Man-Machine Interaction in Multimedia Document Retrieval:
The MULTOS System

P. Constantopoulos, D. Nastos, M. Theodoridou, Y. Yeorgaroudakis
P. Savino, G.L. Pancaccini

March 1990

FORTH/CSI/TR/1990/023
Technical Report Series
Computer Science Institute, FORTH †
P.O.Box 1385, Heraklio, Crete, 711-10 Greece
Tel.: +30.81.229302, Fax: +30.81.229342, Telex: 262389 CCI GR
E-mail: panos@csi.forth.gr, maria@csi.forth.gr

† "Foundation of Research and Technology – Hellas," formerly "Research Center of Crete" (RCC).
Man-Machine Interaction in Multimedia Document Retrieval: The MULTOS System

P. Constantopoulos, D. Nastos, M. Theodoridou, Y. Yeorgaroudakis

Institute of Computer Science
Foundation of Research and Technology - Hellas

P. Savino, G.L. Pancaccini

Olivetti DOR

ABSTRACT

Document retrieval in a distributed multimedia document filing system is described. Multimedia documents contain formatted data, text, image, graphics and voice. Retrieval is based on textual and pictorial content as well as on formatted data, and is performed iteratively through a uniform, flexible interface offering document set management, structure and document browsers and dynamic query formulation. User interface design and implementation are discussed.

I. Introduction

An increasing effort has been witnessed in developing systems for managing multimedia information. It is expected that in the near future huge amounts of largely multimedia information, currently in the form of letters, journals, periodicals, books, designs, hospital patient records, product catalogues, etc., will be stored in high capacity magnetic and optical disks, while the production, storage, retrieval, presentation, correlation and distribution of such information will be effected by computer [18]. A considerable body of research on the subject as well as implementations of the above functions already exist; see, for instance, [30, 44, 32, 3, 43, 18, 19, 45, 7, 33, 9, 22, 24, 38].

This work has been supported in part by the Commission of the European Communities through ESPRIT project MULTOS (no.28). Partners in project MULTOS are: Battelle (FRG), Epsilon (GR), ERIA (S), ICS-FORTH (GR), IEE-CNR (I), Olivetti-DOR (I) and Triumph-Adler (FRG). Authors' present addresses: P. Constantopoulos (panos@csi.forth.gr), D. Nastos (nastos@csi.forth.gr) and M. Theodoridou (maria@csi.forth.gr); Institute of Computer Science, Foundation of Research and Technology - Hellas, P.O.Box 1385, GR-711 10 Heraklion, Greece; Y. Yeorgaroudakis (hy4all@shapealyy); Shape Technical Center, P.O.Box 174, 2501CD The Hague, The Netherlands; P. Savino (12unix!iconet!psatit!savino) and G. L. Pancaccini (12unix!iconet!psatit!panaccini); Olivetti DOR, via Palestro 30, 56100 Pisa, Italy.
A multimedia document is a collection of formatted (attribute) data, text, images, graphics and audio segments. Its size may vary from a few kilobytes (e.g. mail messages, simple forms) to several megabytes (e.g. documents with large image and voice parts). Similarly, its structure may vary from elementary (e.g. a simple message) to very complex (e.g. a newspaper issue). Future multimedia documents will mostly be electronic entities occasionally appearing in printed form. Consequently, the ultimate goal in multimedia information systems development is to provide integrated information management including editing, presentation, storage, retrieval, input and exchange of multimedia documents. The attainment of this goal requires highly modular and extensible systems and standardization of information exchange. Some results in these directions are documented in the above citations and in [4], yet apparently more work is needed.

Work on multimedia filing systems, in particular, has so far mainly addressed issues of data modeling (e.g. [18, 45, 7, 26, 9]), system architecture (e.g. [18, 33, 9]), high-volume storage (e.g. [15, 17, 21]), access methods and query processing (e.g. [16, 29, 9, 25]), and document presentation, formation and information extraction (e.g. [42, 18, 17]). The design of query interfaces for retrieving multimedia documents, though, has not kept pace with these developments [24].

This paper presents the design and implementation of an integrated multimedia document retrieval tool developed within the ESPRIT project MULTOS (MUltimedia Office Server). This tool supports the interactive retrieval of multimedia documents on the basis of attribute data, textual content and pictorial content. Multimedia documents can have very complex structure and varied content, so retrieving them typically is a relatively unstructured, iterative process involving uncertainty on the user's part, use of inexact matching techniques and large volumes of data transmitted to and presented at workstations. The MULTOS retrieval tool aims at ensuring both effectiveness and efficiency in the retrieval process. Effectiveness is achieved by providing large flexibility in the evolution of the process and supporting the resolution of uncertainty. The user interface design, on the other hand, caters for efficiency. The tool is designed to support both the unsophisticated and the expert computer users. Another contribution to efficiency comes from limiting unnecessary transfers of large volumes of data.

MULTOS is a prototype filing system for multimedia documents with an open, distributed, client-server architecture, supporting retrieval on the basis of the structure and content of documents, and using both magnetic and optical storage media. It has its own document model and query language, provides automatic document classification, supports the exchange of documents with other systems in the format recommended by the Office Document Architecture (ODA) standard, and is connected to a multimedia editor. The MULTOS servers undertake the data management functions, while the clients support the man-machine interaction and initiate server actions. The retrieval tool is part of the client which has an object-oriented design and has been implemented using the ANDREW Toolkit and the C programming language.

In section 2 of this paper an overview of the MULTOS system is given. In sections 3 and 4 the architecture of the client and the general user interface design are presented respectively. Section 5 focuses on the function of document retrieval and the design and operation of the document retrieval tool. The implementation of the MULTOS client and the retrieval tool in particular, is described in section 6. The closing
section 7 contains a discussion of this and other related work. The access methods used in query processing are described in the appendix.

2. OVERVIEW OF MULTOS
We now briefly describe the MULTOS multimedia office filing system, specifically the requirements which it seeks to meet, the architecture of the system and of servers in particular, and the document model. Client processes are described in more detail in the next section.

2.1. Requirements
There is a number of special requirements of office filing systems which has borne a strong influence on the design of MULTOS [10]. These are:
Support of the document life cycle. The life cycle of an office document can be roughly divided into three periods. Initially, a document is private to a certain person, filed manually in that person's storage space according to some private organization scheme. During this stage the document usually undergoes several changes. In the second stage, the document is made available to some authorized community and is subject to change by any member of that community. Finally, the document enters a stable, archival state where it may only be inspected but not modified; new versions, however, may be created. The three stages of document filing are called dynamic, current and archival respectively, and have different requirements indicated in Table I.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Dynamic</th>
<th>Current</th>
<th>Archival</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>1-100Mb</td>
<td>100Mb-2Gb</td>
<td>1-200Gb</td>
</tr>
<tr>
<td>Media</td>
<td>Magnetic</td>
<td>Magnetic</td>
<td>Magnetic/optical</td>
</tr>
<tr>
<td>Response</td>
<td>&quot;1-5 sec&quot;</td>
<td>2-20sec</td>
<td>5-200sec</td>
</tr>
<tr>
<td>Queries</td>
<td>temporary name</td>
<td>content/collection</td>
<td>content/collection</td>
</tr>
<tr>
<td>Updates</td>
<td>no restrictions</td>
<td>restricted</td>
<td>restricted</td>
</tr>
</tbody>
</table>

Table I: Requirements of document filing stages

Multimodal retrieval. Classification techniques used in traditional information retrieval systems ensure retrieval efficiency, yet they are often inadequate for office documents. So, retrieval on the basis of arbitrary conditions on document content, in addition to classification attributes, should be supported. Such conditions may concern the text, image or voice parts of documents and, in fact, MULTOS addresses textual and pictorial content queries.

Flexible architecture. The perceived need for office information systems to serve entire organizations (rather than specialized groups), growing and changing gracefully along with the needs of the organizations, suggests that the architecture of an office system should allow easy expansion, simple redistribution of resources and well-defined, publicly documented interfaces.
Standard document format. The exchange of documents with other systems (e.g., via electronic mail) is enabled, and the integration with such components as document editors and input/output devices is simplified by representing documents in a standard format.

2.2. General architecture

The architecture of MULTOS is based on the server-client paradigm [41] which leads to an open system that meets the above requirement of flexibility. The system consists of autonomous, passive server processes and independent client processes which activate the servers (Figure 1). Servers perform all the data management operations and handle the storage devices. Clients mainly support the interaction with the users, issue requests to the servers and display results.

It is in distributed systems that the rationale behind the server-client paradigm becomes apparent. Here servers usually reside on specialized machines while clients reside on single user workstations and issue requests to the servers over the network. Thus servers and clients correspond to distinct network nodes. However, it is possible to run the server and client software on a single shared system. So in Figure 1, boxes can be viewed as processes rather than physical entities, and the network as an interprocess communication channel.

The functions of archival and current document filing are provided by the servers while dynamic filing is provided as a local function by the clients.

Servers are autonomous in the sense that they do not communicate with and thus ignore the existence of each other; and passive in that all operations are initiated by clients. The storage managed by a particular server is located at a single network node and does not overlap with that of another server.

2.3. Server processes

The general structure of server processes, shown in Figure 2, is here very briefly described. For detailed descriptions see [9,5]. The internal composition of server components, in particular the storage subsystem, may vary from one server to the other, yet these variations are concealed under the server interface offered over the network.

The Server Status Broadcaster periodically broadcasts over the network information about the server, which may be of use to the clients.

The Dialogue Manager mediates all server-client communication. The structure of documents, essential for the exchange of documents between server and client through the dialogue manager, is represented by the document model (see below).

The Server Controller controls the execution of server operations, document management functions and system administration functions.

The Type Handler stores information about document types in the type catalogue and allows document access through classification attributes.

Similarly, the Collection Handler manages document collections and supports querying within a collection scope.

The Structure Translator maps requests issued by the Server Controller onto operations of the storage subsystem.
The Document Storage Subsystem provides storage containers for document content and for document structures and maintains the access structures for document retrieval. The Document Storage Subsystem has been built as an ad hoc file system on top of the UNIX file system and incorporates an indexing package for fixed-format data and specially built data managers for textual and pictorial data.

Finally, the Multi-Storage File System comprises two separate file systems, one for magnetic disks and one for optical disks, and supports the Document Storage Subsystem.

2.4. Document model

A multimedia document is a complex collection of components of possibly different structure. For example, the body of a paper is structured in sections and paragraphs consisting of text and containing embedded images and formatted data. The design of a suitable data model is an important consideration. The MULTOS document model, presented in [9], is here briefly reviewed.

Three kinds of structure are distinguished in the MULTOS document model: layout, logical and conceptual. The layout and logical structures are adopted from the Office Document Architecture (ODA) standard and are used to convey information about document presentation [2,4]. The logical structure determines how syntactic document components, such as sections and paragraphs, are related to document content portions, such
as text or images. The layout structure determines how layout components, such as pages and blocks, are related to document content portions. These two structures are related through layout directives.

The conceptual structure, on the other hand, provides an organization of the information space which is useful for operations such as query specification and document creation, as it captures information which is meaningful to the user. A portion of the document with some specific semantic content, e.g. the title, authors and abstract in a paper, or the date, receiver name and address in a letter, is designated as a conceptual component. The structure of a document in terms of its conceptual components is called the conceptual structure.
Due to the lack of uniformity even among documents of the same "type", a low ratio of document instances per type is expected in a potential document classification schema. Thus, weak document types are introduced in the conceptual document model, as opposed to strong types encountered in data models for database applications: A weak document type is the specification of the conceptual components shared by a set of documents. So defined, a weak type only partially specifies the structure of its instances. Component definitions can be refined, thus yielding a specialization ("is-a") hierarchy of weak document types. A formalism called Conceptual Structure Definition (CSD) [39] represents the MULTOS conceptual document model and allows the definition of conceptual components for both document types and instances.

Viewing the conceptual structure as a tree, leaf nodes are called terminal or basic conceptual components, while non-leaf nodes are called intermediate or complex conceptual components. Multivalued components, indicating potentially multiple instantiations of those components in a document, are allowed in a type definition.

Figure 3 shows an example of the conceptual structure of two document types: "Generic_Letter" and "Business_Product_Letter" which is a Generic_Letter. The is_a link derives from the specialization of the component "Letter_Body". The "+" symbol attached to "Receiver" means that this component is multivalued. Notice that identically structured nodes, such as "Name" and "Address", can appear in more than one subtrees.

The document type hierarchy defines an organization of the office documents which is useful in document retrieval. Another, complementary organization is provided by the definition of document collections, i.e. sets of documents of possibly different types. Document collections are user-defined, either ad hoc or from query answer sets.

3. CLIENT PROCESSES
MULTOS clients are independent processes running on user workstations, which support the interaction with the users, issue requests to the servers and display results, thus rendering the entire functionality of the MULTOS system.

Before describing the client architecture we introduce the terms tool, task and working session.

A tool is a set of related operations, called tool operations, which allow the user to manipulate documents, document types, conceptual components and document collections. For instance, tools are used to create, update, archive and print documents. Document handling is a tool including such operations as insert, delete, move and display document. Tools are distinguished into basic and composite. A tool is called basic when all its operations are implemented with ad hoc procedures, whereas it is called composite when at least one of its operations is implemented using another tool (either basic or composite).

From the user viewpoint, a task is a unit of work performed towards the achievement of some goal(s). From the system viewpoint, it is a sequence of activations, suspensions and resumptions of several tools. Finally, a working session is a set of tasks performed by the user between a login and a logout.

Turning now to the client architecture, a client process comprises three layers of software, shown in Figure 4: the User Interface Level, the User Interface Manager and the MULTOS Function Level.
3.1. The User Interface Level

The User Interface Level (UIL) supports the actual media of man-machine interaction and comprises several software packages: the I/O support library which includes the device drivers for keyboard, screen, mouse and audio devices; a graphics package which provides primitives for presenting textual and pictorial information on a bit-mapped screen; and a window package which supports the creation and handling of windows on the screen. In addition to this third party software, custom made software provides operations not available through the packages, such as a hierarchical structure editor, a window with buttons, etc.

The UIL receives software interrupts and data from and sends requests for services and data to the User Interface Manager.
3.2. The User Interface Manager

The User Interface Manager (UIM) is the middle level of software coordinating the activities of the UIL and the responses of the MULTOS system. It contains the following modules: Request parser for parsing UIL requests; implementations of screen layouts and user interaction modes; storage handlers for objects presented at the UIL; data translation routines for some of the UIL requests; and interfaces to the MULTOS Function Level commands.

The UIM sends requests to the Local Workspace, the Multiserver Coordinator and the MULTOS Tools at the MULTOS Function Level, having the general form <request id, data, flags>. Both data and flags are optional. The responses received by the UIM have the general form <type of data, data>. For example, the type of data can be file id, error message, panic message, ODA tree, etc.

3.3. The MULTOS Function Level

At this level, the MULTOS functions are implemented, including the MULTOS Tools, the Local Workspace, the Multiserver Coordinator, and the Dialogue Manager.
3.3.1. The MULTOS Tools

The tools provide the main MULTOS user functions. The major tools are:

a. *Structure Browser* which allows users to navigate through, view and select document collections, types and conceptual components with related data.
b. *Document Browser* which allows inspecting individual documents and sets of documents at various levels of detail.
c. *Document Retrieval Tool* which integrates the necessary operations for retrieving documents from the servers or from the local workspace.
d. *Document Handling Tool* which provides manipulation of single documents and document collections.
e. *Document I/O Tool* which supports the acquisition and printing of documents.
f. *Document Preparation Tool* to provide document editing facilities and type assignment.
g. *Automatic Document Classification Tool* which provides knowledge-based automatic synthesis of the CSD of documents acquired through a network or a batch document entry system.
h. *Document Type Administration Tool* which provides manipulation of document types.
i. *System Administration Tool* for system administration operations, such as recovery, volume management, user authorization, etc.

3.3.2. The Local Workspace

This module provides the dynamic filing function. Documents received from a server or created by the user can be held in the local workspace for future use. Users view the local workspace as a local server with most of the capabilities of remote ones. As the resources of a client machine and the volume of local documents are supposed to be less than those of a server, only a generic document type with prespecified conceptual components and document collections are supported in the local workspace.

The rest of the modules of the MULTOS Function Level provide auxiliary, yet indispensable functions.

3.3.3. The Multiserver Coordinator

Certain user-level operations, such as moving a document from one server to another, issuing a query to more than one servers, etc., require issuing requests to more than one servers. Such operations are overseen by the multiserver coordinator. A daemon process, called *Name Server*, maintains information about active servers, their addresses in the network, their logical names and aliases, and user login privileges. See [5].

3.3.4. The Dialogue Manager

The dialogue manager of the client, in cooperation with the respective modules on the servers, handles the communication between the client and the servers to which connection has been established. The communication is in terms of the server command language, while documents are encoded according to the document model.
The functionality of MULTOS, provided by the tools and the local workspace of the client, is actually rendered through the man-machine interaction supported by the user interface. In the next section we describe the design of the user interface in detail.

4. USER INTERFACE DESIGN

The user interface (UI) is a critical part of the client process as it determines the level of user-system interaction and the community of prospective users. The objective of the user interface design is to make the full range of functions of the multimedia filing system accessible not only to the sophisticated computer users, but to everyone who may need its services, with minimum training.

A number of general principles [6, 12, 11, 22, 36, 37, 31, 34, 40, 1] have been observed in the UI design. First, the UI should support a variety of dialogue styles and levels of verbosity in order to accommodate both expert and novice users. Often users repeat certain sequences of operations during a working session. The UI should be adjustable to such individual behaviors by giving priority to frequent operations. Constant and incremental feedback as well as on-line help facilities are also essential for the user to master the available operations. Last, but not least, uniformity and consistency of the UI are required for efficiency.

Three modes of user-system interaction are supported by the User Interface Manager (UIM):

*Operation-driven interaction*

An operation requiring keyboard input gets this input through a suitable window which it generates near the area of the screen where the operation was invoked. When end of typing is confirmed, the window accepts the input and disappears.

*Tool-driven interaction*

A tool or tool operation can be selected with the keyboard or the mouse. If a tool operation has already been chosen then the objects over which it is to be performed can similarly be selected.

*Object-driven interaction*

In a browsing environment a displayed object can be selected so as to view the operations available for it. Objects can belong to one of the following classes: document types, conceptual components, documents and document collections.

In tool-driven and object-driven interaction pop-up menus are used for the display and selection of available options.

These modes of interaction provide adequate degrees of freedom as well as guidance for using effectively the functionality of the system without having to know it in advance.

To deal with sets of objects, the concept of *container* is introduced. A container is a temporary collection of objects defined by the user at run time, the members of which can be passed as arguments to MULTOS commands. The scope of a container is determined by the environment in which it is defined. Thus, there is one global container, used for the exchange of information between different tasks and tools, while local containers can be defined within individual tools. Container operations include:
Clean, Move and Copy (for the entire content of the container); List (the members of the container); Delete, Move and Copy (for individual container members).

Containers are represented by small icons.

4.1. Screen layout
On connecting to the system the user gets a ribbon window as shown in Figure 5.

![Ribbon Window Diagram](image)

Figure 5: General screen layout

The ribbon window is the main system window that provides the basic environment where tools run. It consists of either a horizontal or vertical area, the ribbon, where labeled icons (64 x 64 bit maps with labels) represent tools (or tasks) instantiated by the user. At system startup time, one instance of each tool is displayed by default on the ribbon. Tool instances may be added on the ribbon by selecting the desired tool from a menu and adding the corresponding labeled icon on the ribbon. Multiple instances of a tool are allowed. The area next to the ribbon belongs to the system global container and is used to display its objects. Both the ribbon and the container may extend on either side of the visible area, thus scroll bars are provided for viewing.

4.2. Tool activation
When a tool is instantiated a characteristic icon is presented on the ribbon. As mentioned, more than one instances of a tool can exist. For example, several retrieval tools may be running, one performing query formulation, another document browsing, etc.

In order to interact with a tool the user has to open it. More than one tools can be open on the screen simultaneously, yet interaction is possible with only one of them, called foreground tool. All other tools, if any, are either background, stopped or inactive. A tool is made background in order to change the tool the user interacts with (foreground), while letting the previous tool continue unfinished operations (e.g., query execution) or perform operations that are asynchronous to user activity (e.g., send/receive
Mail). Foreground and background tools are active. On the other hand, a tool is stopped as a result of suspending its operation and inactive if it has not been activated since its instantiation. Finally, a tool instance is released with a quit operation and disappears from the ribbon.

Each tool has a tool status. The status information is automatically saved when a tool is suspended and is restored when the tool is resumed, so that operation can continue from the point of interruption. Inactive and background tools can be closed back into icons to maintain a cleaner screen. Should an important event occur in a closed background tool, its icon flashes.

The display of information by a tool can be done in one or more windows in either graph or list form.

In abnormal situations a message is generated in a window next to the mouse position on the screen. All operations are frozen until the user takes some action, possibly suggested by the message.

4.3. Pop-up menus

The UI includes a three-button mouse with the following button functions: the left button is used for tool activation, selections and commands; the right button for complementary selections, such as value ranges; and the middle button for inquiries about objects pointed to, or for invoking the appropriate menu depending on the area the mouse resides in.

Five kinds of pop-up menus can be obtained using the mouse. Some of them have a fixed, predefined list of selections, while others are dynamic, i.e. the list of selections depends on the tool or object which the menu appears on. These are:

a. System menu, used for general purpose operations like window or task management.

b. Ribbon menu, used to customize the appearance and order of tools on the ribbon. The available commands are: move, for changing the place of an icon on the ribbon; quit, for dismissing an icon; copy, for multiple instantiation; rename, for changing an icon label; and create, for creating a new tool. The last one is mainly intended for expert users and system administrators.

c. Tool menu, used to activate/deactivate tools. Tool status is indicated by the corresponding icon on the ribbon.

d. Tool operation menu, used to activate operations of the foreground tool. Only allowed operations, determined by the tool status, are displayed. For instance, a sort operation is included in the menu of a structure browser displaying the document type catalogue in list form, but not if the display is in graph form.

e. Object operation menu, used to supply the operations applicable on an object pointed to (object-driven interaction). The operations offered for each object class are listed in Table II. Again, some operations may be disallowed like in the tool operation menu.

Finally, pop-up menus feature acceleration: a selection of the menu can be connected with one of the mouse buttons, which then directly initiates the corresponding action.
<table>
<thead>
<tr>
<th>Documents</th>
<th>Edit, I/O, Communication, Restriction, Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document Types</td>
<td>Selection, I/O, Communication, Restriction, CSD Display, Refinement</td>
</tr>
<tr>
<td>Document Collections</td>
<td>Selection, I/O, Restriction, Document Display, Collection Handler</td>
</tr>
<tr>
<td>Document Conceptual Components</td>
<td>Selection, I/O, Restriction, Description Display</td>
</tr>
</tbody>
</table>

Table II: Object class operations

5. DOCUMENT RETRIEVAL

In MULTOS, as well as other multimedia document systems (e.g. MUSE [33], MINOS [18]), the multimedia document is the unit of information, containing its components as parts which belong exclusively to that document and are only accessed through it. Alternatively, components may be shared by various documents and enjoy relative autonomy: multimedia documents are composed from parts through a set of relations. This view is adopted in object-oriented multimedia databases, e.g. see [45] (also by MINOS, to a limited extent). In either case, retrieval of a document can be based on a number of clues: the content and structure of components, the structure of the document as a whole, and semantic associations between components. Thus, in MULTOS, a retrieval strategy is realized combining several access methods in a complementary capacity: access by document identifier, which involves scanning a set of document instances; index access, which involves scanning an inverted file of document component values (attribute index) or its B-tree; text signature access, which involves scanning a set of text signatures; and image signature access, which involves scanning image signatures. These access structures are reviewed in the Appendix. The point of interest here is how a query language which expresses complex conditions on multimedia documents can be effectively supported by the user interface.

In document retrieval, users tend to be vague about what they are looking for and often have difficulties in expressing it. Besides, multimedia documents can have very rich structure and content. The resulting requirements for the user interface are that it should support flexible, dynamic query formulation and resolution of user uncertainty, while maintaining simplicity of use. In MULTOS, the function of document retrieval is performed with the Document Retrieval Tool. Before describing its design and operation, it will be helpful to review the underlying query language.

5.1. The MULTOS query language

The MULTOS query language has been presented in [9].

Queries have the following general form:
FIND DOCUMENTS [VERSION version-clause;]
   [SCOPE scope-clause;]
   [TYPE type-clause;]
WHERE condition-clause;
(Square brackets enclose optional items.)

Document versions form a sequence, so restrictions like LAST, FIRST and ALL can appear in the version-clause. The scope-clause is used to restrict the query on a particular document collection. The type-clause restricts the query on documents belonging to one or more document types and their subtypes.

The condition-clause can express conditions on both the content and the conceptual structure of documents, and is a boolean combination of predicates on document components. The general form of predicates is

WITH component
for conditions on structure and
component restriction
for conditions on content, where component is, in both cases, the name of a conceptual component, and restriction is an operator followed by an expression. Usually a component is referenced not just by its name but, rather, by a pathname of the form

\[ \text{name}_1 [\text{\cdot}^*] \cdots \text{name}_{n-1} [\text{\cdot}^*] \text{name}_n \]

This means that the referenced component is name\(_n\) and is contained within component name\(_{n-1}\), etc. Component names in the pathname are separated by either a "." or a "*", the "." denoting direct inclusion and the "*" transitive.

The WITH operator allows the specification of conditions on the existence and structure of conceptual components. Queries on content, on the other hand, can address formatted data (attributes), free text and images.

Predicates on attributes have the form

\[ \text{component rel op constant} \]

where rel op is one of the operators: =, ≠, <, >, ≤, ≥, BETWEEN (for range conditions) or LIKE (for partial match of strings).

Predicates on text have either of the forms:

\[ \text{component CONTAINS string-expr-list} \]

or

\[ \text{TEXT CONTAINS string-expr-list} \]

where string-expr-list is a list of string expressions and a match is required for each string expression. In the first case the query is restricted on a particular text component, whereas in the second it addresses all text components of the searched documents.

Predicates on images have either of the forms:

\[ \text{component CONTAINS image-expr-list} \]

or

\[ \text{IMAGE CONTAINS image-expr-list} \]

where image-expr-list is a list of symbolic images and a match is required for each symbolic image. In the first case the query is restricted on a particular image component while in the second it addresses all image components of the searched documents. Each
symbolic query image has the form: \((domain-name)\)symbolic-image-subquery, where symbolic-image-subquery contains at least one of the following clauses: \([cluster-clause]\)\([object-clause]\). The object-clause is a symbolic representation of the query image containing a list of objects, while the cluster-clause specifies a cluster of similar images.

Comparisons between components of the same kind are possible through predicates of the form

\[ \text{component rel_op component} \]

The optional quantifiers in the form

\[ [\text{ALL/SOME}] \text{ component restriction} \]

apply the restriction to all or at least one of the values of a multivalued component. Another special form for multivalued components is

\[ \text{component IS IN set-expression} \]

restricting the values within a particular set.

An example query (adapted from [9]), referring to the document types of Figure 3, is the following:

```
FIND DOCUMENTS WHERE
  Document.Date > /30/5/1989 AND
  (*Sender.Name > "Olivetti" OR
   * Product_Presentation CONTAINS "Olivetti") AND
  * Product_Description CONTAINS "Personal Computer" AND
  (*Address.Country > "Italy" OR TEXT CONTAINS "Italy") AND
  WITH * Company_Logo;
```

5.2. The Document Retrieval Tool

From the retrieval standpoint, the set of all documents in any filing stage may be divided into two partially overlapping sets: a search space and an auxiliary space, the content of which is listed in Table III. The search space is the set of documents where retrieval is to be made from, while the auxiliary space includes documents which may be helpful to the user in formulating or altering a query. The classification of documents into document types and document collections induces an organization of the document spaces which aids navigation through them and enables the restriction of queries on relevant subsets of the search space.

The document retrieval tool integrates all the functions performed during the retrieval of documents from the servers or from the local workspace [19]. These functions are provided by special components of the document retrieval tool as follows (also see Figure 6):

a. **Structure Browser** for browsing through the document type, conceptual component and document collection catalogues kept in the servers and the client.

b. **Query Editor** that supports both expert and novice users in formulating queries.

c. **Query History** that provides information about queries issued in the current working session.

d. **Query Execution** for issuing queries.
<table>
<thead>
<tr>
<th>SEARCH SPACE</th>
<th>AUXILIARY SPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>• archival store (types, collections)</td>
<td>• query catalogue</td>
</tr>
<tr>
<td>• temporary collections (derived from query answer sets)</td>
<td>• temporary collections</td>
</tr>
<tr>
<td>• permanent private documents</td>
<td>• permanent private documents</td>
</tr>
<tr>
<td></td>
<td>• temporary documents</td>
</tr>
</tbody>
</table>

Table III. Document spaces pertinent to document retrieval

c. **Document Browser** for browsing through documents qualifying as query answers or otherwise selected.

The component tools of the retrieval tool and their functions will be discussed further below. It will be shown that, with the structure defined above and the functionality offered by the general user interface design, the retrieval tool has a set of features which support the efficiency and effectiveness of the retrieval process. The retrieval of multimedia office documents is considered a *loosely structured* process which, therefore, requires flexibility on the part of the supporting system.

The main features of the retrieval tool are:

- a. Query-by-example [46, 13] is supported, which allows the specification of restrictions by appropriately furnishing attribute values, text strings or sample pictures.

- b. The search space, the auxiliary space, as well as individual documents, are displayed at the desirable level of detail. The display of the search space (e.g. type catalogue, collection catalogue, private documents) informs the user about the available information in an organized manner, while the display of the auxiliary space helps the user decide what to ask for.

- c. Browsing is performed repeatedly during the retrieval process to guide query reformulation and to reach final document selection, thus being an important instrument for resolving user uncertainty. Fast browsing through sets of documents using miniatures, as well as through individual full documents is supported.

- d. Easy transition and transfer of control between the component tools of the retrieval tool is important for flexibility in man-machine interaction. For instance, while editing a query, the user decides to browse a local collection where an interesting document is located; this is then examined in greater detail and a word is found in it, which is transferred to the query editor as a text restriction. The high degree of flexibility provided is apparent from Figure 6.
from associated components, e.g., an image caption is a text component referring to an image component. The components involved in a query are called active, while those not involved are called passive. Components are activated by selection from the menu. The sequence in which components are activated is unimportant. The middle part of the window is the editing area where queries are edited graphically. Even very complex queries involving all document components can be formulated in simple, guided steps. In the bottom part of the window, the query is displayed in the syntax of the query language. This subwindow can also be used for typing queries directly in the query language.

Figure 7: The query editor

Queries are supposed to be in conjunctive normal form. In the graphical editing area, intended mainly for unsophisticated users, complex restrictions are formulated in
expanded form, without parentheses. Naturally, they can be concisely expressed using the query language directly in the appropriate window. The expansion of query restrictions occurring in graphical editing is illustrated in section 5.3.1.

5.3.1. Attribute queries

Assuming that a particular document type or document has been chosen from the search space, the conceptual structure of that type or document appears in an attribute specification window in the editing area (Figure 8). The conceptual components of this structure can be edited using operators from a dynamic pop-up menu. The available operators depend on the type of the component, i.e. operators for numeric components include EQUAL, LESS, GREATER, operators for string components include LIKE and operators for date components include BETWEEN. The user is also allowed to extend the displayed structure by adding nodes and links, possibly including unspecified intermediate nodes. While the user specifies restrictions on component values, the system assumes an AND operator between the predicates. ORing predicates is allowed by reselecting the type or document and specifying the predicate that will be ORed with the previous one. The following illustrate the process:

Example 1:

```plaintext
find version all; type unix_1;
where *example_call = "who(1)" AND *reference = "WHOAMI";
```

The sequence of user actions for the above query is the following: The user selects the unix_1 type and the type definition is presented in the attribute specification window; component example_call is selected and its value specified as "who(1)"; component reference is selected and its value specified as "WHOAMI". The query is now ready for execution.

Example 2:

```plaintext
find version all; type unix_1;
where *name = "whoami-print effective user id" OR *reference = "who";
```

The user selects type unix_1 and specifies that the value of component name is "whoami-print effective user id", then again selects type unix_1 and specifies that the value of component reference is "who". The two attribute specifications are finally linked with the OR operator.

Example 3:

```plaintext
find version all; type unix_1;
where NOT *reference = "exist" AND
   (NOT *reference = "all" OR NOT *example_call = "current");
```

The user selects type unix_1 and specifies that component reference should NOT have the value "exist" AND should NOT have the value "all", then again selects type unix_1 and specifies that component reference should NOT have the value "exist" AND component example_call should NOT have the value "current". Query formulation is completed by ORing the two type specifications.

The access structure used in processing attribute queries is the attribute index.
5.3.2. Text queries

Queries on textual content can require the presence or absence of a number of words, conjunctively or disjunctively. For editing such queries, a text specification window is opened in the query formulation area where the required words are entered (Figure 9). A pop-up menu will display on request the available operators: AND, OR, DONE. The list of operators is extensible. For instance, a LIKE operator is needed to support similarity retrieval. The use of the operators has been illustrated in section 5.3.1.

Words entering the query specification may well have their origin in document browsing. Moreover, the decision about using a word in a query may be made just after having seen this word. So, we find useful to offer the possibility of opening a document browser window during query formulation, from which words can be simply copied into
the query specification area. Spelling errors, typing effort and having to remember words are thus minimized.

The access structure used in processing text queries is the text signature file.

5.3.3. Image queries

Figure 10 shows the layout of the query editor interface as it appears for queries on pictorial content. The main window here is the image specification canvas which offers a query-by-pictorial-example [13] interface. Two kinds of image query terms are distinguished: graphic images and bitmap images. Graphic images can be drawn using the primitives of a graphic package and/or predefined application domain-specific primitives. Bitmap images can be input through a scanner. Either kind of image can also be loaded
from some existing file. The contains window is used to construct the query restriction in conjunctive normal form, the terms of which \((image_1, image_2, ..., image_n)\) are local, temporary names for images specified through the image specification canvas. A pop-up menu offers the operators AND, OR, DONE. Other auxiliary windows, such as catalogue of image classes, image class vocabularies, text specification window, scanner user interface and object attribute specification window, are available through a menu (see below).

![MULTOS Ribbon Window](image)

**Figure 10: Image query**

Although image query terms can be supplied in the form of either graphic or bitmap images, query processing is based on a symbolic image representation. A symbolic image consists, in general, of objects belonging to a given finite vocabulary. In addition, it may describe relations among those objects, and properties of individual objects as well as of the image as a whole. The vocabulary can be generic, such as the set of
primitives of a graphic package, or it can be specific to some application domain. The images related to some domain constitute an **image class** and the corresponding object vocabulary, which represents the class, is the **image class vocabulary**. A domain-specific vocabulary consists of custom-made graphic primitives, perhaps together with the generic ones, or, for bitmap images, of a set of objects which have to be recognized either by a human or by a specialized image analysis and understanding module. As such modules are highly application-specific, it is not our concern to include one in our prototype system. Rather, a graphic package is made available through the image specification canvas, offering generic primitives and supporting the composition of new ones. Furthermore, manual classification of the objects in bitmap images is supported by the image specification canvas.

Object and image features are described by object and image attributes respectively. Finally, relations among objects considered here are relative position in space and composition (of complex objects from simpler ones).

In conclusion, defining image query terms on the image specification canvas results in attributed symbolic images that express restrictions on the semantic content and structure of document image components. It is these symbolic images that are stored in temporary files referred to as $image_1, \ldots, image_N$ in the above description of the **contains** window.

In addition to direct image query specification, restrictions on associated text, such as image captions and in-text references, can be used to indirectly address the pictorial content of multimedia documents. This is supported by a text specification window available through the menu of the image query interface.

The access structures used in processing direct image queries are the **image signature file** and the **pictorial object index**.

### 5.4. The Query Execution Tool

When query execution is activated, the current query is sent to the servers or to the client (if local) for processing. In order not to wait for the answer of the system, the user can close the retrieval tool. By flashing the corresponding tool icon on the ribbon, the system informs the user of the end of processing. The initial response of the system is displayed by the query history tool.

### 5.5. The Query History Tool

The query history is by default displayed at the bottom of the query editor (Figure 11). The query history tool enables inspecting previous queries and their qualifying documents. For each query, the tool displays the order of the query in the current working session, whether it was evaluated in the servers or in the local workspace, and the number of qualifying documents.

By pointing to the query number, a window is opened where the corresponding query is displayed. The selected query becomes the current query. The user can inspect, but not edit, the query in that window. To reformulate the query, the query editor has to be opened.

By pointing to the number of qualifying documents, a window is opened where icons of the documents are displayed. If a document has been fetched from a server, the
corresponding icon is highlighted. Icons can be selected by the user and the following operations are allowed:

*Display:* The document browser is invoked for document inspection.

*Copy to container:* The document is copied to the container specified. This option allows the definition of temporary collections from previous query answers, to be used in subsequent query formulation.

### 5.6. The Document Browser

A query answer set may contain irrelevant documents due to user uncertainty in query specification, inexact matching techniques (e.g., text signatures [28]), or both. Document browsing is thus the natural final stage of document selection. Moreover, browsing can
be useful in query formulation: inspecting the important features of documents accessed by a previous query, or of those belonging to a particular document collection, or of an individual document, can help the user decide what exactly to look for. Thus, the combined use of document browsing and dynamic query formulation support the resolution of user uncertainty.

Two levels of browsing are provided: inter-document browsing, i.e., browsing through a set of documents, only displaying relevant features; and intra-document browsing, i.e., browsing through individual documents in any desired detail.

5.6.1. Inter-document browsing
This is browsing through a set of documents, most often a query answer set, and aims at quickly inspecting the important features of the documents, so that relevant documents (or even just promising, at an early stage) can be identified.

Inter-document browsing comprises two stages of feature presentation. In the first stage, only the features are retrieved from the server, not the full documents. The purpose of this stage is to support a fast screening of the documents. Interesting documents are marked after feature inspection. In the second stage, marked documents are fully retrieved and more features presented. Thus fetching full documents from the servers, which causes response delays and network congestion, is avoided when possible.

Four levels of detail are supported during the presentation of document features:

a. Simple card view: This operation allows the user to view the simple cards associated with the selected documents, containing such information as name, doc_id, size, owner, date of last modification, etc.

b. Type card view: Each document type has an associated type card (defined and possibly updated at type definition/update time) that contains the most significant conceptual components of that type.

c. User card view: The user can define private user cards by selecting some conceptual components and putting them in the card view. A user card can be specified as default card for document display.

The information in the above cards, used in the first stage of presentation, is inserted together with the document during its creation, either manually or automatically. Though possibly redundant, it is better to store this information separately from the document so that its retrieval will be fast and will not require the retrieval of the entire document.

d. Query card view: A query card contains the extra features presented during the second stage of document feature presentation. These are component values which satisfy query restrictions. For example, if a query requests documents that contain a specific image, the corresponding image component is regarded as an important feature of the document, and the matching images are shown in the query card view.

In both stages of feature presentation, it is necessary not only to see the presented information but also to be able to compare its occurrences in different documents. In order to fit more than one document on the screen of the workstation, the information associated with each document is presented in miniature form.
A text miniature is created in real time by using a smaller point size. An image miniature is created using an image reduction technique and stored together with the document. A voice miniature consists of a short part, e.g. first three seconds, of the voice recording.

5.6.2. Intra-document browsing

This is browsing through a fully retrieved individual document. During intra-document browsing, the user can move from one page to another (both forward and backward), use a scroll bar to define the approximate desired location, or define the exact page number. Additionally, the user can move to a page that contains a specific word, image or voice recording. Pages are presented in their real size unless they do not fit in the screen. In this case, scrolling inside the page is also provided. Pages may be folded if they exceed the size of the regular pages of the document and can be unfolded by the user. Finally, browsing inside a document can be based on its attribute values. The attribute values of the document are presented in a separate window next to the document presentation window. The user can point to an attribute value and see, in the presentation window, the page that contains it.

5.7. The Structure Browser

The structure browser supports navigation through, and inspection and selection of the structures of the document types, document collections and document conceptual components. It operates on the catalogues of document types, collections and conceptual components, and on individual document types and collections. The following functions are offered:

a. Selection. An object belonging to the structure currently being browsed can be selected, i.e., a document type from the type catalogue; a collection from the collection catalogue; or a conceptual component from the conceptual component catalogue, a document type or a collection. When a document type is selected, all the conceptual components included in that type are implicitly selected too. Selected objects can be used in query formulation.

b. Navigation. The user can view the entire structure (reduced to a miniature, if necessary) and the position of all the windows open on it. Navigation is performed by either using vertical and horizontal scroll bars, or by choosing relative position on the global view of the structure with the mouse. During navigation the displayed objects are mouse-sensitive for object-driven interaction. For example, by clicking over a particular type during navigation through the type catalogue, the display operation can be invoked to present the conceptual structure of that type in a subwindow, as shown in Figure 12. Two navigation modes are supported:

List mode: All data in the system can be displayed in list form.

Graph mode: Hierarchies can be presented as graphs either vertically or horizontally (see section 6, class treeview for details).

c. Sorting. During list mode navigation, the list can be sorted by some of the list object fields (name, time stamp, etc.).

d. Set defaults. A default document collection and document type can be defined for use by the system when not otherwise specified by the user.
6. Implementation

The implementation environment of MULTOS consists of SUN-3 workstations running Sun UNIX 4.2 Release 3.4 connected through an Ethernet local area network, an Optinnem 1000 optical disc and a Datacopy 730 scanner connected to the network through an Olivetti M24. For the communication between server and client processes, the virtual circuit protocol TCP/IP is used. Server processes are entirely written in the C programming language and use the CISAM indexing package for maintaining attribute indices. Client processes are developed on the SUN workstations using the ANDREW toolkit (Version 1.0) [35] on top of the X window system (Version 11, Release 3). ANDREW provides an object-oriented environment for building programs using the C programming language. In addition, ANDREW provides tools which complement
MULTOS with other multimedia document management functions, in particular a multimedia editor (ez) and an image editor (zip). The chosen software environment is fairly portable and allows installation of MULTOS in various hardware configurations.

ANDREW has a set of fundamental object classes and allows the programmer to create new classes suitable for specific applications by using a single inheritance mechanism to build complex object classes from simpler ones. In building the MULTOS client, the objects were first specified and then implemented using existing ANDREW classes or, if necessary, by creating appropriate subclasses.

The object classes of ANDREW fall into two major categories: data objects and views. The class dataobject provides the data and methods needed to maintain, store, and retrieve information. On the other hand, the class view provides all that is needed to display information in a rectangular area on the screen, request display updates and respond to such requests, and respond to input events generated by the user. In general, a view can be seen as an input/output interface to a dataobject, which represents memory or permanent storage.

In designing the MULTOS objects, reusability and system maintenance were among the main concerns. Although object-oriented programming is by itself a step towards reusability and maintenance, none of them is achieved automatically. Effort was made to identify low-level objects that could be used by as many of the tools of the client as possible. Each tool can be viewed as a complex object that is composed of simpler ones. Low level elements of a user interface, and MULTOS is no exception, are windows, menus, icons and labels that represent entities of the system on the screen. ANDREW supports windows and menus but not icons. In MULTOS icons are used to represent tools (in the RibbonWindow), documents (in the QueryHistory and the DocumentBrowser) etc. Usually, it is desirable to have a label associated with an icon to provide additional information about it (e.g. a name). A label may even replace an icon if the user prefers so. Hence the need to define new classes to support icons and icons with labels.

First is defined the class icon as a subclass of raster which exists in ANDREW and handles bitmaps. The size of an icon is limited to 64x64 pixels. The object class labeledicon is derived from the class icon by refinement. Information about the label is added to the data of the class, while some methods need to be overwritten (replaced) since they are not valid any more. In Figure 13 the ANDREW definition of the labeledicon class is shown.

As described in previous sections, the ribbon window, which is the main system window, is a ribbon containing labeled icons that represent tools. Each tool is seen as a separate process, therefore an object class, called iconicprocess, is created as a subclass of labeledicon, to handle labeled icons with associated processes. The new class keeps information about the name of the associated process, its argument list, its status (inactive, running, stopped), and its process identifier when execution starts, in addition to the information it inherits from its parent (labeledicon). It also provides the methods to execute, kill, suspend and resume the process. Class iconicprocessview is a view for an iconicprocess.

Having defined iconicprocess, the ribbon is a list of iconicprocesses with attached menus and scrollbars. Thus two objects are defined, the list, which is a subclass of
#define labeledicon_ChangedLabel 1
#define labeledicon_ChangedFont 2

class labeledicon: icon {
    overrides:
        Read(FILE *file, long id)
            returns long;
        Write(FILE *file, long writeid, int level)
            returns long;
    class procedures:
        InitializeObject(struct labeledicon *self)
            returns boolean;
    methods:
        SetIconLabel(char *label);
        SetFontName(char *fontname);
        SetFileName(char *filename);
    macromethods:
        GetIconLabel() ((self)->iconLabel)
        GetFontName() ((self)->fontName)
        GetFileName() ((self)->fileName)
    data:
        char *iconLabel;
        char *fontName;
        char *filename;
};

Figure 13. Labeledicon class definition
dataobject, and the listview, a subclass of view. The list uses a single linked list to manipulate the iconic processes and provides methods for inserting, deleting or accessing its data, as well as storing/retrieving its content to/from a file. The listview is responsible for presenting the contents of the list on the screen, and provides scrollbars and menus to make viewing and manipulation more convenient to the user. For example, the insertion of a new object to the list is carried out through a menu selection, a multiple choice selection for choosing an object from a predefined list, and a mouse click for specifying the place of the insertion (screendump). A double click on an iconicprocess executes the process.

The list and listview objects are designed to communicate with each other by sending appropriate notifications (messages). When, for instance, a new item is added to the list, the listview receives an insertion notification, so that it can keep its view up to date without the programmer having to call a method requesting this update.

Using the above object classes, the ribbon is implemented as a list of iconic processes. Each tool corresponds to an iconic process object that knows how to execute, suspend, kill and resume itself upon user request. New tools are added to the ribbon by adding iconic processes to the list. Tool removal is analogous. The Ribbon Window, however, also includes a container part, which is a list of markbuffers. A markbuffer is an object that serves as a temporary buffer, where dataobjects can be stored in or retrieved from. It provides methods much like the ‘Cut-and-Paste’ operations of text editors or window systems. The container being a list of markbuffers suggests that it could be implemented using a list suitably modified to handle markbuffers.

One approach to the construction of the modified list was to create new objects by duplicating the code of the list with slight modifications, so that different objects could be manipulated. An alternative was to redesign the object and its methods, so that it would become more general and could handle any dataobject, then create specific lists by sub-classing the general one. The second approach was chosen, and the general list was designed in such a manner, that only a two-line method has to be rewritten when creating sub-classes. This method is called when an instance of a new object is created. The general list is a list of dataobjects while the container, for example, is a list of markbuffers, which is a subclass of dataobject. Thus, the container is created as a subclass of the general list and the method NewObject() is overwritten and written so that it creates markbuffers. Once a markbuffer is created, through its own New() method, it can be handled as a dataobject, so no other part of the general list really has to be modified.

This approach required more effort than the first one, because parts of the object had to be redesigned, and extensive changes of the code had to be made. On the other hand, specializing the list to create a container is trivial and the list is easily reused.

The listview was also changed accordingly, and code was added so that it could display the contents of the list in either a horizontal or vertical layout with virtually no effort. The definition of the list is shown in Figure 14. Figure 15 shows how the objects defined above are combined to create the Ribbon Window.

A significant part of the information handled by MULTOS such as document types, document collections and document conceptual structure, has a tree structure. The following object classes have been constructed to support the definition, editing and display of tree structures:
/* Notifications sent when an update of the view is needed */
#define list_Add 1 /* insertion */
#define list_Delete 2 /* deletion */
#define list_Overwrite 3 /* overwriting */
#define list_Cleared 4 /* deletion of all the contents */

/* the data structure */
struct objectlist {
    struct dataobject *object;
    char *fileName;
    struct objectlist *next;
};

class list: dataobject{
    overrides:
        Read(FILE *file, long id) /* Method for reading in a list */
            returns long;
        Write(FILE *file, long writeid, int level)
            returns long; /* Method for saving a list */

    methods:
        /* Add a new member, stored in fileName, at position place */
        AddMember(struct dataobject *member, char *fileName, int place);
        ...
        /* Return the object stored at position place */
        GetMember(int place)
            returns struct dataobject *;
        /*
        * Create a new member that will be inserted in the list.
        * This is the only method to be overwritten when sub-classing
        * the list.
        */
        NewObject()
            returns struct dataobject *;

    macromethods:
        /* Return the number of objects in the list */
        GetPopulation() (self->numberOfMembers)
        ...

data:
    int numberOfMembers;
    struct objectlist *objectList;
    ...
};

Figure 14: Definition of the class list
a. Class node which keeps information about the contents of a node and has methods to edit and communicate this information.

b. Class nodeview which performs the display of the corresponding node on the screen.

c. Class tree which keeps information about an entire tree and has methods that operate on it. Currently supported operations include: read/write a tree from/to a file; select, add, and delete a node and its sub-tree; and copy/move a node or a sub-tree from one location to another.

d. Class treeview which performs the display of the corresponding tree. Four choices of display layout are currently available resulting from a combination of two tree orientation schemes with two node positioning schemes. The tree orientation schemes include vertical and horizontal display, where children nodes are displayed below or to the right of their parents respectively. The first node positioning scheme displays each node centered above/next to its children. The display space required depends on the depth and the number of leaves of the tree. The second scheme displays the nodes of each level in the tree equally spaced. The display space is reduced as it now depends on the depth of the tree and the number of nodes in the widest level. On the other hand, the second scheme sometimes achieves space economy at the price of a relatively cluttered view of the tree structure.

The above classes are used by the Structure Browser and the Retrieval Tool to display and edit the type hierarchy and the document conceptual structure. For example, in editing an attribute query, the user can request to see the definition of a document type.
The type definition is loaded from the server and becomes a tree object while each conceptual component is an object of the class node which knows about its type and its valid relational operators. This information is presented through the corresponding view objects. The user can then modify the displayed structure using the node and tree class operations. Text specification, as well as the direct query language editing are implemented using instances of the text and textview classes defined in ANDREW [35].

In Figure 16 the design of the Retrieval tool is presented schematically indicating the classes used to implement its parts. Comparing figures 15 and 16, the reuse of classes is obvious.

Figure 16: Retrieval tool

7. CONCLUSION

We have described interactive multimedia document retrieval in an integrated multimedia filing system, discussing in particular issues of user interface design posed by the required functionality, an approach to solving them effectively and a prototype implementation.
The retrieval tool presented offers multimodal retrieval of multimedia documents under a uniform interface, including retrieval by pictorial content; extensive browsing capabilities through information structures, sets of documents and individual documents; management of information spaces pertinent to the retrieval function, including query history; interactive graphical query editing as well as direct use of the query language; and maximum flexibility in alternating between and iterating through these functions.

The implementation of all the functions of the retrieval tool has been completed with exception of the document browser which is in progress. A desirable feature, not included in the original design, is nonlinear navigation during browsing in a hypertext-like manner. The basic device proposed for the resolution of uncertainty, due either to the user or to inexact matching, is the operational flexibility of the retrieval tool itself, supporting the user to (a) decide what to look for and (b) identify an acceptable answer. Other methods for the resolution of uncertainty, such as rankings based on subjective importance and similarity measures (e.g. [24, 25]) can be incorporated under the same interface with minor additions, which is actually pursued.

The MULTOS client has been integrated with the ez multimedia editor and the zip graphical editor, both independent applications of ANDREW. Although these are not filing functions, their being directly accessible from a filing system is important for overall operational efficiency. Other systems, such as MINOS [18], have included editing as part of their own design. Integration with the outside world raises issues of handling document input and output and achieving device transparency. Automatic document classification and document exchange according to the ODA standard are features of the MULTOS client promoting integration, yet there is more to be done.

The development of the MULTOS system has taken place in two stages producing, correspondingly, two prototypes. The first prototype mainly focused on the server with only a demonstrator client. The second prototype includes a full client and enhancements to the server, such as pictorial query processing and optical disk management. The design of the client drew on the experience gained from the first prototype and a related experimental filing system, MUSE [33]. The merits of the object-oriented design and implementation of the client cannot be overstressed. Code reuse, extensibility, maintainability and the ease of integrating software produced by geographically scattered co-workers are among the benefits reaped.

Planned for the immediate future are the creation of a substantially large and varied data base of multimedia documents and the evaluation of the functionality and performance of the system in particular application environments.
APPENDIX
A MULTOS server process is composed of a number of specialized components as described in section 2.3. Among these, the document storage subsystem (DSS) allows the storage of multimedia document content, maintains the access structures and provides access methods for document retrieval. The architecture of the document storage subsystem is shown in Figure 17. The document handler is responsible for the communication of the document storage subsystem with the other server subsystems. During document insertion, it receives MULTOS multimedia documents composed of the Andrew datastream and the CSD structure, decomposes the documents into the different types of information (CSD, text, image) and builds temporary entry maps for each data type to allow creation of the retrieval access structures.

![Diagram of Document Storage Subsystem General Architecture](image)

Figure 17: Document Storage Subsystem General Architecture

The text retrieval, image retrieval and csd handlers, using the entry maps created by the document handler, create the appropriate access paths for the different data types and have storage containers for the content and the access structures of the document.

In the following we will describe the access structures used in MULTOS.

Index access
The CSD Handler stores document CSD structures received by the Document Handler and creates attribute indices. The attribute indices built by the CSD Handler allow conceptual components to be accessed both by name and by value. All the conceptual components belonging to a CSD associated with a document stored in the DSS, are present in a table (CC_List) composed of the cc_name field and the index field that specifies
whether or not the component is indexed. If the index exists, it maintains all the information regarding the document associated with the corresponding conceptual component. Otherwise, another table (LDI_List) contains the list of document identifiers associated with the conceptual component, their corresponding type, the conceptual component identifier (cc_id), used to univocally identify the conceptual component within a document type with more than one conceptual component with the same name, and a pointer (offset and length) to the Data_List containing their value.

During the creation of an index on a specified conceptual component, indexing is automatically extended to all the documents containing that conceptual component.

Document retrieval is performed by specifying either the presence of a particular conceptual component or the value associated to a CSD attribute. The result of an index scan is the set of LDIs of the documents satisfying the search condition.

It must be noted that, contrary to the other Retrieval Handlers that only maintain links to their data inside the document stream stored by the Document Handler, the CSD Handler also provides primitives for storing the entire CSD and for rendering it both as a whole structure and as individual conceptual components. To directly access them, specific Entry Maps have been built.

Text access

For accessing the text part of multimedia documents, MULTOS has chosen the method of text signatures with superimposed coding [29, 28]. According to this method, the text in each document is divided into logical blocks with a fixed word length. A signature for each block is generated by ORing the signatures of all the non-common words of the block. The signature of a word is a bit pattern of length F which is generated by hashing the word. The hashing function sets m bits to "1" in the bit pattern.

The document signature is represented by the concatenation of the signatures of its blocks and the signatures of documents are stored in a sequential text signature file managed by the text retrieval handler. New document signatures are simply added to the end of the file.

In order to query the archive for a word w, the same transformation is applied to w and the m corresponding bits are determined. The resulting bit string is referred to as query descriptor. Then, all the indicated bits are checked in all the signature blocks by scanning the signature file. A signature scan has as input parameters a set of text conditions \( T_1, T_2, ..., T_n \) and an optional set of document identifiers. A text condition has the form \( e_1, e_2, ..., e_m \), where \( e_i \) is a string expression list. A document satisfies a text condition if it satisfies at least one of the string expression lists in the set. A document satisfies a string expression list if it contains all the string expressions contained in the list. By passing the text retrieval handler several string expression lists, one can evaluate all the text predicates within a query with only one signature scan. When a set of document identifiers is specified, the signature scan is performed only for documents in this set. The result of a signature scan is \( n \) sets of document LDIs: \( D_1, D_2, ..., D_n \). Each \( D_i \) is the set of LDIs of the documents that satisfy \( T_i \).

It should be noted that, since the hashing function is not invertible, each set \( D_i \) may also contain documents which do not satisfy \( T_i \). These are called false drops. Their relative frequency is controlled by adjusting the parameters F and m. In the MULTOS
implementation false drop rates in the order of 0.001 have been achieved. The above described organization of the signature file is sequential. An alternative organization which reduces I/O time drastically is the so called bit-sliced. For a comparison of the two, as well as for a combination of relevance ranking techniques with text signatures see [25]. Both bit-sliced signatures and ranking techniques are being tested in MULTOS.

Image access

The image retrieval handler has two access methods for image document components for the sake of subsequent comparative study of their performances. They are both offered through the same user interface, however they have to be explicitly selected. This is acceptable in an experimental system, yet the choice of access method should probably be transparent in a real application environment.

As mentioned in section 5.3.3, image query processing is performed on symbolic images which consist of objects, relations among objects and descriptions of object and image properties. Each application domain is, in general, characterised by an alphabet of basic objects and production rules whereby complex objects can be composed from the basic ones. The basic objects can be particular fixed arrangements of even more elementary objects, such as the primitives of a graphical editor, or the elementary objects themselves.

The two access methods employed in MULTOS are:

Method I [20]

A symbolic image is considered to be a triple

\[ P = (S, R, a_I) \]

where:

\( S \subseteq V \times A_0 \) the set of objects in the image,

\( V \) the vocabulary of objects and

\( A_0 = A_{01} \times A_{02} \times \cdots \times A_{0n} \) the object attributes;

\( R : S \times S \rightarrow R_P \) a mapping expressing the spatial relationships between objects,

\( R_P = \cup_{(a, b, SE)} \) denotes that \( b \) lies southeast of \( a \) and \( (a, b, :) \) that they are in the same position; and

\( a_I \in A_I = A_{I1} \times A_{I2} \times \cdots \times A_{Im} \) the global image attributes.

A fixed vocabulary of basic and complex objects is assumed. The production rules are supposed to be unavailable outside the graphical editor (graphical images) or the image understanding module (bitmap images). Thus, all objects are effectively treated as basic. Also, the mapping \( R \) only represents the relative positions of objects.

Encoded as explained further below, symbolic images are stored as image signatures in an image signature file with pointers to the documents containing the corresponding images. Query processing involves scanning the image signature file to match the signatures of the image query terms (duly generated) with those contained in the file. The image signature file is updated by appending new signatures.

An image signature consists of an image attribute vector, holding the global image attributes, and an attributed two-dimensional string. The latter encodes the objects, their relative positions and their attributes, and is an extension [20] of a data structure called two-dimensional (2-d) string [14], to include object attributes. A two-dimensional string
is a pair of strings representing the order relations of the projections of the objects contained in a symbolic image along two axes. Partial picture matching is obtained by two-dimensional substring matching at various levels of matching accuracy. Adding attributes to the objects gives rise to attributed two-dimensional strings which enable a combination of image access by object attributes with access by relative positions, giving priority to attribute conditions so as to restrict, as much as possible, the search space of the two-dimensional substring matching.

Assume that picture $Q$ (query) is matched against picture $P$. The objects $p$ in picture $P$ are first assigned to groups $G_q \subset P$ such that every object $p$ in $G_q$ matches $q$. A one-to-one correspondence is established between objects $q$ in picture $Q$ and groups $G_q$. Note that, since an object $p$ of $P$ may match more than one object $q$ of $Q$, the assignment of $p$ to $G_q$ is not unique and the groups $G_q$ constitute a cover, but not a partition of $P$. Now, an object $p$ matches an object $q$ if the two are of the same kind (same member of the image class vocabulary) and have compatible attribute names and attribute values. Assuming that a nonempty $G_q$ has been formed for each $q$ in $Q$, the 2-d string matching will proceed by searching for distinct representatives of each $G_q$ which have similar spatial relations with the respective $q$'s. This search is polynomial in the number of objects contained in pictures $P$ and $Q$.

Method 2 [23]

This method takes explicitly into account the production rules for composition of complex objects in an application domain and is based on the theory of evidence[8] for image interpretation and description.

The basic objects of an application domain are described in terms of elements (e.g., lines, curves) and their relative positions and attributes, using an attributed relational graph (ARG) representation [27]. An ARG consists of a set of nodes, usually representing basic elements, and a set of links representing relations between nodes. Properties of nodes or links are described by respectively assigned attributes.

An application domain definition thus consists of a set of ARG’s for basic objects and set of production rules. Both stored and query images undergo an analysis on the basis of application domain definitions, which comprises two phases:

(a) Low level image analysis for basic object recognition. Basic elements and their relative positions are recognized and the ARG of the image is constructed and compared to the ARG’s of the domain objects in order to determine corresponding similarity measures. This phase has polynomial complexity.

(b) High level image analysis for complex object recognition. A generalized inference mechanism using the production rules recursively is applied on the output of the previous phase to recognize complex objects with a certain belief and plausibility. The number of basic objects is presumed to be small (for performance reasons). Complex objects are inexactness recognized with a recognition degree measuring a "distance" between objects in the image and those in the domain definition. If the image contains $n$ recognized objects, its interpretation is given by the sequence

$$\left[ O_{11}[\mu_{11}], \ldots, O_{1s_1}[\mu_{1s_1}] \right], \ldots, \left[ O_{n1}[\mu_{n1}], \ldots, O_{ns_n}[\mu_{ns_n}] \right]$$

(1)

where $\left[ O_{i1}[\mu_{i1}], \ldots, O_{is_i}[\mu_{is_i}] \right]$ represents $s_i$ different recognitions of object $i$; $\mu_{ij}$ is
the degree of recognition of the j-th recognition of object i. A given object may be present several times in (1). Applying the theory of evidence [8], the different recognitions of the same object in an image are composed in one single recognition measure, given by the belief (b) that the object has been recognized, and on the plausibility (p) of such belief (0 ≤ b ≤ p ≤ 1)[23]. The final image interpretation is given by

\[
O_{11} \left[ b_{11}, p_{11} \right] \cdots O_{1q_1} \left[ b_{1q_1}, p_{1q_1} \right] \cdots O_{n1} \left[ b_{n1}, p_{n1} \right] \cdots O_{nq} \left[ b_{nq}, p_{nq} \right]
\]

where \( q_i ≤ s_i \).

Images may be grouped into clusters containing similar images. The final part of the High Level Image Analysis consists of adding to the image description the information of the membership degree of the image to every cluster.

After the Image Analysis, the image interpretation is used to generate access structures that are used in the retrieval phase. These access structures consists of indices on the objects contained in the image, with their attributes.
References


18. CHRISTODOULAKIS, S., THEODORIDOU, M., HO, J., AND PARTHIA, A., "Multimedia document presentation, information extraction and document formation in


