GENERATING HIGHLY INTERACTIVE USER INTERFACES

Charles Wiecha, William Bennett, Stephen Boies, and John Gould

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

ABSTRACT

Developers of User Interface Management Systems (UIMS) have demonstrated that separating the application from its user interface supports device independence and customization. Interfaces produced in UIMS are typically crafted by designers expert in human factors and graphic arts. Little attention has been paid, however, to capturing the knowledge of such experts so that interfaces might be automatically generated by the application of style rules to additional applications. This paper considers how toolkits and style rules can be structured so that the resulting interfaces take advantage of the best human factors and graphic arts knowledge, and are consistently styled.

KEYWORDS: User interface management systems, interface consistency, graphic interfaces

INTRODUCTION

User interface management systems (UIMS) have traditionally separated applications into three components: application actions, a dialog manager, and user interaction routines. Structured as a set of service routines, the same application can be used with a variety of different dialogs or interaction routines as required.

There are two important shortcomings in this model. First, the interface designer, an expert in human factors and graphic arts, typically creates each interface by directly coding and assembling interaction routines from a toolkit. We believe that the expertise of such designers can be encoded in style rules so that interfaces can be automatically and efficiently generated for varying user groups and hardware environments.

Second, many interesting interfaces depend on extensive knowledge of the application. If separating the application from the interface also removes that knowledge, then such highly interactive interfaces will not be possible. Retaining knowledge about application data types allows the interface to interact more intelligently with users while remaining separate from the application. An interface having only abstract type knowledge can, in addition, be reused by different concrete data types [10]. As shown below, a single tree view might be used for organization charts, file directories, and documents.

FOUR ROLES IN SYSTEM DEVELOPMENT

We are implementing these ideas in the Interactive Transaction Systems (ITS) project at IBM Research. The structure of an application implemented in ITS is shown in Figure 1. An *application expert* defines the content of an application independent of its style. Content includes data types to be used by the application, and tables which store data being shared between the application and the interface. The application expert also specifies the contents of each frame, the flow of control among frames, and the application actions that should be executed at each point in the dialog. An *application programmer* implements actions called by the dialog.

A dialog compiler merges information from the type, table, and dialog files. The parse tree produced by the dialog compiler is passed as input to the style compiler. Rules,

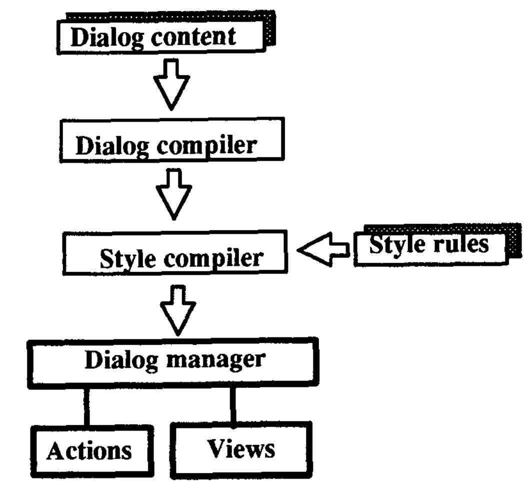
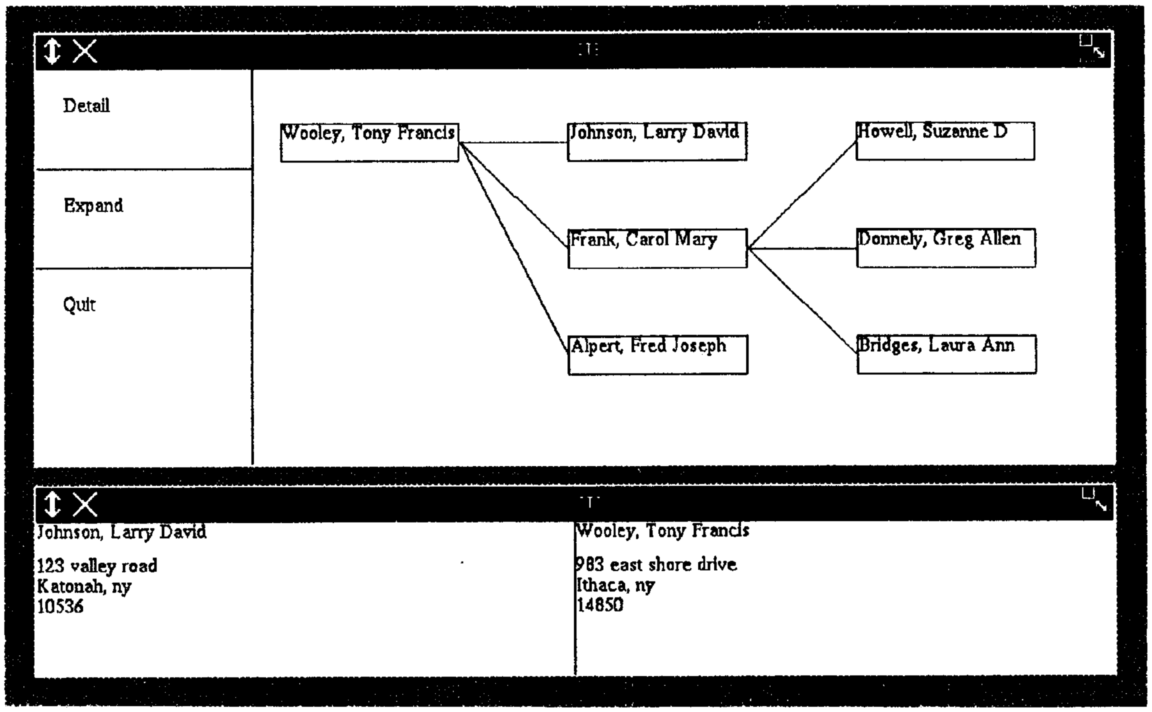


Figure 1. Structure of an ITS application.



Johnson, Larry David
123 valley road
Katonah, ny
10536

Wooley, Tony Francis
983 east shore drive
Ithaca, ny
14850

Figure 2. Screen display of employee browser

coded by the *style expert*, are used by the style compiler to attach interaction techniques and views from a toolkit to each node of the parse tree. In the current implementation all style decisions are made at compile time based on static attributes of the three input files. Finally, at run–time, the dialog manager interprets the fully attributed parse tree by executing actions and calling the view manager to display views coded by a *style programmer*.

The style expert is concerned with two questions: which views should be used for each dialog block, and what attribute values should be set for each view? View attributes might include font or line style choices, or the order in which first, middle, and last names should appear in a name field. The role of the style expert is not, however, to make these decisions for each specific application. Rather, the style expert writes rules which express the regularities of a style. Given those rules in an executable form, they can be applied to many different applications to create interfaces of a specific style rapidly and automatically. Related work on rule–based interface generation includes APEX [1], APT [5], and Perlman's layout axioms [8].

Note that the separation of content and style is a refinement of the traditional separation of interface from application in user interface management systems. Kamran's Abstract Interaction Handler [4] is an early example of this idea but implements style rules in conventional, imperative, languages. We have refined interface design into four separate expert and programmer roles. Application and style experts write executable specifications in languages designed to be usable by nonprogrammers. Languages for application and style programmers remain imperative, to provide for generality in implementing functions needed by the interface.

TOOLS FOR APPLICATION EXPERTS AND PROGRAMMERS

The language used by application experts supports the separation of content and style by providing a set of style independent tags to describe each component of a dialog. The language is based on an analogy with tagged document formatters such as GML [3] and SCRIBE [11]. Tagged documents are structured as a hierarchy of components such as chapter, section, paragraph, and footnote. Each component is then associated with formatting properties contained in a separate style data base. In ITS, dialog is described as a hierarchy of tags such as frame, form, list, and choice. Each tag is associated with formatting routines by executable rules written for a particular style. [1]

We'll illustrate the use of dialog tags by implementing a browser for a hierarchically structured employee data base. The browser shows general and detailed views of selected employees, which in the style described below are drawn as the tree and panels in Figure 2.

1. The dialog and rule syntax reflect minor revisions for readability and extensions currently being implemented.

```
    first, a name type used below...
data type=fullname, structure=disjoint
  di field=last, type=string
  di field=middle, type=string
  di field=first, type=string
edata
    ...then the employee data type
data type=employee, structure=disjoint
  di field=name, type=fullname, emphasis=special
  di field=serial, type=integer
  di field=address, type=us_address
  di field=manager, type=employee
edata
```

Figure 3. Employee data type definition

Each employee is represented by the application as an instance of the data type defined in Figure 3. The employee type has subfields of primitive or user defined types for each item of employee information. The type, and each field in it, also has attributes describing the nature of the data. On the data tag, the structure attribute indicates that the collection of fields is disjoint, i.e. is a collection of independent information about an employee. Dialog attributes such as structure, emphasis, security, and kind are part of the ITS architecture and provide information used by the style rules to select appropriate presentations for each data type.

One or more fields that share data between the application and views are grouped into tables as shown in Figure 4. The browser table contains a single field, an instance of the employee type.

In the dialog segment shown in Figure 5, the employees frame encloses two other frames. The frame called general contains a set of choices controlling the organization chart, and a list block containing the items of the chart itself. The choices allow a user to display an employee in detail, expand the organization chart to show a node's children, or quit. The detailed_view frame contains only the list of employees displayed in detail.

When a node is selected from the main chart the set_name action is executed. Set_name records the node in selected_name, used by the visit and expand actions once a command choice has been made Visit adds a record to the details list for the selected employee. Similarly, expand adds a record to the chart list for each person reporting to the selected employee. As each list is updated, the appropriate view is automatically signalled that it should redraw itself.

```
table name=browser
  ti field=person, type=employee, required=yes
etable
```

Figure 4. Table for employee browser actions

The grouping of choice and list blocks by frames is important both for the dialog and for style processing. In the dialog, frames are statically scoped so that tables created when entering a frame are visible only to actions in that frame and in its children. Frame nesting also provides information used by style rules to influence screen layout.

TOOLS FOR STYLE EXPERTS AND PROGRAMMERS

Style rules represent both graphic arts decisions, such as the choice of background color, and human factors decisions, such as a choice of an appropriate view for a given type of data. Views, once written, undergo behavioral testing to improve them and most importantly to understand the conditions under which they should be used. Those conditions become style rules to govern the appropropriate use of the view library's components.

As an example of this process, we recently built several keyboard entry techniques which differed in the amount of automatic completion of input they provided. The time to perform input tasks using the entry techniques was compared with two selection techniques using cursor keys [2]. The simple entry techniques, offering little or no automatic completion, required more time than the selection techniques

```
frame refid=employees, table=selected_name:browser,
  listname=details:browser
    the "employees" frame has nested frames for general
    and detailed employee views...
  frame refid=general, listname=chart:browser
    define the valid commands...
   choice kind=1_and_only_1
     ci message='detail', action=visit(details)
     ci message='expand', action=expand(chart)
     ci message='quit', pop
   echoice
    ...then a list showing only the name subfield for each
    person...
   list listid=chart, field=person, structure=set
     li field=name, action=set_name(selected_name)
   elist
    ...the end of frame "general."
  eframe
    The detailed list showing all subfields of person...
  frame refid=detailed_view
   list listid=details, structure=disjoint, number=2
     li field=person, area=fit
   elist
  eframe
    ...and the end of frame "employees."
eframe
```

Figure 5. Dialog tags for employee browser

which in turn were slower than the highly aided entry technique. These results might be represented in style rules by stating that (1) when no mouse is available, and (2) when the set of valid inputs is known then use aided entry rather than selection. When the set of inputs is unknown then use simple entry.

This is in sharp contrast to the typical "toolkit" approach. Applications using a toolkit typically give the application programmer (not the interface designer) control over which toolkit objects should be used. We believe, as do most developers of UIMS, that it is unlikely the application programmer should make this decision. In addition, though, we have factored the role of a single interface designer, into separate roles of style programmer and expert. The style programmer creates a library of views. The style expert writes rules which automatically generate interfaces for specific applications using those views.

One alternative to automatically generating an interface with style rules is to support the interface designer in manually designing panels by direct manipulation. Direct manipulation panel design has been most successful for simple, static, panels such as dialog boxes and menus. One system which has had success in generalizing panel specifications built by example is PERIDOT [7].

It is difficult in this approach, however, to capture the multiple rules the designer may have applied to produce a panel's final appearance. Some of these rules may have few conditions and apply to many panels. Titles, for example, should by default appear centered in their fields. Other rules may have more conditions and apply in more specific contexts. Titles being emphasized, for example, should appear in italics as well as according to attributes specified in other rules.

For the style workbench to correctly infer the generality of these rules, and to understand that italics applies only to emphasized fields, the style expert must specify the rule conditions. We believe that workbenches for creating interfaces are important but that they should focus on helping the style expert to express the conditions of these style rules, rather than on helping to design panels directly.

AN EXAMPLE STYLE

Style rules recognize features of a block of dialog tags, such as the tag name, attribute values, and data type, and match a view to the block. The choice block in Figure 5 can be drawn as a menu, cycle button, or command entry field. The application expert has used the kind attribute to indicate that the nature of the choice allows a single response. The style expert has written rules that use this information to draw each choice item (ci block) as a menu button in Figure 2. Had kind been coded to allow multiple choices, the style might use a check box instead.

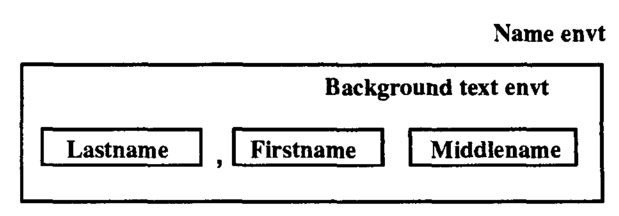


Figure 6: Environments controlled by the name view

Views are assigned to blocks indirectly through environments. An environment is a view name together with a set of style attributes appropriate to that view which control its appearance. Environments can be created both for general interface components such as titles, panels, and dialog boxes and for more specific objects such as nodes in a diagram, transistors or wires in a circuit, and names. Unlike earlier user interface mangement systems, this means there is no screen area not under control of the style mechanism in which the application can write directly. All user interaction with an application is mediated by style rules. The environments can be extended and modified over time by style experts to cover data types, devices, user groups, or other circumstances not foreseen by the developers.

Abstracting a view and its associated attributes into an environment allows a separation of content and style between the style expert and programmer similar to that between the application expert and style expert. As shown in Figure 6, a style programmer has implemented a generic name view which controls the size, position, and punctuation surrounding environments for the components of names. The data types, and views used to draw them, in the firstname, middlename, and lastname environments are hidden. Style rules are responsible for binding the particular types and field names used by the application (first, middle, and last) to the environment names used by the view.

By isolating views from knowledge about the implementation of application data types, environments allow views to be reused for many different types. An application which refines last names with the structure shown in Figure 7 can still use the name view. Rather than binding the lastname envi-

```
data type=compound
  di field=paternal, type=string
  di field=maternal, type=string
edata

data type=fullname
  di field=first, type=string
  di field=middle, type=string
  di field=last, type=compound
edata
```

Figure 7: Refined name data type

general environments for interface objects...

```
define menu, view=table, space=as_required, position=left
define text, view=string, capitals=firstonly,
  justification=left, font=timesroman, size=10
define menuitem, parent=text, xoffset=50, yoffset=50
define title, parent=text, justification=centered
define panel, view=tiledpanel
```

gallery is used for a row of objects...

```
define gallery, view=table, rows=1, space=equal,
  position=bottom
```

chart, node, and link format a group of items as a tree...

```
define chart, view=tree, direction=horizontal, hgutter=100,
  vgutter=100, space=equal
define node, view=boxednode, connections=center
define link, view=graphic, type=line, width=1,
  linestyle=solid
```

next, some environments for the employee and name types...

```
define employee, view=draw_employee
define name, view=personal_name, order=lastfirst,
  font=helvetica, capitals=firstonly
define firstname, view=string
define middlename, view=string
define lastname, view=string, size=11
```

and finally, environments for address components...

```
define address, view=street_address
define street, view=string
define city, view=string, capitals=firstonly, justify=left,
  linebreak=true
define zip, view=string, justify=left, linebreak=true
```

Figure 8. Environments used by tree and employee views

ronment to a string view, as previously, the lastname environment might be drawn by a table view. Each of the two elements in the table itself would be bound to a string view. The string attributes set in the lastname environment would be inherited by the two string views and continue to control last name formatting.

Similarly, when the tree creates links in Figure 2 it does so, not by directly drawing lines between nodes, but by creating instances of link environments with attributes for the start and end points. A variety of different views, including line or raster graphics, could be chosen by the style expert to render links. Environments give the style expert rather than the style programmer control over how the link actually appears.

To create the employee browser shown in Figure 2 , the style expert must define environments and write rules to assign them to dialog blocks. The environments required by the tree and employee views are defined in Figure 8 and the rules matching them to the example dialog are in Figure 9 .

Several of the environments in Figure 8 are common interface components. Menus are drawn using a table view, with whatever space each child requires. The style expert can force the table view to divide space equally among its children by setting the space attribute to equal rather than as_required. Titles are drawn centered and inherit their other attributes from the text environment. Panels, as implemented by the tiled panel view, arrange their children to completely fill the alloted space without overlapping. The tiled panel view looks for position attributes in each child stating preferences for the top, bottom, left, or right of the panel, but will reconcile conflicts among the children itself.

```
 1 if (TAG=FRAME) then (MATCH panel)
 2 if (TAG=CHOICE) then (MATCH menu)
 3 if (TAG=CI) then (MATCH menuitem, input=message)
 4 if (TAG=LIST) & (STRUCTURE=SET)
      & (TYPE=employee) then (MATCH chart)
 5 if (PARENTENVT=chart) then (MATCH node)
 6 if (TAG=LIST) & (STRUCTURE=DISJOINT)
      then (MATCH gallery)
 7 if (TYPE=employee) then (MATCH employee)
 8 if (TYPE=fullname) then (MATCH name)
 9 if (FIELD=first) then (MATCH firstname)
10 if (FIELD=middle) then (MATCH middlename)
11 if (FIELD=middle) & (EMPHASIS=BACKGROUND)
      then (MODIFY middlename, characters=firstonly)
12 if (FIELD=last) then (MATCH lastname)
13 if (TYPE=us_address) then (MATCH address)
```

Figure 9. Rules matching environments to dialog tags

Other environments are created when additional knowledge is required to adequately interact with application data. In the example, a tree view displays the organization chart formatted left to right, with each layer of the tree starting at the top of the frame. Links are drawn by a graphics package as a single width solid line. Names themselves have specific first, middle, and last environments so they can be separately styled and drawn in the right order. In this style, the first letter of each name is capitalized; the first and middle names follow the last with appropriate punctuation.

The rules in Figure 9 assign environments to each component of the dialog based on tag names, data types, and attributes. The employees drawn as a tree and the two drawn in detail are both coded with list tags in the dialog. The application expert, however, used the structure attribute to indicate that the second group is disjoint while the first is related as a set. The style rules can use the additional information provided by the structure attribute to render the disjoint group as a collection of unrelated views, and the ordered group as a tree. The data type (employee) is used by Rule 4 to decide that a tree view is appropriate for this particular set structured list.

Rule 5 recognizes that individual elements in the list (instances of the browser table) are children of a chart environment, and matches node environments to them. Nodes display a rectangle surrounding their contents, and define connection points into and out of them for links.

To see employee names, we must insert name environments as children of each node. Since only the name field of the browser table is coded on the first listitem (li) tag, rule 8 can

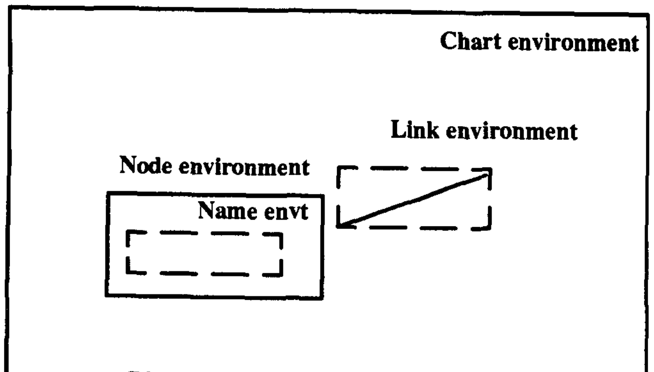


Figure 10: Environments created for nodes and links in an organization chart. Dashed boxes indicate borders of environments without visible boxes.

match directly to that tag. Rules 6 and 7 similarly match gallery environments to disjoint lists. The employee children of the gallery will draw more detailed views of the employee type as the user selects nodes of interest in the diagram. Figure 10 shows how these rules combine to create a tree of styled views for each node in the organization chart.

ITERATIVE DESIGN OF APPLICATIONS AND THEIR TOOLS

We have implemented two prototypes of these tools during the first year of the Interactive Transaction Systems, or ITS, project. One prototype is based on DOS and OS/2[TM], and the other on Berkeley Unix [TM]. Where the two prototypes differ, the Unix version has been the basis for discussion here.

A number of dialogs have been implemented in the OS/2 version, including an overview of the ITS project itself, and an interactive workbench for writing style rules. Lists, tables, pie-menus, and styled text views have been implemented in this version. The table view has been extensively reused to implement a wide variety of interface objects including menus, forms, and cycle buttons. Input techniques include variable rate scrolling, and automatic completion of text input.

The Unix version is implemented using the ANDREW toolkit [6] and X window system [9]. Views have been implemented in this version for frames, panels, menus, strings, and trees, as well as for the more specific employee, name, and address types required by the example in this paper.

These views have been sufficient to generate the interfaces to a number of demonstration applications. We plan to continue with more extensive applications developed both by our group and by external users. At first, two workbenches will be implemented to help application and style experts. The application workbench will contain syntax-directed editors for dialog tags and attributes, and a variety of editors for control flow. The style workbench will focus on creating, modifying, and understanding style rules. Of particular interest will be explanation tools to trace the execution of rules which have led to specific style decisions.

At the same time, we are testing the usability of the tools and the applications developed with them. Studies of tool usability are particularly helpful since many applications will reuse interaction techniques originally developed for tools. We look forward to describing these activities in future reports.

ACKNOWLEDGEMENTS

We thank Tom Cofino, Sharon Greene, Shirley Hsieh, Petar Makara, Uri Shani, and Ann Vosburgh for their contributions to the ITS project and comments on this paper.

REFERENCES

1. Feiner, S. An Architecture for Knowledge-Based Graphical Interfaces. In *Proceedings of AAAI/Lockheed Workshop on Intelligent Interfaces* (Mar. 29–Apr. 1). 1988.
2. Gould, J., Boies, S., Meluson, A., Rasamny, M., and Vosburgh, A.M. Empirical Evaluation of Entry and Selection Methods for Specifying Dates. *Human Factors*, in Press.
3. IBM Corporation. *Document Composition Facility: Generalized Markup Language Starter Set User's Guide*. SH20–9186–04, May, 1987.
4. Kamran, A., and Feldman, M.B. Graphics Programming Independent of Interaction Techniques and Styles. *Computer Graphics* (January, 1983), 58–66.
5. 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.
6. Morris, J., Satyanarayanan, M., Conner, M., Howard, J., Rosenthal, D., and Smith, F. Andrew: A Distributed Personal Computing Environment. *Communications of the ACM* 29, 3 (March 1986), 184–201.
7. Myers, B.M. Creating Interaction Techniques by Demonstration. *IEEE Computer Graphics and Applications* 7, 9 (Sept. 1987), 51–60.
8. Perlman, G. An Axiomatic Model of Information Presentation. In *Proceedings of the Human Factors Society–31st Annual Meeting*. pp.1229–1233. 1987.
9. Scheifler, R.W. The X Window System. *ACM Transactions on Graphics* 5, 2 (Apr. 1986), pp. 79–109.
10. Szekely, P. Separating the User Interface from the Functionality of Application Programs. Ph.D. Thesis, Carnegie Mellon University, 1988.
11. Unilogic, Ltd. *SCRIBE Document Production Software, User Manual*, June, 1985.