for its management. *ACM Comput. Surv. 21,* 1 (1989), 5–92.
10. HAYES, P., SZEKELY, P., AND LERNER, R. Design alternatives for user interface management systems based on experience with COUSIN. In *Proceedings of CHI '85* (San Francisco, April 14–18). ACM, New York, 1985, pp. 169–175.
11. HUDSON, S. Graphical specification of flexible user interface displays. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (Williamsburg, Va., Nov. 13–15). ACM, New York, 1989, pp. 105–114.
12. IBM CORP. *Systems Application Architecture, Common User Access Panel Design and User Interaction.* SC26-4351-0. Dec. 1987.
13. KRASNER, G., AND POPE. S. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.* (Aug./Sept. 1988).
14. LINTON, M., VLISSIDES, J., AND CALDER, P. Composing user interfaces with interviews. *IEEE Comput. 22,* 2 (1989), 8–22.
15. MACKINLAY, J. Applying a theory of graphical presentation to the graphic design of user interfaces. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software* (Banff, Alberta, Canada. Oct. 17–19). ACM, New York, 1988, pp. 179–189.
16. OPEN SOFTWARE FOUNDATION. *OSF/Motif Style Guide, Revision 1.0.* Open Software Foundation, Cambridge, Mass.
17. PALAY, A., HANSEN, W., KAZAR, M., SHERMAN, M., WADLOW, M., NEUENDORFFER, T., STERN, Z., BADER, M., AND PETERS. T. The Andrew toolkit: An overview. Tech. Rep. Carnegie Mellon Univ. Information Technology Center.
18. PFAFF, G., ED. *User Interface Management Systems.* Springer-Verlag, Berlin, 1985.
19. SCHUMUCKER, K. *Object-Oriented Programming for the Macintosh.* Hayden Books, Hasbrouck Heights, N.J., 1986.
20. SEI. Serpent, a user interface management system overview, version 1. Tech. Rep. 89-UG-2. Carnegie Mellon Univ./Software Engineering Institute, Feb. 1989.
21. SIBERT, J., HURLEY, W., AND BLESER, T. An object-oriented user interface management system. In *Proceedings of SIGGRAPH '86* (Dallas, Tex., Aug. 19–22). ACM, New York, 1986, pp. 259–268.
22. SINGH, G., AND GREEN, M. Chisel: A system for creating highly interactive screen layouts. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (Williamsburg, Va., Nov. 13–15). ACM, New York, 1989, pp. 86–94.
23. SUN MICROSYSTEMS AND ATT. *Open Look, Graphical User Interface Application Style Guidelines.* Addison-Wesley, Reading, Mass.,
24. SZEKELY, P. Separating the user interface from the functionality of application programs. Ph.D. thesis, Dept. of Computer Science, Carnegie-Mellon Univ., 1988.
25. VANDER ZANDEN, B., AND MYERS, B. Automatic, look-and-feel independent dialog creation for graphical user interfaces. In *Proceedings of CHI '90* (Seattle, Wash., April 1-5). ACM, New York, 1990, pp. 27–34.
26. WIECHA, C., AND BOIES, S. Generating user interfaces: Principles and use of ITS style rules. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology* (Snowbird, Utah, October 3–5). ACM, New York, 1990, pp. 21–30.
27. *X Toolkit Intrinsics Programming Manual, The X Window System Series,* vol. 4. O'Reilly and Associates, Sebastobol, Calif., 1989.