User Interface Management System Embedded in a Multimedia Document Editor Framework*

Takashi Ohtsu[1] and Michael A. Harrison[2]

[1] Information Systems Research Lab., Matsushita Electric Industrial Co., Ltd., 1006 Kadoma, Kadoma, Osaka 571 Japan
[2] Computer Science Division, University of California at Berkeley, 571 Evans Hall, Berkeley, CA 94720 U.S.A.

Abstract. This paper describes *Duma*: a *Data-based User interface management system for Multimedia Application*, which is embedded in a multimedia document editor framework (MMDEF). MMDEF is the core of a multimedia document editor, which can adapt to externally defined media types and operations, and enables the user to work on documents composed of multimedia objects, including objects of newly defined types, through a coherent user interface. Duma introduces an extensible data model called *interactor* that abstracts the user interaction between application semantics and user interface components. Also. Duma's *data-based UIMS architecture* embodies an interactive UI design environment in which interfaces to the interactor model are given.

1 Introduction

Remarkable advances in CPU performance and system software enable recent desktop computers to process and present non-textual objects with relatively inexpensive cost. Accordingly, in the field of document processing, a number of DTP systems have started incorporating capabilities to let non-professional users author *multimedia documents* that combine not only text objects but multiple kinds of non-textual media objects. An example of such systems is Microsoft's Word which is equipped with the 'plug-in' capability that allows externally-defined media objects and their subeditors to be hooked up to its main editor. Meanwhile, most of these DTP systems treat foreign media objects as special textual objects that inherit features of text in terms of presentation and editing. Therefore, in editing a multimedia document, users have to explicitly go back and forth between the main editor for textual media and other medium-specific editors as shown in Fig. 1(A). We observe that this type of user interfaces (UI's) will become unacceptable for users when the number of foreign media explodes, since users will have to take care of the differences of all media incorporated in documents.

* This research has been sponsored in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00039-88-C-0292, monitored by Space and Naval Warfare Systems Command, and under Grant MDA972-92-J-1028. The content of the information in this paper does not necessarily reflect the position or the policy of the U.S. Government.

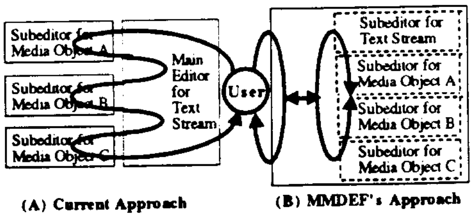

Subeditor for Media Object A
Subeditor for Media Object B
Subeditor for Media Object C
Main Editor for Text Stream
User
Subeditor for Text Stream
Subeditor for Media Object A
Subeditor for Media Object B
Subeditor for Media Object C
(A) Current Approach
(B) MMDEF's Approach

Fig. 1. Editing Environment for Multimedia Documents

To deal with this problem, we have been designing and implementing a multimedia document editor called *Ensemble*. Ensemble, whose appearance is shown in Fig. 2, is built on our research framework called MMDEF (MultiMedia Document Editor Framework). MMDEF is the core of a tightly integrated multimedia document editor that adapts to different media. Ensemble uses a novel architecture to utilize common services unlike a system like Quill [3] which is a collection of subeditors. The user of Ensemble may have the illusion that s/he is in a subeditor when editing a MODULA program. For the integrated multimedia document editor on MMDEF, all of the available media are adapted. In other words, the differences among media are taken care of by MMDEF instead of by users as shown in Fig. 1(B).

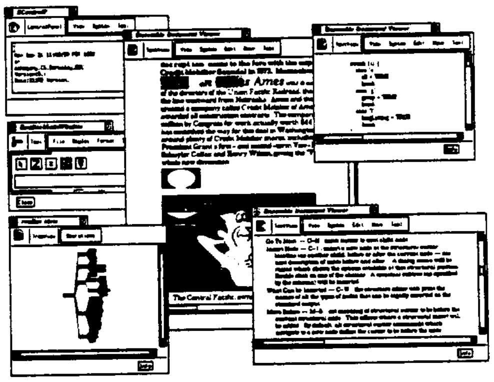


Fig. 2. A Screen Dump of Ensemble based on MMDEF

We claim that MMDEF plays the role of a user interface management system (UIMS). In addition, MMDEF should offer software repository of UI's that can be shared among adapted media at runtime. The repository might store diverse models of user interaction that can be reused rapidly by newly adapted media.

In this paper, we describe the design and the implementation of *Duma* [7]: an embedded data-based UIMS in MMDEF, that provides users, UI designers, and AP designers with coherent UI's. Duma's achievement is characterized by the following two medium-independent services. First, Duma introduces an extensible data model called *interactor* that abstracts runtime user interaction between applications (AP's) on MMDEF and UI widgets. Secondly, Duma's *data-based UIMS architecture* embodies an interactive UI design environment in which both UI and AP designers are given interfaces to the interactor model by means of a UI specification language and coherent tools provided by MMDEF. Fig. 3 shows the architecture of Ensemble consisting of MMDEF and Duma.

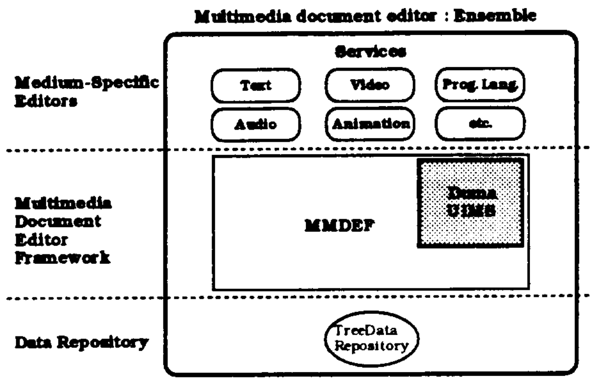


Fig. 3. Ensemble Architecture

In the following chapters, we discuss the data-based UIMS followed by the architecture of MMDEF. Then, we introduce the interactor model, the UI design layers we adopted into Duma, and the Duma prototype we have implemented. Finally, we discuss our conclusions and our plans for future work.

2 Related Work

2.1 Graphical UIMS Architecture

Since the Seeheim model [10] calls for three components in a UIMS – presentation, semantic interface, and dialog control – a number of UIMS architectures have been proposed. Recently, as window applications proliferate, the role of UI widgets in terms of graphical UIMS's is getting more important. On the other hand, we observe that currently available graphical UIMS's such as Motif, OpenLook, and ET++ [11] are still primitive and do not clarify the border between UI's and AP's.

For example, to switch a set of radio buttons into a menu, one must replace lines of code statically written in the AP and recompile it because s/he has to verify compatibilities of these UI widgets by hand. We observe that problems mentioned above arise because most traditional UIMS's have been focusing on the sequence of users' events rather than the semantics of user interaction between UI's and AP's.

2.2 Data-based UIMS Architecture

Depicting universal semantics of user interaction is unrealistic; but considering the fact that most interactions using typical UI components like buttons and panels contributes to transfer certain data objects to/from AP's/users, representing the user interaction as data types that encapsulate two concepts is realistic and effective in reducing AP/UI designers' chores. The two concepts are: the class of the data to be transferred, and the scenario that demonstrates the detailed runtime manner of user interaction. In this paper, we define a *data-based UIMS* as one that explicitly supports such abstract data types of user interaction. The design of Duma is based on this UIMS architecture.

The ITS architecture [12] is considered to be a data-based UIMS architecture. In ITS, data values to be passed among application semantics and UI's are stored in data tables. User interaction of ITS is represented by two models of communication: one between the data tables and the application semantics, and another between the tables and the UI widgets. As for the style specification, ITS adopts if-then rules. The if part specifies a pattern of attributes. The then part specifies an attribute environment in which the UI is presented. These rules are compiled statically and produce runtime objects that keep the UI consistency defined by the rules. However, neither users, nor the UI designers, nor the AP designers can interactively edit the rules at runtime.

Selectors [6] are widgets used in the ACE environment that enhances the ITS architecture. Selectors are classified according to semantics of interaction rather than their appearances. Though this idea is close to what we would like to have, we pursue the following three properties in addition to those of the Selectors: the rapid adaptability of new UI objects to the application and the UIMS, the extensibility of abstraction for runtime interaction, and the interactivity of UI design environment.

3 MMDEF Architecture

MMDEF treats each multimedia document as an object instantiated from a certain document class. A document class can be composed of multiple media while a document class itself is also considered to be one composite medium. Please notice that we use the term *medium* (or *media* in plural form) in a slightly different way from the general usage. Common examples of media, such as `Text` and `Video`, can not be decomposed, therefore, we call them *primitive media*.

The research goal of MMDEF is to clarify medium-independent services for multimedia document processing and to guarantee the adaptability of externally defined document class to its multimedia document editor. MMDEF should coordinate editing protocols and UI's of media objects of multimedia documents so that their boundaries could be virtually hidden. We claim MMDEF to be composed of several well-organized modules each of which serves as a medium-independent abstraction for a certain purpose. We call such a module a *mediator*. MMDEF consists of five mediators in its initial design: *tree data mediator*, *structure mediator*, *presentation mediator*, *editor mediator*, and *UI mediator*. Fig. 3

shows the Ensemble architecture including Duma and MMDEF. The architectural goals of Duma meet those of the UI mediator of MMDEF. In the following, we briefly discuss each mediator.

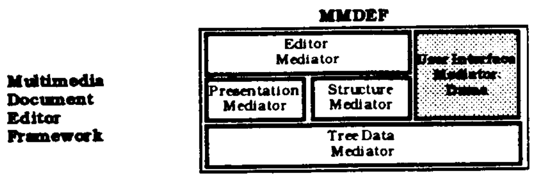


Fig. 4. MMDEF Architecture

3.1 Tree Data Mediator

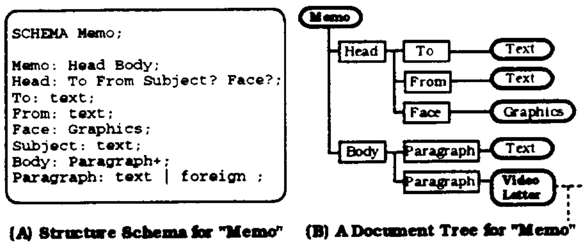
The fundamental information regarding documents in MMDEF such as their logical structure have the form of trees, which are stored in the tree data repository shown in Fig. 3. The tree data mediator provides interfaces to effective operations on those tree-formed data.

One distinguished service of the tree mediator is its *virtual tree service* which is for multiple clients who share the same logical tree data simultaneously. In our design, a virtual tree class takes an essential tree as a *base tree* and derives multiple virtual trees whose structures are, at first, the same as that of the base tree. Structural modifications of the base tree are reported to each virtual tree immediately. On the other hand, modifications to virtual trees are not propagated to the base tree and other derived trees. This mechanism allows clients to annotate the same tree as they please without affecting the underlying data structure. Clients will use virtual trees for their own transitory data representations. To maintain the integrity of such tree-structured data, the tree data mediator also provides transaction services in terms of a database management system to ensure atomic editing operations on trees.

3.2 Structure Mediator

The structure mediator maintains the structural consistency of documents. In other words, according to requests to modify the structures of documents, the structure mediator examines whether the requests are acceptable or not by applying a set of rules to the logical structure of the document. The set of rules is called a *structure schema*, which is given to each document class and is stored as an external ASCII file. An example of the structure schema for the memo document class is shown in Fig. 5(A).

The logical structure tree of a document, which is maintained by a certain structure schema, is called a *document tree*. An example of the document tree which is derived from the **memo** structure schema is shown in Fig. 5(B).



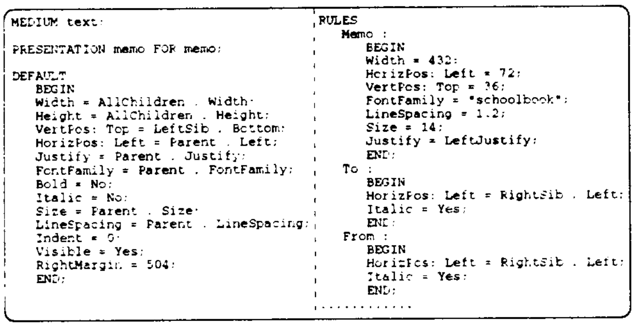
```
SCHEMA Memo;

Memo: Head Body;
Head: To From Subject? Face?;
To: text;
From: text;
Face: Graphics;
Subject: text;
Body: Paragraph+;
Paragraph: text | foreign ;
```

Fig. 5. Structure Schema and Document Tree for **Memo**

3.3 Presentation Mediator

The presentation mediator supports the management of multi-media document presentation [4]. In the first place, the presentation mediator accepts a document tree, maps a set of rules to it, and derives a *presentation tree* that includes the structural information regarding its presentation. Finally, the presentation tree is formatted to various kinds of physical devices such as printers for page-oriented documents, bitmap displays for graphics, speakers for audible objects, and so on.

The set of rules called *presentation schema* are stored as an external ASCII file. Each document class may have more than one presentation schemas, which implies that the documents of the class might have several presentations simultaneously. An example of the presentation schema for memo document class is shown in Fig. 6.



```
MEDIUM text;

PRESENTATION memo FOR memo;

DEFAULT
   BEGIN
   Width = AllChildren . Width;
   Height = AllChildren . Height;
   VertPos: Top = LeftSib . Bottom;
   HorizPos: Left = Parent . Left;
   Justify = Parent . Justify;
   FontFamily = Parent . FontFamily;
   Bold = No;
   Italic = No;
   Size = Parent . Size;
   LineSpacing = Parent . LineSpacing;
   Indent = 0;
   Visible = Yes;
   RightMargin = 504;
   END;
```

```
RULES
   Memo :
      BEGIN
      Width = 432;
      HorizPos: Left = 72;
      VertPos: Top = 36;
      FontFamily = "schoolbook";
      LineSpacing = 1.2;
      Size = 14;
      Justify = LeftJustify;
      END;
   To :
      BEGIN
      HorizPos: Left = RightSib . Left;
      Italic = Yes;
      END;
   From :
      BEGIN
      HorizPos: Left = RightSib . Left;
      Italic = Yes;
      END;
..............
```

Fig. 6. Presentation Schema for **Memo**

3.4 Editor Mediator

The editor mediator receives editing requests on multimedia documents and interprets them as a set of procedural commands on involving different media. Then, it invokes appropriate services with those commands.

For example, with an editing request on objects presented in a view, the editor mediator functions as follows. First, the editor mediator specifies objects on documents to which the request is issued by both the locational information from the presentation mediator and the structural information from the structure mediator. Then, it finds services corresponding to the objects. Finally, the editor mediator interprets the request as a sequence of commands and dispatches them. The editor mediator should also take care of multiple users' requests on the same logical document, since one logical document may be viewed with different presentations simultaneously.

3.5 UI Mediator: Goals of Duma

The UI mediator offers a medium-independent UI model to AP's such as services and viewers, and embodies a UI development environment where highly interactive design and customization are possible. Our goal here is to make UI's adaptable to MMDEF as well as externally defined document classes so that newly adapted document classes can reuse and share UI's. This adaptability must take two forms. First, it must be relatively easy to define new UI's. Secondly, it must be possible to register these new UI's with MMDEF without having to directly modify the code. To explore the adaptability of UI's, the medium-independent UI model of the UI mediator utilizes the model of user interaction which was introduced in the data-based UIMS architecture. We call the model that we discuss in detail in the next section the *interactor model.*

At runtime where particular interaction is requested, the UI mediator deploys a UI agent who binds the AP to UI widgets by examining the data types of the user interaction. Then, AP and UI widgets are allowed to communicate with each other through the agent. The characteristics of the agent include understanding which class of values should be transferred and which UI widgets are valid to effect the user interaction in exchanging the values of the class.

The ultimate goal of the UI mediator is to provide MMDEF with the user interaction models and UI widgets as one media type. By doing so, the UI mediator can utilize the full set of services given by MMDEF to edit, author, and customize UI specifications. We expect that the UI mediator will make it possible to embed UI components in documents as well as other media objects. We are looking at the Embedded Buttons architecture [2] as the first step. In the long run, the UI guideline itself will be produced as one multimedia document on MMDEF.

4 Interactor Model

The interactor is the fundamental model that the Duma adopts in order to combine AP's with UI's under the data-based UIMS architecture. As introduced in section 2.2, we represent user interaction as the transfer of primitive/composite objects and the scenarios that describe the detailed manner of interaction. For example, suppose we have an interaction 'get one full-name of an employee of the company'. In this case, an employee object, whose name attribute is **Takashi_Ohtsu**, will be transferred and its scenario might be 'to select one highlighted name from the employees listed sequentially in a scrollable window pane widget'.

Therefore, the interactor model also represents both the scenario and the class of media objects to be transferred in a common model to all the media supported in MMDEF. An interactor consists of the following three objects: *virtual interactor*(VI) that provides interfaces to the interactor with AP's, *interactor widget*(IW) that provides interfaces to the interactor with UI's, and *real interactor*(RI) that connects a virtual interactor and an interactor widget. An interactor is represented by one of the following tuples:

```
1) (a VI, a IW, a RI)
2) (a VI, a RI)
3) (a RI, a IW)
```

Most interactors are represented by `1)`. Interactors defined of the form of `2)` and `3)` are called *application-defined interactor* and *user-defined interactor* respectively. AP's can not access the user-defined interactor, while UI's can not access the application-defined interactor.

In the following, we discuss each object in detail.

4.1 Virtual Interactor

The virtual interactor is an object that manages data to be passed to/from users and represents the essential interaction method which is completely independent of detailed behaviors and appearances of UI's. In Duma, AP's communications with UI widgets are substituted by communications with virtual interactors.

Each virtual interactor is typed by an *interactor type*. The interactor type is represented by a decision tree that consists of classes of media objects to be transferred in interaction and partial interaction scenarios which are independent of the appearences of UI widgets. Schematic examples of the interactor types are shown in Fig. 7.

In Fig. 7, each leaf of an interactor type is a *basetype* that represents a class name of the media object adapted to MMDEF. We assume that the basetypes have a class hierarchy. In Fig. 7(A), a basetype **FONTNAME** is shown, which may have values of **Helvetica**, **Courier**, or **Schoolbook**. Meanwhile, internal nodes of an interactor type are instances of the **CHOICE** function, which takes a single argument k representing the number of choices that can be made simultaneously. The value of k may be a positive number or either of the two special values, **ANY** and **EVERY**. The semantics of the **CHOICE** function make it pointless for an internal node to have a single internal node child. The semantics of the **CHOICE** function depend on the type(s) the of node's child(ren). Namely, if the **CHOICE** function has a single base child, k is the maximum number of basetype values that may be selected, while if it has multiple internal children, k is the maximum number of subtrees whose decision points can be selected. A value of **ANY** indicates that any number of subtrees are selectable while a value of **EVERY** indicates that a value should be obtained for every subtree.

In Fig. 7(A), k is equal to one. Accordingly, the interactor type shown in Fig. 7(A) has the semantics of choosing one of the known fontnames. Meanwhile, in order to get a font size from the user, s/he will define an interactor type **GETFONTSIZE** as shown in Fig. 7(B). Fig. 7(C) shows the font face style selection that we often see in commercial products such as Microsoft's Word. The value of **OTHERTYPEFACE** is either **Bold**, **Italic**, **Underline**, **Shadow**, or **Outline**, while the **NORMALTYPEFACE** takes the value **Normal** only.

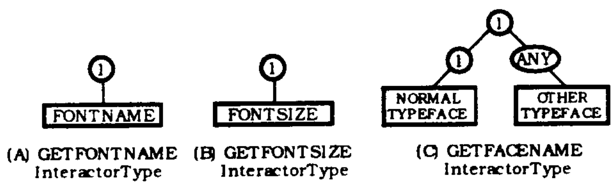

1
FONTNAME
(A) GETFONTNAME InteractorType
1
FONTSIZE
(B) GETFONTSIZE InteractorType
1
1
ANY
NORMAL TYPEFACE
OTHER TYPEFACE
(C) GETFACENAME InteractorType

Fig. 7. Interactor Types

4.2 Interactor Widget

The interactor widget is an object that embodies one well-organized widget set composed of more than one primitive widget component such as buttons and panes, which behaves as one integrated UI component. Selectors widgets are thought to be categorized in these interactor widgets. We show the appearances of interactor widgets in Fig. 8. Like the virtual interactor, the interactor widget is given a type by the interactor type except that the interactor type for the interactor widget has only two nodes, a root and a leaf, where the **CHOICE(x)** function represents how many media object values of the basetype could be selectable at most by the user, where **x** could be a natural number or **ANY**. The basetype of the leaf is the type of the media object that the interactor widget can manipulate. In our approach, interactor widgets are UI guideline independent, namely, file selection boxes of Motif and OpenLook are treated equally regardless of their different policies.

In Fig. 8, each interactor widget has a **CHOICE(1)** function where (A) has the basetype **FONTNAME**, (B) has the basetype **IMAGE**, and (C) has the basetype **STRING**. A simple string label and an icon are considered to have **CHOICE(0)** function.

4.3 Real Interactor

The real interactor mediates between a virtual interactor and an interactor widget. It assigns a virtual interactor to an interactor widget once it verifies the compatibility of their interactor types at runtime. The algorithm applied to this verification is shown in Fig. 9. It first checks whether all the basetypes of the

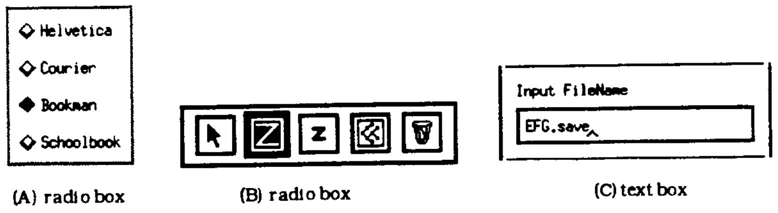


Fig. 8. Interactor Widgets

interactor type are "compatible" with the basetype of the interactor widget. We define a type **A** to be *compatible* with a type **B** if **A** is defined as a subclass of **B**.

Since a font name is expressible by a string, the basetype **FILENAME** is defined as a subclass of **STRING**. Therefore, according to the algorithm, the interactor type **GETFONTNAME** in Fig. 7(A) not only matches the sophisticated file selection box, but also matches a simple interactor widget as we show in Fig. 8(C). Cf. [5] for a more elaborate design of a file browser for Ensemble.

```
CheckTypesOf(ITV:InteractorType of a VirtualInteractor,
             ITW:InteractorType of an InteractorWidget){
  // check all base types
  for (all leaves of ITV){
    if(basetype of leaf is not compatible the base
      type of ITW) return ERROR;
  }
  // check CHOICE types
  for (traverse ITV){
    if(current node is CHOICE(x) node){
      if(children are base types)
         currentnode.maxchoice = x;
      if(children are CHOICE nodes)
         currentnode.maxchoice =
          maximum selectable items at this level
          considering x and the maxchoice of each child;
      }
  }
  Node node = rootNode of ITV;
  if(node.maxchoice <= y ) // where ITW  has a type
                           // of CHOICE(y)
    return AGREE;
  else
    return ERROR;
}
```

Fig. 9. Compatibility Checking Algorithm

5 Data-based UIMS Architecture in Duma

Duma's data-based UIMS architecture provides a UI design environment for window-oriented AP's such as services and viewers on MMDEF according to the interactor model described in the previous section. Duma divided the UI design process into the following five layers: *medium action*, *dialog*, *style composition*, *layout composition*, and *command dispatcher* as shown in Fig. 10.

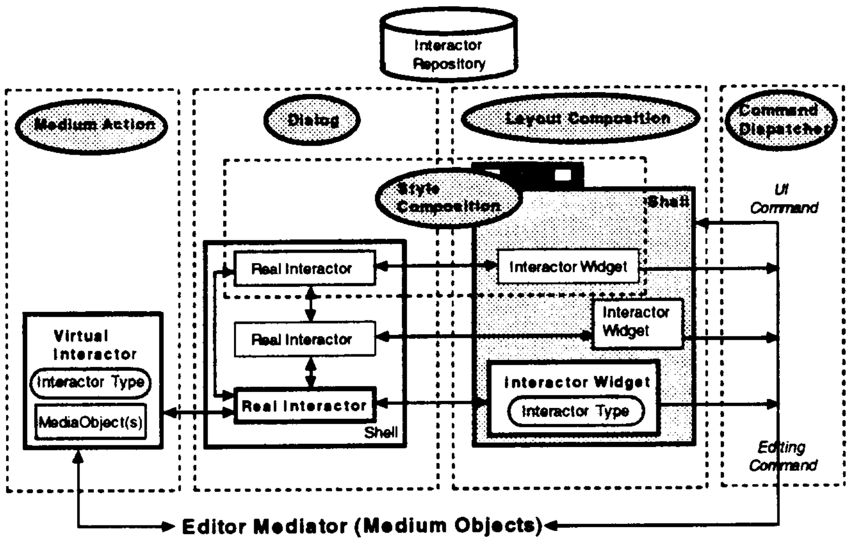


Fig. 10. UIMS Architecture in Duma

Medium Action Layer The medium action layer provides interfaces to design interactor types by means of basetypes adapted to MMDEF. Please note that the design of interactor types is a process independent from the AP semantics and interactor widgets. At runtime, AP's instantiate virtual interactors according to the interactor type defined in this layer. For now, as soon as a virtual interactor is instantiated, a real interactor is also instantiated and bound to it. Then, the real interactor waits for requests to bind interactor widgets.

Dialog Layer The dialog layer provides interfaces to group semantically related interactors into one object called a *shell*. The shell is considered a container object whose constituent elements are interactors. It gives one scope in the sense of programming language to those interactors. The shell is one of interactors and is assigned to an interactor widget whose appearance is an window. An interactor may belong to more than one shell. The constraints among interactors are described as behaviors of shell objects.

For example, suppose we design a dialog that prompts for several font properties such as name, size, and face style at a time. The application designer might define a shell object called **FontSelectDialog** with real interactors whose interactor types are **GETFONTNAME**, **GETFONTSIZE**, and **GETFACENAME**.

Style Composition Layer The style composition layer provides interfaces to assign interactor widgets to interactors according to the compatibility checking described in section 4.3. Since this assignment is performed at runtime, UI designers can easily prototype interactor widgets on the fly. Also, UI designers may search for interactor widgets compatible with the interactors from the tree data repository.

Layout Composition Layer The layout composition layer provides runtime formatters for interactor widgets according to a certain layout policy that the user can specify. The examples of the layout policies are the simple hierarchical model, the constraint based model, and the $\TeX$'s box and glue model. The fundamental services of the formatters should be similar to that of the presentation mediator of MMDEF. For now, the layout composition layer has its own formatters.

UI designers may append user-defined interactor widgets to the shell and lay them out as well as the usual interactor widgets. These widgets are the user-defined interactor widgets, which are, in fact, not bound to any AP's semantics. However, they can work as assistants that guide the user's operations by offering informative messages.

As an example of layout results, the appearance of a shell window derived from the shell object **FontSelectDialog** we mentioned earlier is shown in Fig. 11. The layout policy we adopted here is a simplified box and glue model.

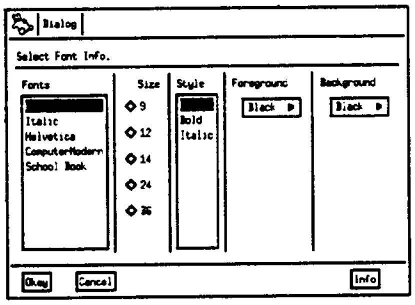


Fig. 11. The Appearance of a Shell **FontSelectDialog**

Command Dispatcher Layer The command dispatcher layer provides interfaces to assign commands to command-dispatchable interactor widgets such as menus, buttons, and keys. The commands are acceptable either by Duma or system services adapted to MMDEF. Duma has a built-in basetype called **COMMAND** for the interactors that dispatch commands. In other words, menus and buttons are considered to have interactor types whose basetypes are **COMMANDS**, and they dispatch commands by users' operations at runtime to the editor mediator of

MMDEF. Commands that are acceptable by the editor are called *editing commands* and commands acceptable by Duma's objects are called *UI commands*. Commands that open/close windows are examples of UI commands.

6 Duma Prototype

The current Duma prototype is implemented as a module of Ensemble, which is written in C++ using the GNU C++ library and the OSF/Motif X11 toolkit on Sun SparcStations. At the time of this writing, our group is near completion of the interactor model and the data-based UIMS architecture with a limited number of primitive media provided by MMDEF.

Interactive UI specification in the Duma prototype is done through a human-readable declarative language, or *USPEC*. USPEC specification can be loaded at any time while the MMDEF is running, and its modifications are promptly reflected in the UI design of the AP's. For now, USPEC specifications are edited through a text editor of Ensemble.

All the objects of interactors designed by USPEC have their own identifiers. The identifiers are unique. At runtime, AP's and UI's can access these objects by querying the built-in **INTERACTOR** class with identifiers.

In the following, we introduce several UI specification examples that have been implemented.

Virtual/Real Interactor Specification In the current implementation, interactor type specification is done by naive C++ description. For example, the interactor type **GETFONTNAME** and **GETFONTSIZE** we mentioned earlier are defined with the fragment shown in Fig. 12 where **IDT_ONE**, **IBT_FONTNAME**, and **IBT_FONTSIZE** represent **CHOICE(1)**, a basetype **FONTNAME** and a basetype **FONTSIZE**, respectively. Then, using these specified interactor types, interactors are generated as shown in Fig. 13. In the third line of the Figure, an interactor **size** whose interactor type is **GETFONTSIZE** is generated and stored in the tree data repository.

```
new InteractorType("GETFONTNAME", IDT_ONE, IBT_FONTNAME);
new InteractorType("GETFONTSIZE", IDT_ONE, IBT_FONTSIZE);
```

Fig. 12. Interactor Type Specification

```
1     registerInteractor("body", "LABEL");
2     registerInteractor("fonts", "GETFONTNAME");
3     registerInteractor("size", "GETFONTSIZE");
4     registerInteractor("style", "GETFACENAME");
5     registerInteractor("bgcolor", "GETCOLOR");
6     registerInteractor("fgcolor", "GETCOLOR");
```

Fig. 13. Virtual/Real Interactor Specification

Shell and Interactor Widget Specification In Fig. 14, we show the USPEC specification of the shell object **`FontSelectDialog`** that consists of the six interactors previously specified in Fig. 13. At the same time, the Figure shows which interactor widget is assigned to each interactor. For example, in the third line of Fig. 14, the real interactor ***body*** is assigned the interactor widget of the class **`SIMPLE_LABEL`**, which is a label widget whose the interactor type is **`CHOICE(0)`** of **`STRING`**. The rest of interactor widgets used by this dialog have class names beginning with **`ONE_OF_M`** indicating the type **`CHOICE(1)`** of **`STRING`**. The compatibility checking is done when this specification is loaded. If the checking fails, an appropriate interactor widget is chosen as default from the tree data repository.

```
1      DIALOGSET FontSelectDialog
2      BEGIN
3          PARTS = (%body,    SIMPLE_LABEL),  // label
4                  (%fonts,   ONE_OF_M_LIST), // list selection box
5                  (%size,    ONE_OF_M_RBTN), // radio button
6                  (%style,   ONE_OF_M_LIST), // list selection box
7                  (%fgcolor,ONE_OF_M_PBTN),  // push button
8                  (%bgcolor,ONE_OF_M_PBTN);  // push button
9      END
```

Fig. 14. Interactor Widget Style Specification for a Shell **`FontSelectDialog`**

Layout Specification Currently, layout specification in Duma is based on a tree structured presentation whose internal nodes have one of two types, *VPack* or *HPack*. A VPack node implicitly specifies that its children should be packed vertically. An HPack node implies horizontal packing.

Fig. 15 shows the part of the USPEC that defines the tree structure of the shell window **`FontSelectDialog`**. This specification defines a presentation tree using a function-call-like notation.

```
DIALOGSET FontSelectDialog
BEGIN
  STYLE = VPack(%body,Sep(),
            HPack(VPack(SLabel("Font"), %fonts),Sep(),
                  VPack(SLabel("Size"),%size), Sep(),
                  VPack(SLabel("Style"),%style),Sep(),
                  VPack(SLabel("Foreground"),%fgcolor),Sep(),
                  VPack(SLabel("Background"),%bgcolor)));
END
```

Fig. 15. Layout Specification

Command Dispatcher Specification The specifications in the command dispatcher layer are for keyboard commands and pulldown/popup menus, which are written in USPEC. Fig. 16 shows an example for popup-menu and menubar specification where Fig. 17 shows the generated hierarchy of the menubar from Fig. 16.

```
MENUSET Default BEGIN
  MENUBAR   (View, Edit(Insert, Delete));
  POPUPMENU (Insert, Delete);
  DEFINE View BEGIN
    LABEL="View"; MEDIUM= TextEd;
    ITEMS=("Open","editor SuperEditor OpenDocument"),
          ("Save","editor SuperEditor SaveDocument"),
          ("Export","editor TextEd ExportDocText"); END
  DEFINE Insert BEGIN
    LABEL="Insert"; MEDIUM= TextEd;
    ITEMS=("Node","editor StructEditor InsertNode"),
          ("Query","editor StructEditor QueryInsert");END
  DEFINE Delete BEGIN
    LABEL="Delete"; MEDIUM= TextEd;
    ITEMS=("Atom","cursor DeleteAtom"),
          ("AtomBack","cursor DeleteAtomBack"),
          ("Word","cursor DeleteWord");          END
  DEFINE Edit BEGIN
    LABEL="Edit"; MEDIUM= ;
    ITEMS=("Redraw","editor StructEditor Repaint"); END
END
```

Fig. 16. Command Dispatcher Specification for Menus

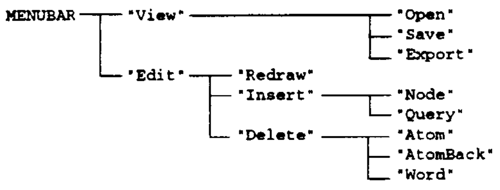


Fig. 17. Complete Menu Structure for a Menubar

7 Conclusion and Future Work

We have designed and implemented Duma as the UI mediator of the integrated multimedia document editor, Ensemble, on MMDEF. Duma adopts the interactor model and the data-based UIMS architecture to realize a highly extensible UI design environment. Though the number of available media given by MMDEF is currently limited, all windows of Ensemble including the control panel, the document viewers, and the dialog panels shown in Fig. 2 are built by Duma. Most of the UI features are customizable by Duma through the USPEC description, while the detailed settings such as what X resources do are not fully supported. Currently, Ensemble on MMDEF exceeds 90,000 lines of code of which the UI mediator occupies about 15%.

Our future work is to eliminate all C++-style UI specifications and implement direct manipulative editors for UI design. Furthermore, Duma should have multi-view interactive editors for UI design. The two-view approach is proposed by FormsVBT [1] and we extend their idea by incorporating MMDEF's multi-presentation services. At this point, the tree-structured data viewer *realize* [9] is attached to MMDEF and gives fundamental interactive features.

We also plan to enrich interactor types and interactor widgets so that the interactor object model could support a wider range of media objects. The very first step is to review the decision tree representation of the interactor type. We currently consider attributes to be annotated on the CHOICE function. Moreover, we consider the runtime command interpreter on the interactor object model. This enriches the semantics of the events to be passed to interactors. For that reason, we are planning to incorporate the Tcl/Tk [8] architecture with the interactor model soon.

References

1. Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. A two-view approach to constructing user interfaces. *Computer Graphics*, 23(3):137–146, July 1989.
2. Eric A. Bier. EmbeddedButtons: Documents as user interfaces. In *proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 45-53, November 1991.
3. Donald D. Chamberlin, Helmut F. Hasselmeier, Allen W. Luniewski, Dieter P. Paris, Bradford W. Wade, and Mitch L. Zolliker. Quill: An extensible system for editing documents of mixed type. In *Proc. of the 21st Hawaii International Conference on System Sciences*, pages 317-326, Kailua-Kona, Hawaii, Jan 5-8 1988.
4. Susan L. Graham, Michael A. Harrison, and Ethan V. Munson. The Proteus presentation system. In *ACM SIGSOFT symposium on software development environments*, pages 130–138, Tyson's Corner, VA, 1992.
5. Michael A. Harrison and Thomas A. Phelps. The next best thing in file browsers. In *Proceedings of the TCL/Tk Workshop*, pages 110–112, Berkeley. CA, June 1993. Computer Science Division, University of California at Berkeley.
6. Jeff Johnson. Selectors: Going beyond user-interface widgets. In *proceedings of CHI '92 Conference: Human Factors in Computing Systems*, pages 273–279, NY, May 1992.
7. Takashi Ohtsu. Duma: a data-based user interface management system for multimedia application. Master's thesis, Computer Science Division, University of California, Berkeley, Berkeley, CA 94720, October 1992.
8. John K. Ousterhout. An X11 toolkit based on the tcl language. In *USENIX Summer Conference Proceedings*, pages 105-115, 1991.
9. John L. Pasalis. Realize: An interactive graphical data structure presentation and rendering system. Master's thesis, Computer Science Division, University of California, Berkeley, Berkeley, CA 94720, 1992.
10. Gunther E. Pfaff, editor. *User Interface Management Systems*. Spring-Verlag, Berlin, 1985.
11. Andre Weinand, Erich Gamma, and Rudolf Marty. ET++ - an object-oriented application framework in C++. In *OOPSLA 1988 Proceedings*, pages 46–57, September 1988.
12. Charles Wiecha, William Bennett, Stephen Boies, John Gould, and Sharon Greene. ITS: A tool for rapidly developing interactive applications. *ACM Transactions on Information Systems*, 8(3):204–236, July 1990.

This article was processed using the LaTeX macro package with LLNCS style