An Object-Oriented Multimedia Database System for a News-on-Demand Application

by

Chiradeep Vittal

Technical Report TR 95-06
June 1995

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

Contents

List of Figures

Abstract

Multimedia applications need support from an underlying multimedia storage system to store and retrieve multimedia objects. The presence of spatio-temporal and composition relationships between the objects, their large volume and their inherent distribution pose interesting modeling and implementation requirements. These requirements cannot be fully met by conventional means such as file servers and relational databases. The design and implementation of a multimedia database to satisfy these requirements is described in this thesis. The design is targeted to a News-on-Demand application. The features of this work are the use of object-oriented database technology, and the use of document standards to represent multimedia documents.

News-on-demand is a distributed multimedia application that uses broadband network services to deliver news articles to subscribers in the form of multimedia documents. A type system is developed to model the individual media components (monomedia) of the documents. To capture the composition and spatio-temporal relationships between the monomedia objects in the news articles, the SGML and HyTime document standards are used. This is done by designing an SGML/HyTime document type declaration (DTD) for multimedia news articles. This DTD is mapped to a type system. An *annotation* scheme ensures efficient storage of the text component of the document. These type systems are implemented on an object-oriented database system and fully satisfy the requirements of the news-on-demand application.

Chapter 1

Introduction

1.1 Motivation

Multimedia information systems integrate a variety of data types such as text, images, sound and video to enable different multimedia applications. These applications have several requirements, including the storage and retrieval of multimedia objects. The characteristics of multimedia data make the use of conventional storage systems (file systems or relational databases) inadequate or problematic. Some of these characteristics are:

1. Large volume. For example, each of the following takes 1 Mbytes of storage in uncompressed form [Fox91]: six seconds of CD-quality audio, a single 640x480 color image with 24 bits/pixel, single frame (1/30 second) CIF video, or one digital X-ray image (1024x1024) with 8 bits/pixel.

2. The presence of spatial, temporal and logical relationships between components of a multimedia object. Examples of spatio-temporal relationships include the synchronization of captioned video and audio. Multimedia data is often presented in the form of *documents*. Multimedia documents[1] are structured complex objects containing a number of primitive objects (*monomedia* objects) such as video, digitized voice and images. Document components have logical relationships between themselves; the composition relationships between chapters, sections and subsections in this document are examples of logical relationships. These relationships should be modeled explicitly as part of the stored data.

   For a database where such multimedia documents are stored, there should be facilities for (a) accessing objects based on their semantic contents, and (b) accessing different components of these objects. Furthermore, there are relationships among the multimedia objects (i.e., classification, specialization/generalization, and aggregation hierarchies) that need to be modeled [WKL86, DG92, Thur92].

3. Multimedia information systems require an extensible data model that allows application designers to define new types as part of the schema. For example, since monomedia objects are not just strings or integers or characters, new types have to be defined to handle them. Furthermore, the applications themselves must be able to add and delete new object types

[1] The terms 'composed multimedia data' and 'multimedia documents' are used interchangeably.

dynamically [Thur92]. Therefore, multimedia systems must not have static schemas and the storage system must be able to handle dynamic schema changes.

4. Distribution. The storage requirements of multimedia data and the development of gigabit networks and server technology has meant that multiple servers are used to meet the demands of applications such as news-on-demand [LG91, Stev91, GDC+92].

These characteristics make multimedia information systems excellent candidates for the use of database management system (DBMS) technology. However, most current multimedia information systems do not use DBMSs. This implies that system support for standard DBMS functions such as querying, update control through transactions, etc. is absent in these systems. As next generation multi-user systems are developed (such as news-on-demand, collaborative and interactive work and electronic publishing) multimedia DBMSs that provide native support for these functions will become more attractive.

Although several "multimedia file systems" have been implemented, file systems do not possess the full functionality of DBMSs. File systems leave to the user the responsibility of formatting the file for multimedia objects as well as the management of a large amount of data. Even if the multimedia data is stored in files, the spatial, temporal and logical relationships need to be stored in some DBMS. This has been the traditional role of DBMSs in multimedia information systems; the term "multimedia database" often refers to a centralized directory service for data stored in various file systems. Distributed DBMS technology [ÖV91] can be used to efficiently and transparently manage data distribution; distributed file systems are no match for distributed DBMSs in their functionality. The development of multimedia computing systems can benefit from traditional DBMS services such as data independence (data abstraction), high-level access through query languages, application neutrality (openness), controlled multi-user access (concurrency control), fault tolerance (transactions, recovery), and access control.

However, relational DBMSs are also not adequate to support the requirements of multimedia systems. Neither is there support for the new media types in relational systems nor is there a way to extend the type system to incorporate them (extended relational systems are an exception). The "binary large objects" (BLOBs) that are supported in some relational systems are not sufficient to model these entities. One can store an image, for example, as one BLOB, but it is not possible for the relational DBMS to interpret this BLOB (i.e., access parts of it or perform image-specific operations on it). Furthermore, it is difficult to represent the complex structure of multimedia documents, the spatio-temporal relationships between document elements and the classification and specialization/generalization relationships in the relational model.

As a central component of a distributed multimedia information system, a multimedia database should offer solutions to the storage and retrieval requirements posed by multimedia applications. News-on-Demand is a typical distributed multimedia application with the characteristics and requirements mentioned above. This thesis presents the design and implementation of a multimedia database for the news-on-demand application. The particulars of the design are the use of object-oriented database technology and the use of document standards to represent multimedia documents in the database.

1.2 Characteristics

1.2.1 Object-oriented Databases

Since relational databases fall short in their ability to support the storage and retrieval requirements of multimedia informations systems, alternate technology must be used. The emerging object-oriented DBMS technology [DÖBS94] is specifically targeted for "advanced" application domains such as multimedia information systems. Object-oriented DBMS offer solutions to the requirements of multimedia data mentioned above.

In particular, object-oriented systems provide an extensible data model, and can support dynamic schema changes. Object-oriented DBMSs[2], even though they may not provide native support for the new media types, can at least be extended to include them as part of the multimedia DBMS extensions. They can model the complex hierarchical structure of multimedia documents. In addition, the classification, specialization/generalization and aggregation relationships between multimedia objects have direct support in an object-oriented system.

There is also an ulterior motive in the use of OBMS technology. For years, research and development of OBMS technology has been motivated by the claim that it is best suited to meet the demands of "advanced" applications which include multimedia information systems. Unfortunately, reports of functional applications that use OBMS technology are scarce. The work on this thesis provides a test for this claim.

1.2.2 Document Standards

As indicated before, multimedia data is often presented in the form of documents. For example, in the news-on-demand application, information providers insert multimedia news articles in the form of documents into the multimedia database for subsequent retrieval by subscribers. To ensure a sufficiently open system, international standards for multimedia document representation are used. This is essential because the target application demands that a standard representation be used, for which various authoring tools are available. The tools themselves can be different, but they should at least be based on the same document representation. This is one way to support heterogeneity of tools while providing a unified database representation.

SGML [ISO86] and HyTime [ISO92] have been chosen as the standards to follow because of their suitability for the target application and their relative power. SGML has widespread use (for example, the Hypertext Markup Language, HTML, that is the basis of World Wide Web is an application of SGML) and is the basis of the HyTime hypermedia representation standard. SGML mostly deals with textual documents while HyTime adds support for hypermedia documents (e.g., links, video, etc.). The two other alternatives to follow would have been the Office Document Architecture (ODA) Standard [ISO89] and the MHEG Standard [Pric93]. ODA is not sufficiently rich (the standard defines formats only for text, and graphics) to be used in this application and the MHEG standard (even in draft form) was not yet released when this work was started. While SGML/HyTime is gaining acceptance and tools are being developed for it, MHEG is still in draft form. The SGML and HyTime standards are used by designing a document type declaration (DTD) for multimedia news documents. This DTD is used to encode (by means of markups) complex multimedia news articles. The DTD forms the basis of the design of the database.

[2]Henceforth, the term 'Objectbase Management System (OBMS)' will be used in place of 'Object-oriented database management system'.

1.3 Scope and Organization of the Thesis

At the core of this thesis is the design and implementation of a type system which can be used to store monomedia objects and SGML/HyTime-encoded news-on-demand documents containing these objects, in an object oriented database. The design satisfies the requirements of multimedia data, and addresses the following issues in particular:

- the modeling of monomedia objects,
- the modeling of document structure,
- the modeling of spatio-temporal constraints between document components and
- the modeling of the meta-information required by other system components of a multimedia information system.

The implementation of the design on a commercial object-oriented database and its integration with other components of a distributed multimedia information system validates the choice of OBMS technology and the use of document standards.

The rest of this thesis is organized as follows. Chapter 2 describes the application environment while highlighting the important characteristics of the target application. Chapter 2 also gives a running example of a multimedia news-on-demand document that is used in this thesis. Chapter 3 gives an overview of the SGML and HyTime standards. Chapter 4 presents the main contribution of the thesis and addresses the main design issues indicated. Chapter 5 describes details of the implementation of the design. Chapter 6 reviews related work reported in literature. Chapter 7 concludes with a discussion of the features and limitations of the design and implementation. Chapter 7 also presents future directions for the work.

Chapter 2

Application Environment

2.1 The News-on-Demand Application

News-on-Demand is an application which provides subscribers (or end users) of the service, access to multimedia documents. These documents are news articles that are inserted into a distributed database by news providers (or information sources). The news providers are commercial news gathering/compiling organizations such as wire services, television networks, and newspapers. The news items that they provide are annotated and organized into multimedia documents by the service providers (who may also be news providers). The subscribers access this multimedia database and retrieve news articles or portions of relevant news articles. This is typically a distributed service where clients access the articles over a broadband network from distributed servers (see Figure 2.1).

The above scenario for the News-on-Demand application brings up two issues:

- There are several news providers inserting documents into the database from different remote sites, over a network. Although the news providers may employ various proprietary formats to represent their articles, all articles have to be ultimately inserted into a database. This means that everybody has to follow a standard for news article representation and encoding to enable transmission over the network and insertion into the database. This implies that the representation of the document in the database should be architecture independent. There is a similar concern at the user's end, where different browsers and interfaces may be used to access the articles.

  The choice of SGML/HyTime as the standard for document representation is reflected in the overall organization of the News-on-Demand multimedia information system application (Figure 2.1). News providers compose hypermedia articles on their own authoring systems. These articles are then translated to the SGML/HyTime representation. An SGML/HyTime compiler checks the document being inserted against the document type declaration (DTD) which describes the acceptable document structure. It then instantiates the appropriate objects in the database. Subscribers use a querying interface to access articles and/or article components from the database, which can also be queried by various system components[1] to obtain relevant meta-information. This report deals with the database processing side of Figure 2.1.

[1]System components are described in Section 2.5.

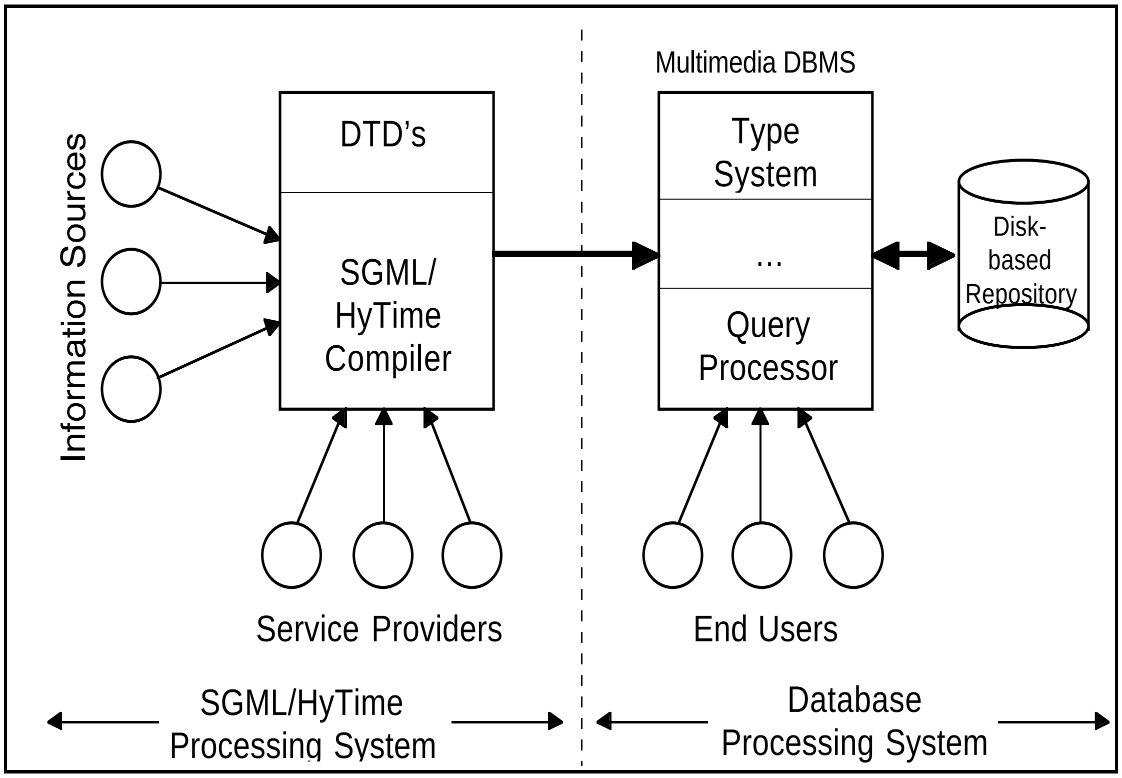


Figure 2.1: Processing Environment

- Once the news article is inserted into the database, it is not updated by either the news provider or the subscriber. Thus, there is a read-only model for the database. However, as time progresses the news provider may insert newer versions of the news article. The database management system would handle the version management issues.

2.2 Project Components

The work on this thesis is part of a larger project which focusses on researching broadband services for multimedia information systems. This multi-university project is supported by the Canadian Institute for Telecommunications Research (CITR) which is one of the Networks of Centres of Excellence funded by the Government of Canada. It was started in 1993 and has a five year duration.

In this component of the project, the focus is on the database issues for multimedia information systems. The other components are the QoS Negotiation component (Université de Montreal), the Synchronization component (University of Ottawa), the Continuous Media File Server (CMFS) (University of British Columbia) and Scalable Encoding for Video (INRS) components.

Within the multimedia database component of the project, two areas were investigated up to this point. The first is the modeling of multimedia data and the related database issues, and the second was the design of a visual query interface to the database described in [EM95].

2.3 Multimedia News Documents

A document is a *structured* collection of pieces of information related to a particular subject. In a multimedia document, these pieces of information are not restricted to conventional text, but include other media such as audio, video, and images. These media themselves may be composite, so that we may have combinations of audio and video, image and text, etc. These individual media objects are called *monomedia.* The structure of the document (i.e. the relationships between various document components) enables the contents of the document to be understood by the reader.

The structure is strictly hierarchical in nature, with the document itself at the root of the hierarchy. As an example, a book can be made up of chapters; chapters can consist of sections; sections consist of paragraphs, and so on. There is a distinction between the document content and the structure of the document.

Two types of structure can be identified: the logical structure and the presentation structure of the document. The logical structure refers to the logical organization of document components; the presentation structure refers to the layout of the components actually displayed to the reader. The logical structure of a book would be the organization into chapters, sections, paragraphs and so on; while the presentation structure has information on the number of columns of text used to display the document, the fonts and font sizes used to display the chapter titles, etc.

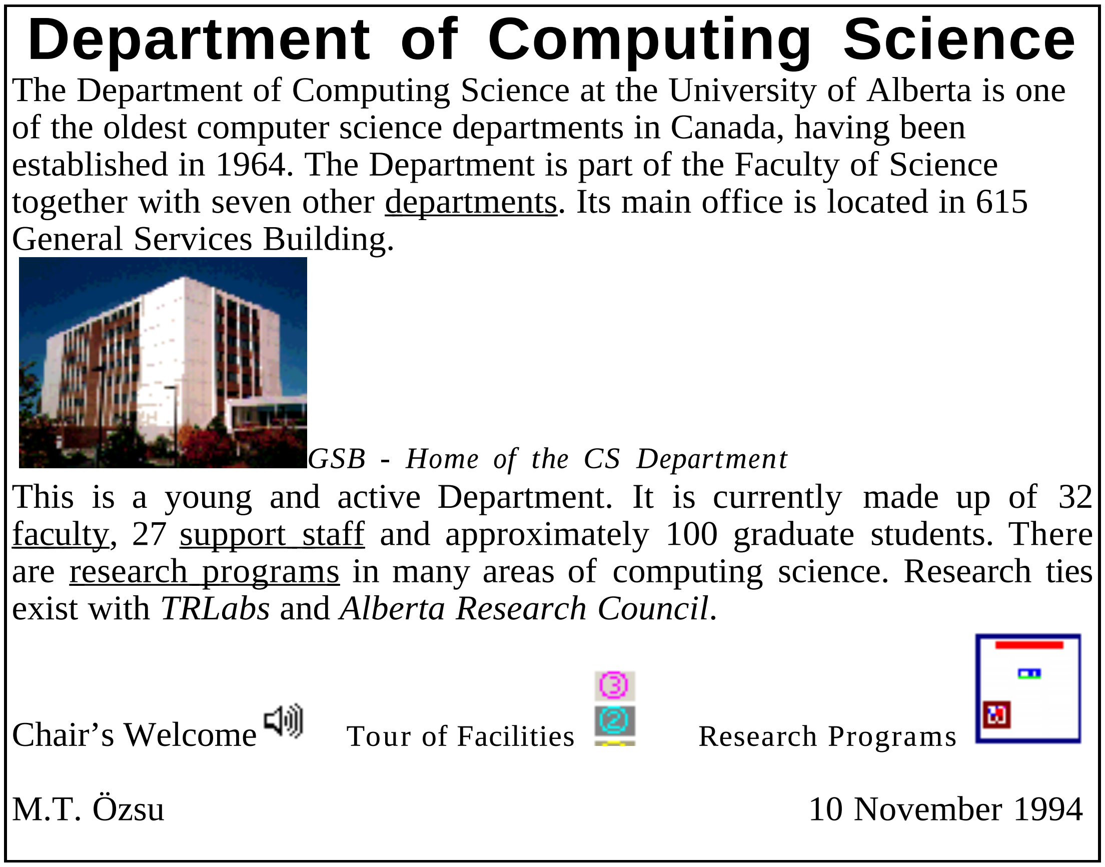
Documents often have links to other documents or document components. Common examples of such links in paper based documents are bibliographic references, footnotes and cross-references. Text overlaid with a link structure is called *hypertext.* In the case of multimedia documents, this term is changed to *hypermedia.*

2.4 A Sample Multimedia News Article

This section describes a sample multimedia news document that will be used as a running example throughout this report. This is an article about the Department of Computing Science at the University of Alberta. The article is organized as a series of news releases which are interlinked. The document components will be described in terms of the media present in the document; the full document is depicted in Figure 2.2.

- The text portion consists of the title, the (optional) subtitle, the keywords, an (optional) abstract paragraph, the date and location of the news release, the paragraphs that make up the article's content, the author, and the captions of any images appearing in the text. This information can also contain text that may not be shown in the presentation of the document, such as the keywords.

- The images in the document are any pictures related to the subject of the article. In this case, the picture of the building which houses the department is included in the document. The image can be stored in any format (GIF, TIFF, JPEG, etc.). The presentation of the image is also independent of the logical structure, because the user may choose to reproduce the image inline with the rest of the document, or display it in a separate window.



Department of Computing Science

The Department of Computing Science at the University of Alberta is one of the oldest computer science departments in Canada, having been established in 1964. The Department is part of the Faculty of Science together with seven other departments. Its main office is located in 615 General Services Building.

GSB - Home of the CS Department

This is a young and active Department. It is currently made up of 32 faculty, 27 support staff and approximately 100 graduate students. There are research programs in many areas of computing science. Research ties exist with *TRLabs* and *Alberta Research Council*.

Chair's Welcome   Tour of Facilities   Research Programs

M.T. Özsu   10 November 1994

Figure 2.2: Sample Multimedia Document

- The sound or audio component of the document is the recording of a welcome message from the Chair of the Department. Here again, the representation format is independent of the logical structure of the document. The tone and volume of the audio playback are examples of presentation attributes.

- The video component is a tour of the facilities. The representation format of the video data (MPEG, MJPEG, Quicktime, etc.), and the presentation aspects (frame rate, size of the window, etc.) may not be information relevant to the logical structure of the document. Video is seldom displayed on its own — there are associated media played back, or synchronized along with the video. Therefore, in the video clip about the facilities, the voice of the commentator is synchronized with the video so that the viewer does not find the displayed video out of phase with the sound of the voice being played back. There could be text subtitles displayed with the video and commentary, simultaneously giving the French translation of the commentary.

- The subscriber typically would like more information on the various events and people mentioned in the article that may not be found in the document itself. By providing links to other documents, or document components where further information can be found, this document enhances its information capacity. Another possibility is that the user may want to make comments (annotations) on the text that would be visible the next time the document is retrieved.

In Figure 2.2, the links to other documents are marked by underlined text. There could be other more obvious icons used to denote the link. This may depend on the preferences of the viewer, the type of terminal and the author's own choice. These are issues of presentation, and are not related to the logical structure of the document.

It is important to note that Figure 2.2 represents only one possible "rendition" of the news article. The user for example, may prefer not to see any text at all, or if the available display is an ASCII terminal, only the text portion may be presented, causing the system to skip the retrieval of the image, audio, and video components of the documents.

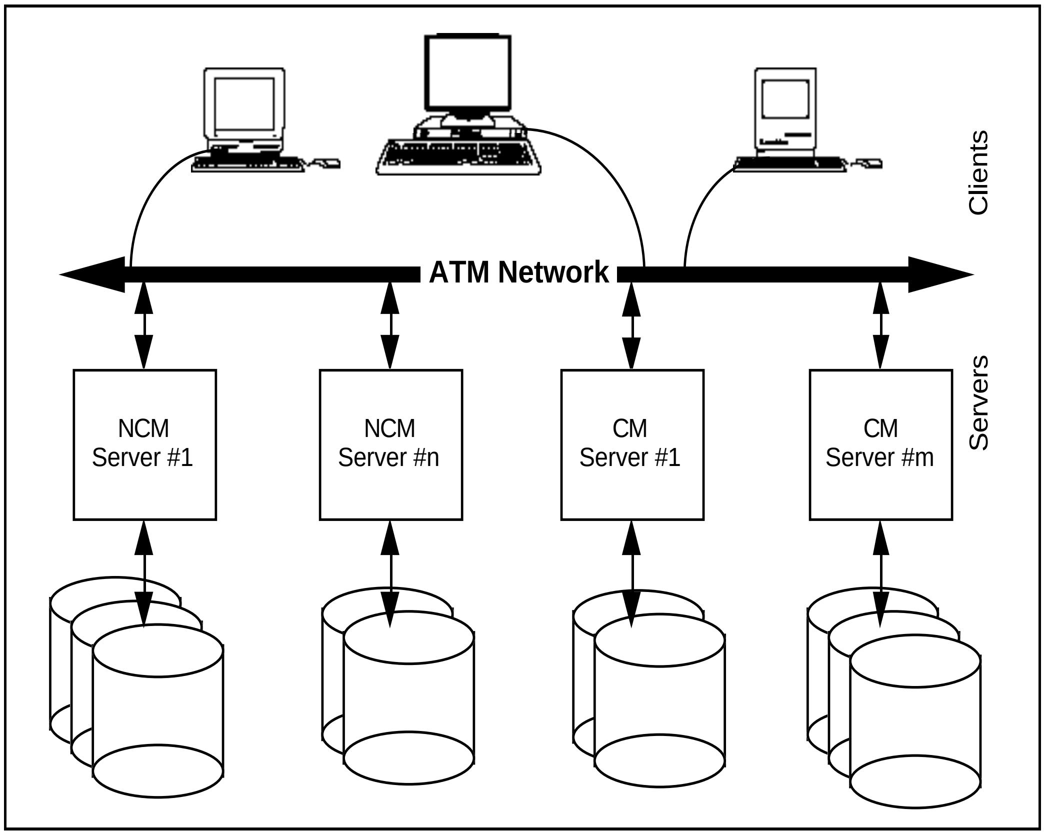


Figure 2.3: Distributed System Architecture

2.5 System Architecture

The stored multimedia data are classified as continuous media and non-continuous media. Continuous media refers to those types which have to be presented at a particular rate for a particular duration of time. These include audio and video. Non-continuous media such as text and still images do not have the real-time constraints of audio and video. Typically, continuous media are stored on a separate storage server which is designed to meet the real-time constraints of the data. In this system, continuous media and non-continuous media are stored on different servers.

2.5.1 Distributed System Architecture

Figure 2.3 shows the architecture of the distributed multimedia system with data distributed between a number of non-continuous media servers (NCM servers) and a number of continuous media servers (CM servers). The distribution of data is transparent to the users since they interact with a querying facility [EM95, ÖSEV95] at the client, rather than directly accessing individual servers. The client machines contain the query interface, the multimedia object-oriented DBMS client, synchronization modules, and the decoders for MPEG and Motion JPEG data streams. All accesses to the servers are routed through the client OBMS.

The current architecture however, does not integrate the continuous media servers with the database. Instead, the database stores meta-information about the files on the continuous media file server. The database is queried by other system components to determine the location of a particular piece of a multimedia object. After obtaining the file name and the server on which it resides on, the file is accessed directly from the file server. This architecture is necessary since the database system chosen for implementation of the application does not provide any native support for continuous media objects.

The retrieval of the document involves several system components, each of which must access the database to determine information necessary for the completion of its tasks.

Briefly, the user chooses a document to display on the client workstation after having browsed the database through a Visual Query Interface [EM95]. The user negotiates through the Quality of Service (QoS) Negotiator [HBB$^+$94] with the distributed system for the desired level of quality and cost of access. Then the Synchronization component [LG94] takes over by coordinating, over the network, the delivery of several streams of monomedia data contained in the document to the client. For continuous media, it has to request the Continuous Media File Server (CMFS) [NY94] to retrieve the appropriate files and start the streams. Non-continuous media components of the document are retrieved by the client OBMS.

The QoS negotiation module has to determine the media types in the document, their QoS parameters, the capabilities of the user's hardware and the bandwidth availability in order to perform its task. The synchronization component needs a representation of the temporal constraints between the media types in the document in order to determine the schedule of delivery of data over the network. The continuous media file server needs to know the names and locations of the files it is supposed to retrieve.

To determine the media types in the document and their quality of service parameters (cost, quality, delay, size, etc), the QoS negotiation module queries the database. The temporal constraints are represented in the structure of the document according to the HyTime standard. The database responds to a query by the Synchronization system component by translating this representation into the desired data structure. The database stores a unique file identifier assigned by the CMFS for each monomedia object it stores. The continuous media server uses this to determine the name and location of the file(s) containing the monomedia object.

There are four types of information. The types of information mentioned in the last paragraph are *meta-information* i.e. information about information. The meta-information is not part of the document content, and is not visible to the user. There is also *presentation information*, which is also not part of the document content, as explained in Section 2.4. Then there is the actual document consisting of the individual monomedia objects and, finally, the logical relationships between the components of a document.

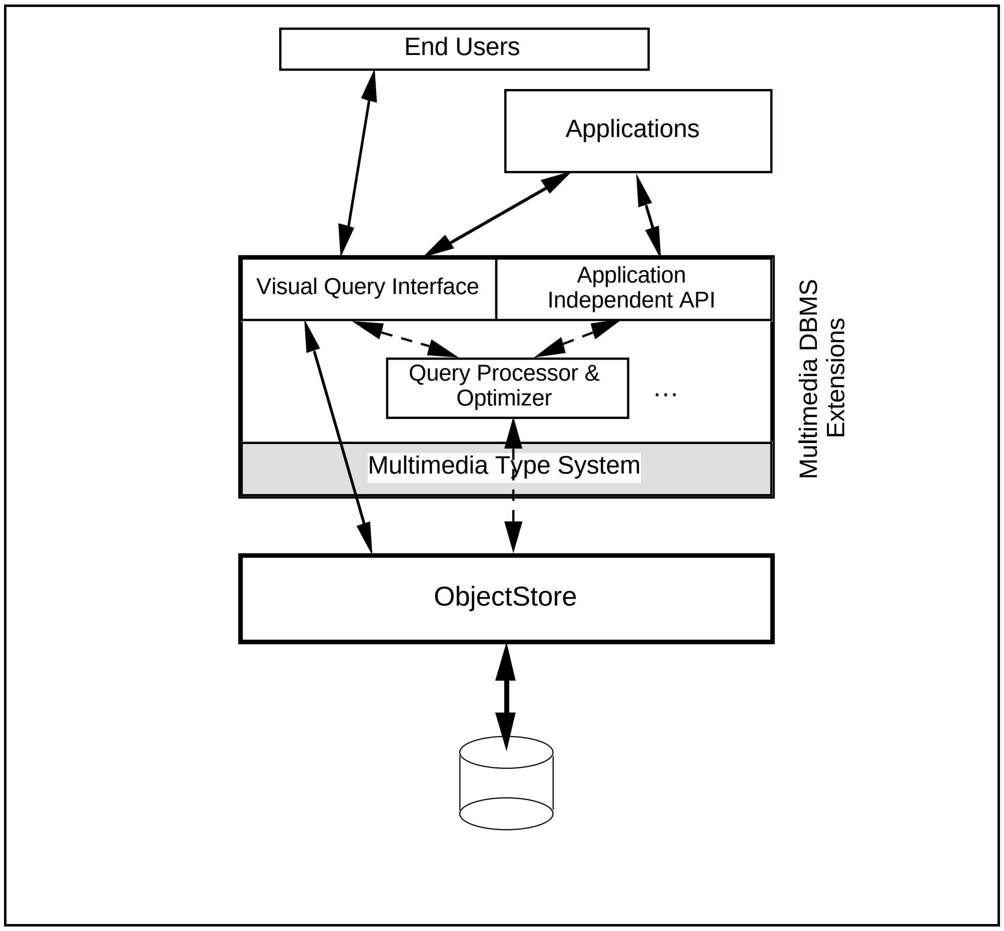

End Users
Applications
Visual Query Interface
Application Independent API
Query Processor & Optimizer
...
Multimedia Type System
Multimedia DBMS Extensions
ObjectStore

Figure 2.4: Conceptual Database Architecture

2.5.2 Database Architecture

The current prototype of the multimedia DBMS is an extension of a generic object-oriented DBMS called ObjectStore [LLOW91]. The extensions provided by the multimedia DBMS include specific support for multimedia information systems. The conceptual architecture, omitting many components not yet developed, is depicted in Figure 2.4. The development of a type system that supports common multimedia types is at the heart of the multimedia extensions. This research has focussed on this central issue as well as the development of a compatible visual query processing interface. These two components enable high-level modeling and access capability for application developers and end users. Future work, as discussed in Chapter 7, includes the development of an application-independent API and a more powerful query model that supports content-based queries of images and video, as well as an optimizer for these queries.

The fact that this work currently uses a generic object-oriented DBMS introduces some important restrictions. There is no native multimedia support and there is no access to source code. Therefore, the only way to extend the generic DBMS is to use standard object-oriented techniques to build a multimedia layer.

Currently, the visual query interface [EM95, ÖSEV95] interacts directly with the ObjectStore query processor via the multimedia type system. Each menu item is linked to an ObjectStore query which is invoked when the selection is made. As the application-specific query processor and optimizer development progresses, the visual query interface will interact with it rather than with the ObjectStore system. The new interaction is shown by a dashed line (Figure 2.4).

This architecture is open so that it can accommodate various multimedia servers. Many of these servers are file system servers without full database management functionality (e.g., querying). If file system servers are used, but the applications require database functionality, then a multimedia DBMS layer can be placed on top of the file system servers and the underlying storage system can be modified accordingly.

As indicated earlier, this is a distributed system where a number of clients access a number of servers over a broadband network. In the prototyping environment, the clients and servers are IBM RS6000/360 machines interconnected by a broadband ATM network. This is a multiple client/ multiple server system (Figure 2.3).

2.6 Query Examples

The larger database project aims at developing query languages, access primitives, and visual query facilities [ÖSEV95] that would allow sophisticated querying of these databases, including content-based querying of all types of media. While the work on the thesis is about logical modeling of multimedia databases, the work on the query interface deals with querying the database in greater detail. Since the type and frequency of the queries also affect the design of the logical model, a few sample queries are described here. As noted before, queries on the document are read-only in nature with no updates after the document is inserted. In the following, examples of queries on multimedia news articles are provided. A more detailed description is found in [EM95].

2.6.1 Retrieval Scenario

The following retrieval scenario elaborates on the type of queries the user and the system may perform.

- The retrieval process is started by a search on the article database by the user. As an example, the user wishes to see some articles about educational institutions. Alternatively, the request may be to view some articles featuring the University of Alberta. Therefore, the database is queried for all documents with the keyword *education* in them (or *University of Alberta*). Section 2.6.2 discusses searches in greater detail.

- As a result of the above query, the database returns a list of titles of articles with the required keywords. Along with the title, the user may also see an abstract paragraph of the article. Other information displayed could be the list of media types in the article, and the nominal cost of retrieval of the document. This cost changes as the user negotiates the quality of service desired (or that can be paid for) with the system. Note that each of these additional pieces of information is obtained through the user interface by querying over the documents in the list.
- The user then selects one particular article (for example, the one described in Section 2.4), and retrieves the document after negotiating the cost of access.
- The retrieval process itself triggers additional queries to fetch the necessary information for accessing and displaying the document. This includes fetching meta-information and presentation information as described in Section 2.5.
- The user can perform queries on the displayed document as well. Text string matching is a common example. Following the links within the document could trigger a new set of queries by the system to determine the meta-information associated with the new document.

2.6.2 Searches

Attribute Based Queries

An attribute based search is the most likely scenario, in which queries are predicated on news article attributes such as 'date' and 'title'. For example:

- select documents with the words 'Department' or 'Science' in their titles,
- select documents with the location 'Edmonton', published within the last year,
- select documents by authors whose names contain 'Smith'.
- select documents in the category 'Education' and having the keywords 'Research' and 'Databases', with a date later than January 1, 1995, etc.

Content Based Queries

Examples of these queries include "select documents with the string 'graduate studies' within the text of the article," "select documents with buildings in their images," and "select documents with video clips containing scenes of people walking." While searching text is possible and efficient, searching the contents of images and video still remains a major area of research. This work has so far concentrated on elaborate searching of textual parts of documents and provides means for accessing other monomedia objects by means of keywords.

Queries Based on the Structure of the Document

Since documents are structured, it is possible that queries based on the structure of the document will be posed [CACS94]. For example, the query could be: "select documents with video, but no text". It is also possible that the query retrieves only logical components of a document rather than the complete document. This leads to more complex queries. For example: "select the abstracts and first authors of articles having a section containing the words 'Research' and 'Canada'".

Queries can also be posed without a precise knowledge of the structure: "select all paragraphs in articles with titles containing the word 'Department' in them." Here, paragraphs could be subcomponents of more than one document element, say, section and abstract. The query does not specify whether the desired paragraphs should be subcomponents of `section`s or the `abstract`. The resulting set of paragraphs can be found by following these two different paths in the article composition hierarchy.

The HyTime standard describes a query language known as "HyQ" [ISO92, DD94]. HyQ is intended to handle the hierarchical structures of SGML documents. Although the utility of HyQ lies in its ability to locate SGML document components, content based queries are also possible in HyQ.

Queries which are combinations of the above types of queries are not excluded. For example: "select paragraphs from documents which have images containing buildings in them". However, as noted before, attribute based, and content based queries are considered to be the most likely scenario.

Chapter 3

Overview of SGML and HyTime

The logical structure of the document is necessary for the contents of a document to be understood. For example, document presentation, certain queries (Section 2.6) and hyperlinks, all rely on the logical structure of the document. SGML uses markups to represent this information. Other document representation formats use markups to represent both logical and presentation information (e.g., LaTeX uses `\section` to mark up sections, and `\bf` to present text in bold font).

SGML is intended for publishing in its broadest definition, ranging from conventional publishing to multimedia database publishing [ISO86]. HyTime [ISO92], uses SGML representation syntax; it is thus admirably suited to work with SGML documents. HyTime provides a standard way of representing links, and also defines their processing. In particular, it deals with structured information and the ability to link from and to structured information [DD94]. HyTime also provides a way to represent scheduling and rendering information.

One of the reasons for the success of the World Wide Web (WWW) is the use of the Hypertext Markup Language (HTML) standard for document representation. HTML is an *application* of SGML. The concept of the WWW is close to the idea of open hypermedia, in which links can be made to any piece of information including non-hypermedia representation. HyTime is an attempt at following open hypermedia [DD94].

3.1 SGML Principles

3.1.1 Markups and Logical Structure

Markups were traditionally used in document formatting programs to indicate processing instructions to the formatter. For example, before the beginning of each paragraph, there would be a markup indicating the amount of indentation for the first line of the paragraph, the number of blank lines to leave before starting the paragraph, and so on. This is known as *procedural markup* in which the presentation information is mixed with document structure and document content.

SGML is a *meta-language* which describes the logical structure of a document by using markups to mark the boundaries of its logical elements. The *generalized markup* approach of SGML separates the description of structure from the processing of the structure. The philosophy is that processing instructions can be bound to the logical element at the time of formatting, or display. Descriptive (or generalized) markup identifies logical elements using *start tags* and *end tags* to mark their boundaries. The elements are identified by their *generic identifiers* (GI), or tags. In the following example, the GI is `hdline`, and marks off the headline of the sample news document:

```
<hdline> Department of Computing Science </hdline>
```

The processing instruction which is stored separately, in this case could be, "set all **hdline** elements in 18pt bold Helvetica font."

The markup in SGML is *rigorous* [Gold90] in that elements can only contain other elements to form a hierarchy with constraints on the type of elements which can occur at a particular position in the hierarchy. Thus, `chapter` elements can only contain `title` and `section` elements; `section` elements can contain `paragraph` elements and so on. The hierarchy is a tree, and whole subtrees can be manipulated as one unit. In other words, an SGML document consists of instances of document elements arranged in a hierarchical structure.

3.1.2 Document Type Declaration

SGML does not specify what these elements should be, or what the hierarchy should look like. Instead, the list of elements types, and the relationships between them is expressed as a formal specification called a Document Type Declaration (DTD). A DTD is written in SGML by the document designer for each category of document being designed. For example, a DTD represents the HTML standard for hypertext documents. For the news-on-demand database, it is necessary to write a DTD for multimedia news articles, but there could be DTDs for books, letters, technical manuals etc.

A DTD specifies the element types, the hierarchical relationships between element types, and attributes associated with them. Attributes contain information that is not part of the document content. In the example multimedia news document of Figure 2.2, the following element types can be identified: article, headline, date, paragraph, figure, figure caption, emphasis, author, and link. Note that the article itself is considered an element and there may be other elements (e.g., `keywords`) that are not demonstrated in the rendition of Figure 2.2. Omitting the audio and video elements, the marked-up sample news document is:

```
<article>
<front>
<author> M.T. Ozsu </author>
<keywords> computer science, University of Alberta, education
</keywords>
<hdline> Department of Computing Science </hdline>
<date> 10 November 1994 </date>
<location> Edmonton, Alberta, Canada </location>
</front>
<body>
<paragraph> The Department of Computing Science at the University of
Alberta is one of the oldest computer science departments in Canada,
having been established in 1964. The Department is part of the
Faculty of Science together with seven other <link
linkend=sci_depts.sgml>departments</link>. Its main office is
```

```
located in 615 General Services Building.
</paragraph>
<figure filename=gsb.gif>
<figcaption>GSB - Home of the CS Department</figcaption>
</figure>
<paragraph> This is a young and active Department. It is currently
made up of 32 <link linkend=faculty.sgml>faculty</link>, 27 <link
linkend=faculty.sgml>support staff </link> and approximately 100
graduate students. There are research programs in many areas of
computing  science. Research ties exist with
<emphasis>TRLabs</emphasis> and  <emphasis>Alberta Research
Council</emphasis>.
</paragraph>
</body>
</article>
```

This document is declared to be an article type. Thus, the legality of its mark-up is determined according to the article DTD which defines the acceptable article document structure. The following is a segment of the article DTD:

```
<!ELEMENT article - - (front, body) >
<!ELEMENT front   - - (author, keywords, hdline, date, location)>
<!ELEMENT body    - - (paragraph|figure)* >
```

The first line of the DTD (known as an *element type declaration*) declares an `article` element as consisting of a `front` element and a `body` element. The `article` element is said to have a *content model* consisting of a `front` element and a `body` element. The connector ',' implies "followed by". Note that a document structure hierarchy is emerging here with the `article` element at the root with children elements `front` and `body`. This hierarchy is important when the type system is created to represent the document structure. Element `front` represents the front-matter of articles and, as specified in the second line of the DTD, must consist of `author, keywords, hdline, date, and location` elements. The document structure hierarchy should now be obvious. Finally, the last DTD line indicates that the body element consists of zero or more occurrences (symbol '*') of a `paragraph` or (symbol '|') a `figure` element in any sequence. Note that the example mark-up obeys the DTD fragment shown here except that definitions for paragraph and figure must be added. The example given here is a simplified form of the full DTD for the multimedia news articles that is given in Appendix A. An outline of the DTD development process is presented here.

3.1.3 DTD Design

Once the logical elements of the document are identified, their content models need to be defined. This will be done for the elements occurring on the right hand side in the DTD fragment above. The elements `keywords`, `hdline`, `date`, `emphasis`, and `author` do not have other elements within them; they only contain text strings. In SGML syntax, these strings are called `#PCDATA`. Since these elements have identical content models, only the left hand side of the element type declaration will differ. When all these declarations are combined into one :

```
<!ELEMENT (keywords|hdline|date|emphasis|author) - -  (#PCDATA) >
```

In the sample document, `paragraph` elements contain strings of text (`#PCDATA`) interspersed with `emphasis` and `link` elements. Both of these sub-elements can be optional. The figure element contains one sub-element, the figure caption element, `figcaption`. This element just contains text strings, or `#PCDATA`. It can be assumed that this sub-element to be always present. Using these inferences and assumptions :

```
<!ELEMENT paragraph  - - (#PCDATA | emphasis | link)+>
<!ELEMENT figure     - - (figcaption)>
<!ELEMENT figcaption - - (#PCDATA)>
```

The next step is to define attributes for the element types, if any. Looking back at the marked up document, `link` and `figure` elements have start tags with attribute values in them. Writing the attribute lists for link and figure elements in the DTD:

```
<!ATTLIST link
          linkend  CDATA  #REQUIRED>
<!ATTLIST figure
          filename CDATA  #REQUIRED>
```

The `CDATA` refers to the data type of the attribute (character data in this case). The `#REQUIRED` value means that any instance of the link element has to have a value associated with the linkend attribute[1].

To summarize, the DTD contains element type declarations and attribute lists. There are several optional features of SGML not mentioned here; they will not be used. The design of the DTD is based on our concept of a multimedia news document and the requirements of the application, since there isn't an extensive base of pre-existing documents to analyze. The complete DTD based on this concept is presented in Appendix A.

3.2 Why not HTML?

The complex DTD design process raises the obvious question: Aren't there DTDs already written for this purpose, for which standard tools and browsers are available? The HTML DTD for example, is a highly popular application of SGML, and is used to represent a wide range of documents for information transfer over the internet. There are two reasons why the HTML DTD was not used.

First, the HTML DTD has no support for specifying spatio-temporal constraints on the continuous media present in the document. This requires HyTime support as explained in the remainder of this chapter.

Second, the HTML DTD is designed to represent a wide range of documents. In a database however, it is advantageous to use application-specific DTDs. Application-specific DTDs identify elements with significant semantic importance in the application. For example, the HTML DTD does not have the logical elements `date`, `abstract`, `hdline` and `authors`.

If the news article was represented in the database using the HTML DTD, there would be no way of determining the authors of a document. Note however, that an easy translation to the

[1]Other options are `#IMPLIED` (attribute need not be instantiated), and `#FIXED` (attribute can be instantiated to only one value).

HTML standard can be achieved by mapping the elements `author`, `hdline`, `abstract`, `date` in the multimedia news DTD to the elements `heading1`, `address`, `blockquote`, `emphasis` respectively in the HTML DTD. Once this translation is done, a standard HTML browser can be used to view the multimedia news document.

3.3 HyTime Overview

3.3.1 Architectural Forms

The above discussion omitted any discussion of links, audio, and video objects. These are the domain of the HyTime standard. HyTime also uses DTDs to represent document categories. This leads us to the idea that one catch-all DTD could be defined for hypermedia documents which would allow us to represent links, temporal information, and other special needs of hypermedia documents. The DTD would contain element type declarations for these special elements needed by hypermedia documents (for example, `link` elements). The syntax defined by this DTD together with the semantics for the special elements, would be our hypermedia "language." A similar strategy is seen in the design of the HTML standard, for example. This approach is too restrictive — document designers would like to use their own names for these special elements, the semantics defined for the elements may be too basic to be useful for certain applications, etc.

Instead, the HyTime standard uses a *meta-DTD*. The meta-DTD defines several special hypermedia elements, called architectural forms (AF), that can be used in DTDs. For example, there is an architectural form called `clink` which defines a so-called contextual link. A contextual link is a link with an anchor rooted in a particular context, exactly like the links shown in the sample news document. To use architectural forms in HyTime document instances, element types are defined which *conform* to the specification of the architectural forms. Then instances of these conforming element types are used. If the **clink** architectural form (AF) were to be used as the basis for a link element in the news article DTD, the following declarations would exist in the DTD :

```
<!ELEMENT link      - - (#PCDATA)>
<!ATTLIST link
          HyTime        NAME     #FIXED "clink"
          linkend       CDATA    #REQUIRED>
```

The value of the `HyTime` attribute of `link` is fixed to `clink`. This informs the HyTime parser that the element is supposed to conform to the **clink** architectural form. To conform to an AF, an element declaration (or instance) must have the HyTime attribute set correctly, and also have the other attributes declared for the AF in the HyTime standard. For the **clink** AF, there is a linkend attribute declared in the standard; therefore the `link` element must also define that attribute in the DTD.

3.3.2 HyTime Modules

The HyTime standard is divided into modules, each of which describes a group of concepts and architectural forms. These modules are the base module, the measurement module, the location address module, the hyperlinks module, the scheduling module, and the rendition module. Each module may use certain features of other modules lower down in the hierarchy; thus the location address module does define AFs which are used in the rendition module. Each HyTime-compliant DTD must declare the names of the modules it requires.

In the DTD for multimedia news articles, certain features of the base module (as in all HyTime documents), some features of the location address module, some of the features of the hyperlinks module, and some of those of the scheduling module are used. The description of these modules is skipped, except for the scheduling and location modules. Concepts needed from other modules will be defined where required.

3.3.3 Finite Coordinate Spaces

To represent relatively simple spatial and temporal constraints between document elements, the finite coordinate space (FCS) architectural form defined in the scheduling module is used. This, in turn, requires features of the measurement and location modules. In the discussion that follows, several architectural forms will be used in the examples but not explained. It is hoped that the relevant ideas can be understood. The following convention is used: whenever an element type name appears with a 'my_' prefix in an example, then it conforms to the architectural name that follows the 'my_' prefix.

HyTime models space and time using axes of finite dimensions. A finite coordinate space is a set of such axes. All measurements are associated with axes. The units of measurement along axes are called quanta. There are various types of quanta defined in HyTime, besides the normal units of measurement — including characters, words, nodes in trees, etc.

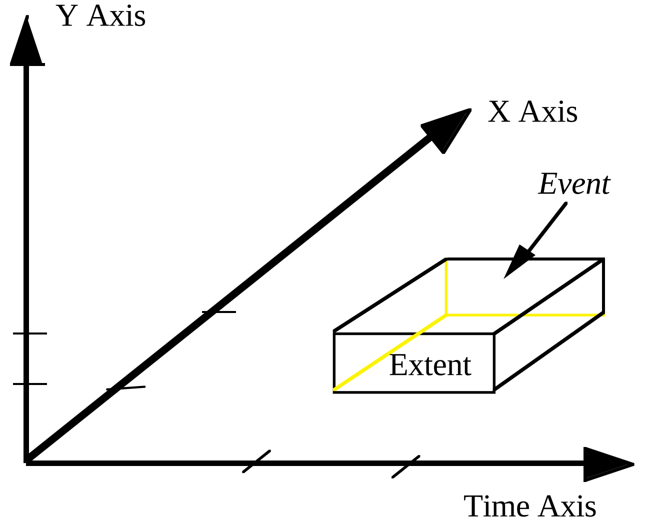


Figure 3.1: Axes, Events, and Extents (adapted from [DD94])

The following element declaration in a DTD defines a time axis conforming to the **axis** architectural form and having an addressable range from 1 to 100,000 seconds.

```
<!ELEMENT time - -  EMPTY >
<!ATTLIST time
         HyTime        NAME        #FIXED "axis"
         axismeas      CDATA       #FIXED "SISECOND"
         axisdim       CDATA       #FIXED "100000"
>
```

The quantum used in this example for axes measurement is the SI second. An FCS can be considered to be a Cartesian product of HyTime axes which is mapped to the real world space and time at the time of presentation/rendition. Figure 3.1 describes the various concepts used. The finite coordinate space shown here has three axes: two spatial, and one temporal.

In HyTime, an *extent* is a set of ranges along the various axes defining the FCS. An **event** corresponds to an extent in the FCS. An *event schedule* consists of one or more events. Extents are specified using the **extlist** architectural form. Events are created using the **event** AF; event schedules using the **evsched** architectural form. The document instance associates a data object with the event. The semantics and the manner in which the events are rendered can be defined by the application. The (meta) element type declarations used for these architectural forms are:

```
<!ELEMENT   axis    - -   EMPTY>
<!ELEMENT   fcs     - -   (evsched+)>
<!ELEMENT   evsched - -   (event+)>
<!ELEMENT   event   - -   (%HyBrid;)>
```

The '`%HyBrid;`' content model means that the content model is unrestricted. Any element, including non-HyTime elements can appear in the content model. Although the attribute lists are not given above, note that the **event** AF has an attribute called `exspec` which is of the type `IDREFS`. This means that this attribute gives the IDs of various HyTime elements conforming to the **extlist** architectural form. These **extlist** elements give the extents of the event along the axes of the FCS.

3.3.4 A DTD Fragment for Closed Captioned Video

In the sample document shown in Figure 2.2, there is an icon to indicate that there is a video presentation associated with the article. This could be, for example, the recording of a department tour, along with French subtitles displayed at intervals at the bottom edge of the screen. This is called closed captioned video (CC Video). The design of the DTD fragment corresponding to the CC Video concept is illustrated here to demonstrate the use of FCSs for storing spatio-temporal constraints.

In the CC Video document, there are three types of events, which roughly correspond to the three types of media present — audio, video, and (synchronized) text. Figure 3.2 shows only the time axis to display the extents of these events, for the first 65 seconds of the presentation. There are five events of type text (because the number of subtitles is assumed to be five, in the 65 seconds), and one each of the audio and video types. There are spatial extents also — two axes are created to represent the X and Y coordinates on the workstation screen (the time axis was presented in the previous section).

```
<!ELEMENT   X       - -   EMPTY>
<!ATTLIST   X
```

```
            HyTime        NAME      #FIXED "axis"
            axismeas      CDATA     #FIXED "virspace"
            axisdim       CDATA     #FIXED "1024" >
```

The DTD declaration for the Y axis is similar, except for the value of the axisdim attribute which is 900. The measurement units are in a HyTime defined unit called "virtual space," or **virspace**, which is used when there are no other pre-defined units available. In this case, the virspace corresponds to pixels on a workstation screen (assumed to be 1024 x 900).

As mentioned before, there are three types of events, which have extents along all three axes (although the audio event will not use the spatial axes). All three DTD entries are collapsed into one:

```
<!ELEMENT (audio|video|text) - -       EMPTY>
<!ATTLIST (audio|video|text)
            file    CDATA #REQUIRED
            -- HyTime Attributes--
            HyTime  NAME           #FIXED "event"
            exspec  IDREFS         #REQUIRED>
```

In this case, a file has been associated with each event. It could also have been a portion of a file, or an object in a database. The filename is given by the value of the file attribute in the element instance.

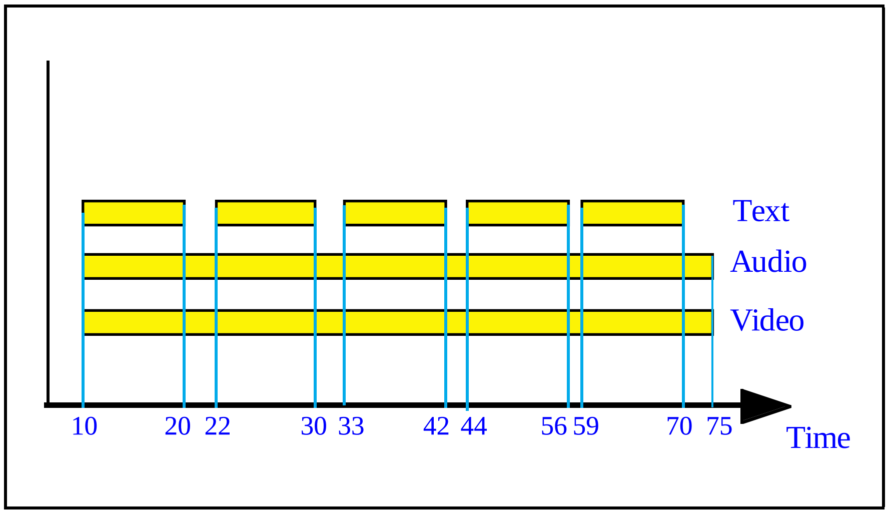


Figure 3.2: Extents Along the Time Axis for Events in CC Video

The event schedule which can represent the timeline shown in Figure 3.2, which consists of one audio, one video, and several text events can be defined as:

```
<!ELEMENT my_evsched - -  (video, audio, text+)>
```

For a complete DTD, including attribute lists, refer to the Appendix A. What remains is the declaration of the FCS:

```
<!ELEMENT my_fcs - - (my_evsched+)>
```

Finally, the CC Video document is declared (which is called audio-visual to make it more general) to be:

```
<!ELEMENT audio-visual - - (x,y,time,my_fcs,my_extlist+)>
```

The `my_extlist` element instances are used to specify the extents of the event instances.

The marked up document representing the schedule in Figure 3.2, conforming to the DTD fragment just written is:

```
<audio-visual>
  <x></x>
  <y></y>
  <time></time>
  <my_fcs>
    <my_evsched>
      <video file="tour.mpg", exspec=video_exspec>
      <audio file="welcome.au", exspec=audio_exspec>
      <text  file="subtitle1.txt", expspec=text1_exspec>
      <text  file="subtitle2.txt", expspec=text2_exspec>
      <text  file="subtitle2.txt", expspec=text2_exspec>
      <text  file="subtitle3.txt", expspec=text3_exspec>
      <text  file="subtitle4.txt", expspec=text4_exspec>
      <text  file="subtitle5.txt", expspec=text5_exspec>
    </my_evsched>
  </my_fcs>
  <my_extlist ID=video_exspec>10 63</my_extlist>
  <my_extlist ID=audio_exspec>10 63</my_extlist>
  <my_extlist ID=text1_exspec>10 10</my_extlist>
  <my_extlist ID=text2_exspec>22 8</my_extlist>
  <my_extlist ID=text3_exspec>33 9</my_extlist>
  <my_extlist ID=text4_exspec>44 12</my_extlist>
  <my_extlist ID=text5_exspec>59 11</my_extlist>
</audio-visual>
```

3.3.5 HyTime Locators

Links represent relations between locations in documents. In HyTime, data objects are located using the architectural forms (or locators) defined in the location address module. To achieve the flexibility required in hypermedia information system, there are several locators defined [DD94]. These locators can be classified according to the method used to locate the object.

Objects can be located by name (by assigning IDs to the objects), by counting (as in, "third element in the list"), and by querying ("paragraphs which are sub-elements of sections and contain the string 'Canada'"). Although all three can be reduced to queries, the distinction is maintained in the standard. An example of the second kind of locator is the **treeloc** locator which is used to locate data items in tree structured documents. A **treeloc** locator works by specifying how to navigate to the element from the root of the tree. For example, this is the second paragraph of the fifth subsection of the second section of the third chapter in this thesis. This would be written as[2] (here, `thesis_body` is the ID of the thesis body element):

```
<my_treeloc locsrc=thesis_body>
     <my_marklist>3 2 5 2</my_marklist>
</my_treeloc>
```

Treeloc locators are considered to be the simplest and most intuitive location method, in the absence of IDs [DD94]. Other locators are fairly complex. For locators which are queries, the HyQ query language is used. This is a type-less, LISP-like language. Although locators are not used in the multimedia news article DTD, they are mentioned here because, in the future, DTDs will be designed which use some of these features. A hypermedia database based on the HyTime standard needs to be able to support these locators indirectly or directly.

[2]This is not strictly true. Figures, lists, titles, and other elements which would increment the counts are neglected.

Chapter 4

Design of the Multimedia Type System

The design of the type system actually involves the conceptual design of the multimedia database. There are four issues in designing a multimedia database:

1. The different media components of the document (i.e., text, image, audio, and video) need to be modeled and stored in the database. These are called monomedia objects and their storage structures in the database is critical for good performance.

2. A representation is needed for the document's logical structure. Not every multimedia information system represents the document structure explicitly. For example, a multimedia system that uses postscript files for text documents containing images ignores the hierarchical structure of the document. It is important to represent this structure explicitly both for querying and for presentation.

3. In multimedia documents, one has to deal with the representation of the spatial and temporal relationships between monomedia objects. These relationships are important for presentation purposes - spatial relationships are used to model the placement of the various components on the screen while temporal relationships are essential for the synchronization of monomedia objects during presentation (e.g., audio synchronization with video or captioned text with video).

4. The meta and descriptive information necessary for the operation of the system components needs to be determined and stored in the database. As well, access routines need to be provided (as part of the API) for easy access to this information.

This chapter focusses on the first three issues which are central to the database design. As indicated earlier, an object-oriented approach is being used. The design of the type system aims for flexibility, portability and extensibility (to other document types and to other applications). The choice of the implementation medium (i.e., xlC/ObjectStore) is another influence on the design of the type system.

Types have *methods* and *attributes*. In the following, the motivations behind each type, the implementation of the types (in terms of attributes and methods) and the influence of the implementation on the design (if any), are discussed. When a method is said to be *declared*, it is not necessarily *defined* (implemented). However, every method in a type has to be defined to be able to instantiate it. Although the type hierarchy was arrived at with a mostly bottom-up approach (with iterations in the design), it is described in a top-down fashion.

4.1 Modeling of Monomedia Objects

The storage of continuous media such as audio and video is a challenging problem, particularly if content-based indexing of these media is considered. Since the continuous media file server is not yet integrated with the multimedia database, only descriptive information about audio and video objects is stored in the database.

As mentioned before, ObjectStore does not provide native support for multimedia data other than text (or strings). Instead, the multimedia DBMS that sits on top of ObjectStore implements these data types as atomic types.

4.1.1 The Type System for Atomic Types

Figure 4.1 illustrates the type hierarchy for atomic types. Instances of atomic types hold the raw (mono) media representation along with other information relevant to the QoS and synchronization system components.

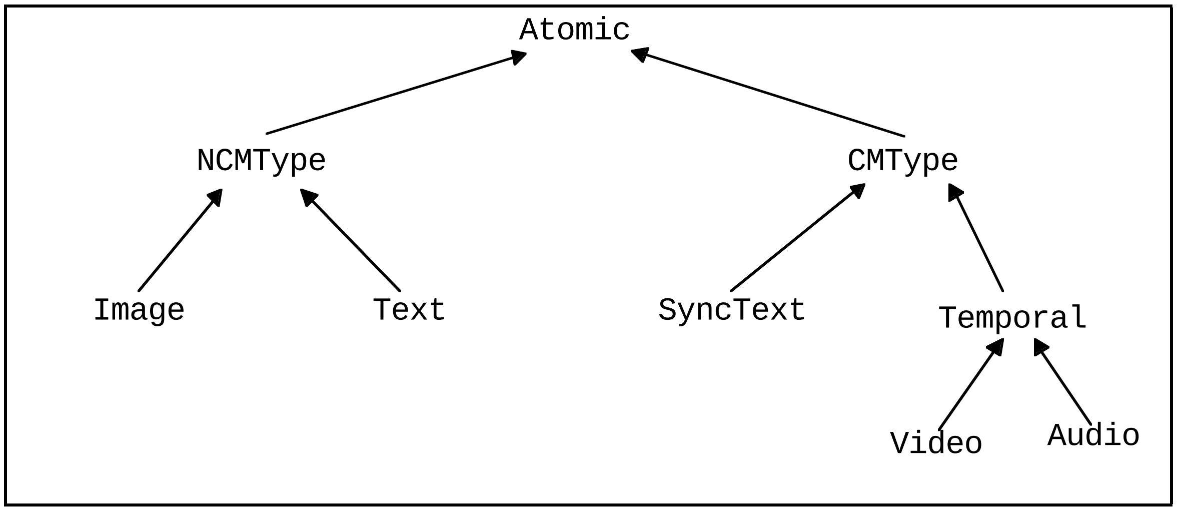


Figure 4.1: Atomic Types Hierarchy

There are two subtypes of type Atomic — one modeling non-continuous media (NCMType) and another modeling continuous media (CMType). The difference between the two types is that instances of NCMType actually store the raw media in the object. Instances of CMType only have meta information about the files which store continuous media. The attributes and methods which are common to both kinds of media are in the Atomic type. The attributes are the size and QoS parameters such as cost and delay [HBB+94].

From the database perspective, there are two kinds of QoS parameters. The first set consists of the intrinsic properties of the media themselves. For example, the width, height and color of an image instance are part of the QoS information required by the QoS negotiation component when accessing an Image instance. The second set of QoS parameters are those not directly related to the media, such as cost and delay. The values of these parameters however depend on the server (also called host) on which the monomedia is physically stored. These QoS parameters are grouped into a separate type HostQoS. For every unique server (host) and monomedia type combination, the values of the second set of QoS parameters are constant. A single instance of the HostQoS type stores these values for every such combination. This avoids duplication and maintains consistency of the QoS data. An added advantage of isolating these parameters is the insulation of the design from the changing requirements of the QoS Negotiator design group. Therefore, type Atomic has an attribute which is a reference to a HostQoS instance.

In the present state of implementation of the distributed multimedia information system, every system component assigns or requires an identifier for every logical monomedia instance. The type Atomic has an identifier attribute (attribute name), that acts as a logical identifier; multiple copies of the same logical object have the same identifier. Multiple copies do not always imply physical copies: for example, the same image can be stored in three different image formats. The issue of who assigns identifiers, and who manages multiple copies is a topic for research. Ideally, the distributed DBMS should handle identifiers and not the users. Currently, the DBMS does not have control over all the components of the distributed multimedia system. In the absence of any consensus, the use of a logical identifier is an ad-hoc solution to the problem.

The NCMType media are further subtyped into Text and Image media types. NCMType has the attribute content which is an array of characters. The Text subtype has additional methods: Match which implements a pattern matching algorithm, and Substring which returns a portion of the text object given the two integers representing the start and end locations. The Image type has additional attributes such as the width, height and color, which describe an image's intrinsic properties, independent of storage format. The Image type can be further subtyped to reflect the different storage formats possible.

A similar subtyping scheme is seen on the CMType side of the type hierarchy. The CMType type represents monomedia instances stored on the continuous media file server (CMFS). It has attributes storing meta-information such as filename, location and UOI (another identifier used by the CMFS). The location attribute is actually an instance of another type which has the host name, network address and directory location as attributes. Synchronized text (SyncText) is not subtyped from Text, since it is stored on the file system and not as an object in the database. The methods Match, and Substring cannot be applied to the synchronized text media. The Temporal supertype of Video and Audio is defined because both have a duration attribute. In addition to the width, height and color attributes, the Video has frame rate and bit rate as attributes. These properties are independent of storage format. Similarly, the type Audio has attributes number of channels, sample rate and bits per sample. Both types can be subtyped to represent the actual storage format.

4.1.2 Storage Model for Text

Text (a character string) is an atomic type which is supported in the database system. However, in the news documents, the text component of the article is richly structured, consisting of many hierarchically arranged components (also called *elements*). One alternative for representing text components of a multimedia document is to define object types for each of these structure components and associate with each of them a fragment of the complete text of the article. This is a direct mapping of the document type definition to the object oriented database schema.

Storing the text content of the article by fragmenting it in this manner can have serious performance implications. For example, to store the second instance of the paragraph element in the sample document of Figure 2.2, three fragments are needed — the emphasis element, the link element and the rest of the text. Accessing the text of the paragraph now involves three accesses to persistent store.

Although there are strategies such as clustering to improve performance, with large objects, these techniques may be inadequate. In any case, the pointer swizzling overhead of these objects cannot be overcome by clustering. Furthermore, if pattern-matching methods are defined on text elements, it would be necessary to reassemble the entire text component of the document. This has performance implications.

In addition to performance issues, there are modeling complications as well. One problem is to decide what the granularity of the fragmentation should be — paragraphs? sentences? words? The granularities can be determined by the granularities of the logical elements of the document. Thus, each logical element would contain a fragment of the text. For example, there would be an Emphasis type for instances of logical emphasis elements. This can cause several copies of the same piece of text to reside in various logical element instances. The second problem which arises is as follows: suppose an emphasis starts at some position in one word and runs until some position of a subsequent word (i.e., does not cover entire words). Since there is a logical emphasis element in the mark-up of this document, it would be necessary to create an instance of the Emphasis type and store the emphasized text as the value of one instance of this type. However, this precludes the possibility of querying for either one of those two words involved in the emphasized string.

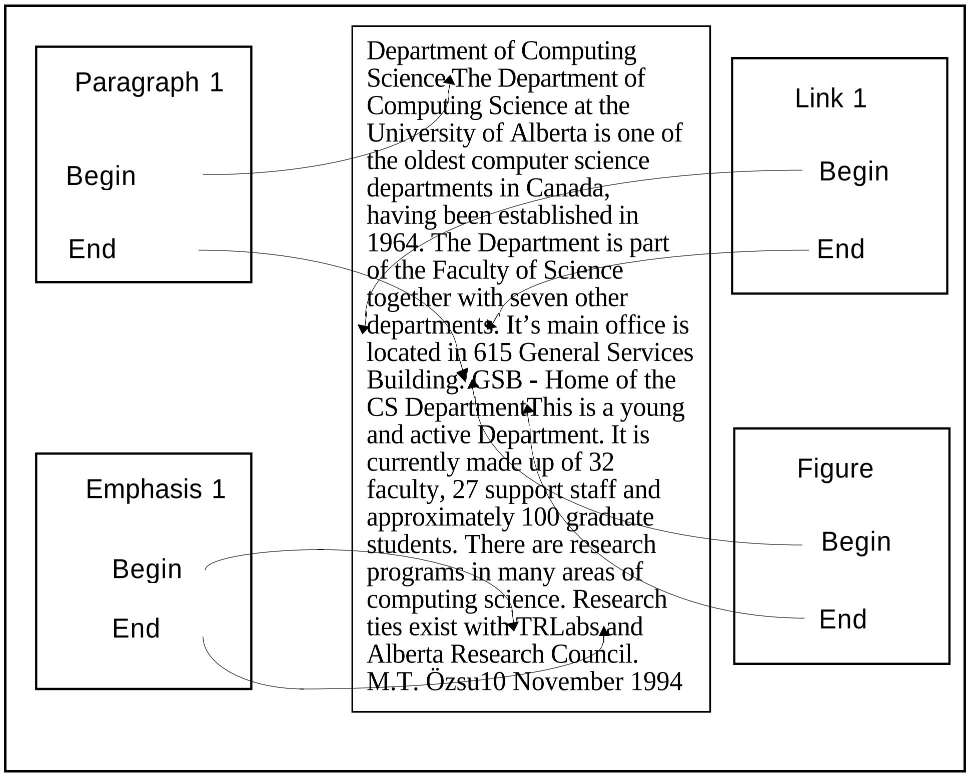
To avoid fragmenting the textual elements in this manner, the entire text content is stored as a single string. To associate a particular instance of an element with its text content the first and last character locations of that portion of text in the entire text content are stored. Pairs of integers such as these are called *annotations*. Using this model the text content of the sample news document can be modeled as depicted in Figure 4.2. In this example, the first paragraph instance has the annotation [33, 338]. The link sub-element of the paragraph has the annotation [264, 274].

Every document instance in the database has a "base" object (Article_root) associated with it which stores the text string forming the text content of the article, together with the lists of annotations associated with each text element type. To display the document, the browser can scan these lists efficiently and determine the presentation of the text. This representation is mapped to a type system by defining the type Text, whose instances store a single string that is the entire text content of a document as represented in Figure 4.2.

There are two distinct advantages of using this storage model for text elements:

- Displaying the complete document text becomes faster, and more efficient because multiple accesses to persistent store and joins are avoided.
- Indexes can be built on these annotation objects which can aid searches for element instances. For example, it is possible to search for emphasized strings in a document.

There is one disadvantage of this approach. Updates to the text content are expensive, since a change to the content of a document may cause many annotations to change. This can be avoided to a certain extent by specifying annotations relative to some enclosing structure, say with respect



Paragraph 1
Begin
End
Emphasis 1
Begin
End
Department of Computing Science. The Department of Computing Science at the University of Alberta is one of the oldest computer science departments in Canada, having been established in 1964. The Department is part of the Faculty of Science together with seven other departments. It's main office is located in 615 General Services Building. GSB - Home of the CS Department. This is a young and active Department. It is currently made up of 32 faculty, 27 support staff and approximately 100 graduate students. There are research programs in many areas of computing science. Research ties exist with TRLabs and Alberta Research Council. M.T. Özsu 10 November 1994
Link 1
Begin
End
Figure
Begin
End

Figure 4.2: Annotations to Mark-Up Text Documents

to a paragraph. Then, after an edit, the only annotations that change are the annotations of the sub-elements in the edited paragraph and the annotations of all following paragraphs but not the annotations for the sub-elements of these paragraphs.

4.2 Type System for Elements

4.2.1 Design of the Type System

General Approach and Conventions

Figures 4.3, 4.4 and 4.5 show the type hierarchies for logical document elements. The general characteristics of the design are:

- Each logical element type in the DTD is represented by a type in the type system[1]. For example, the `paragraph` element is represented by the type Paragraph. It can be argued that the textual element types have no semantic significance attached to them by either the DTD or the SGML standard [BAH93]. That is, there is no difference between the operations applied to a `title` element and those applied to a `paragraph` element. There could be just one type, type Element, with an attribute whose value would indicate the type of the element. However, the markup in SGML is *generalized*. Elements are chosen by the DTD designer because each element has a different semantic significance in the document. This would be especially true of elements conforming to the HyTime standard since the standard assigns semantics to the element. In addition, the *rigorous* nature of the markup means that the DTD constraints need to be maintained. This can be done by ensuring as much static type checking as possible. This is explained in the next item.

- To model the hierarchical structure, any type whose instance occurs in a non-leaf node in the hierarchy has attribute(s) which are reference(s) to child instances. For example, the `frontmatter` DTD element has the content model:

  ```
  <!ELEMENT   frontmatter - -   (edinfo, hdline, subhdline, abs-p)>
  ```

  Therefore, the type Frontmatter has attributes whose types are references to instances of Edinfo, Hdline, Subhdline, and Abs_p types. This design ensures as much static type checking as possible. For example, an attempt to make a Section instance as a child element of a Frontmatter instance would be disallowed at compile time. If there was just one type to represent all elements, then this checking would have to be done at run-time.

- Any SGML attribute defined in the DTD for a particular element (including those defined by the HyTime standard) is represented by an attribute in the representative type. The attributes themselves are always string-valued. Therefore, new types for the attributes are not defined. The types may have methods which access the values of these attributes.

The general convention is that a method beginning with the prefix Get is an access method. Methods which modify the state of the object are not considered since a read-only application environment is assumed.

4.2.2 Top Level Hierarchy

The supertype of all element types is the Element type. This models the is-a relationship — every element instance is an Element instance. As the document instance has a tree structure, the operations common to all element instances are tree operations. For example, HyTime specifies architectural forms for navigating the tree (described in Section 3.3.5). Certain queries (Section 2.6) also involve navigation of the document structure.

The frequently used operations in the navigation are (a) get the parent element, (b) get the nth child element and (c) get either sibling. It is possible to achieve the last operation with a combination of the first two. There are operations and queries common to tree structured data which are not

[1]The only exception is the type Pcdata. Although not a logical element, it is considered to be a *pseudo-element* in the HyTime standard. For this reason, a type is declared for it, and instantiated whenever it occurs in an element with a mixed (logical elements mixed with plain character data) content model.

considered here [SLVZ95]. A detailed analysis on document queries would reveal the salient set of tree operations required [BAH93]. In the absence of such a detailed analysis, the two representative methods: GetNthChild and GetParent[2] are considered.

As these methods are independent of the type of the element instance, the declarations of these two methods could be in the Element type. However, leaf nodes in the document tree never have child elements. All those element types in the DTD which have as content models `EMPTY`, or `(#PCDATA)`, are always leaf nodes in the document tree. In other words, there are two types of elements in any DTD: *structured* (non-leaf nodes) and *unstructured* (leaf nodes) elements. Therefore, type Element only has the method GetParent; type Structured, which is subtyped from Element has the method GetNthChild.

Links can be made to arbitrary elements in different documents. Searches performed over several documents can also return arbitrary element instances. It is often useful to know the article these element instances which they belong to. An example query is, "select articles which have any `section` element containing the word "SGML"." This query could be broken down into two steps: first, the collection of Section instances is searched, and then their articles are extracted. Hence, the method GetArticle is added to the type Element[3].

Type Element is subtyped into TextElement, Structured and HyElement. In the DTD for news documents given in Appendix A, the document is divided into components called `async` and `sync`. This is because of the fact that continuous media with synchronization constraints (the `sync` part) need to be handled by HyTime conforming element types; other (TextElement) ordinary SGML element types are adequate to deal with text and image data. The supertype HyElement represents all the HyTime elements used in the DTD.

4.2.3 Text Elements

Type TextElement models textual DTD elements. Every textual element instance has an annotation instance associated with it. The method to access this is the GetAnnotation method. To obtain the actual character string which the text element instance represents, the method GetString is defined. The TextElement type hierarchy (excluding StructuredText which is described later) is illustrated in Figure 4.4.

The types in Figure 4.4 correspond to text elements that do not have any sub-elements. Most of the types here do not have any additional methods other than those present in TextElement; they have been created as subtypes for classification purposes. This maintains the uniformity of the approach of modeling all element types in the DTD as types in the type system. The exceptions are the Quote, and the Author which have the methods GetSource, and GetDesignation respectively. This is because the DTD elements `quote` and `author` have the SGML attributes `source` (the source of the quote) and `designation` (the designation of the reporter/author in the news organization) respectively.

4.2.4 Structured Text Elements

Textual elements that are structured have a common supertype StructuredText (Figures 4.3 and 4.5). Type StructuredText is a subtype of types Structured and TextElement. The subtypes of this type

[2] These two methods are essential for the HyTime architectural form `treeloc` which is used as a locator. Navigating the structure is also possible by following pointers to child and parent elements: we just don't declare every possible method for every possible navigation.

[3] GetDocument would have been a better name, since there could be other document types in the database.

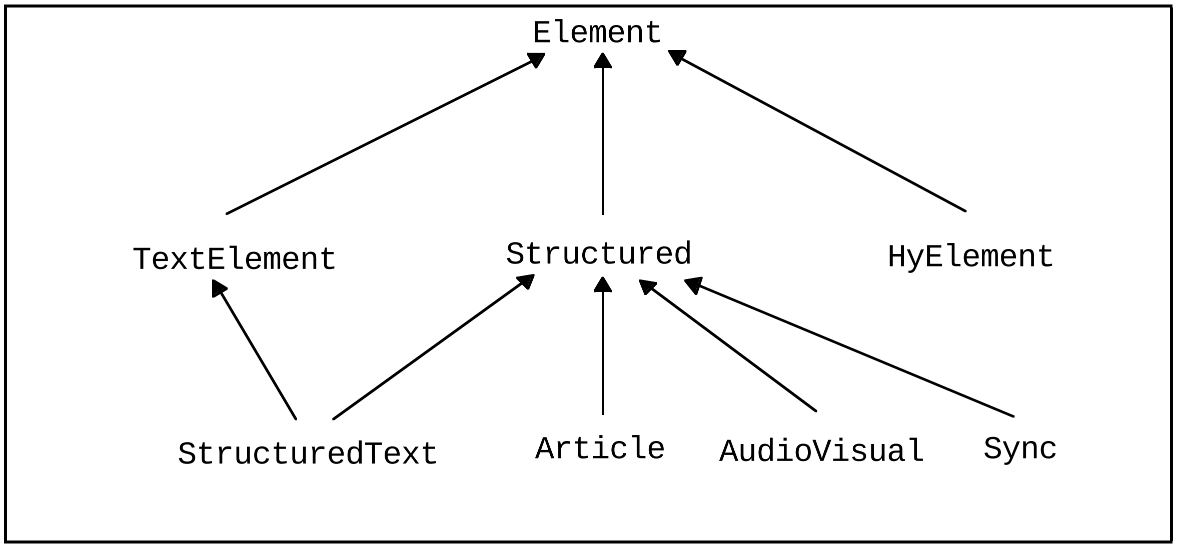


Figure 4.3: First-Level Element Type Hierarchy

include all types representing text elements with complex content models (`list`, `section`, `figure`, `frontmatter`, etc.). Except for type Figure, which has methods GetImage and GetFormat, all other subtypes of StructuredText do not have additional methods declared.

4.2.5 HyTime Elements

The type HyElement in Figure 4.6 is the supertype for all HyTime elements in the type system. All HyTime conforming element types in the DTD have `ID` and `HyTime` attributes. The values of these attributes provide an identifier and the name of the architectural form to which the element conforms. Type HyElement has two methods to access these values: GetID and GetAForm, respectively. The sub-hierarchy rooted at HyElement is depicted in Figure 4.6. Its immediate subtypes are those modeling the architectural forms used in the DTD. The HyTime standard defines semantics and attributes for each architectural form.

There are nine HyTime architectural forms used in the DTD. From Chapter 3, architectural forms are assigned (a) attributes, and (b) meta-content models by the HyTime standard. Types representing architectural forms have attributes modeling the HyTime attributes. The meta-content model determines whether the type is subtyped from Structured, TextElement, or StructuredText in addition to HyElement. If the meta-content model is '`%HyBrid`', then the type is only subtyped from HyElement. However, the actual conforming element cannot leave the content model as '`%HyBrid`'. If the content model is `EMPTY`, then the type is only subtyped from HyElement.

The most significant AFs from the database perspective are the **fcs** and the **ilink** AFs. Recall that the **ilink** elements model hyperlinks with more than one destination or source. This AF has a '`%HyBrid;`' content model; therefore, it could be a Structured element depending upon the content model defined by the DTD designer. A type for this AF is created, called Ilink_AF, as a subtype of the HyElement type. In the DTD for news articles, the `link` element has a complex content model

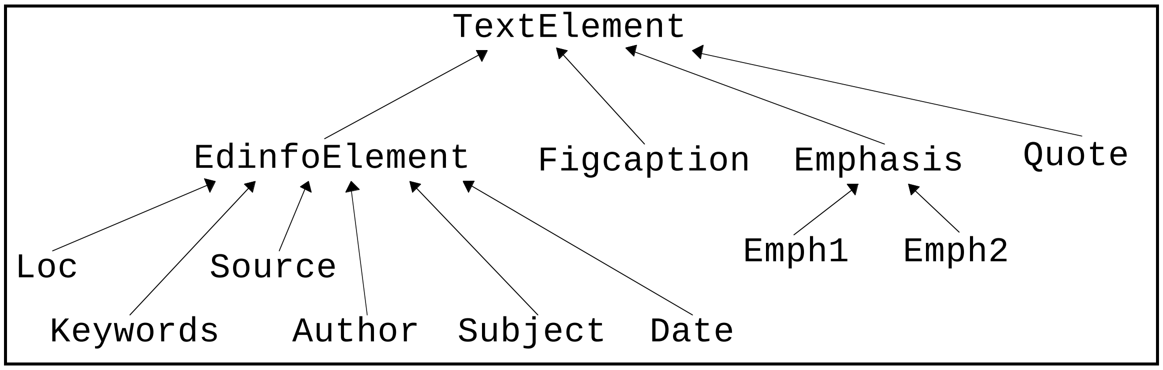


Figure 4.4: Type Hierarchy for Unstructured Text Elements

containing only text elements and conforms to the **ilink** AF. Therefore, the Link type is a subtype of both Ilink_AF and Structured. The Ilink type declares the method Traverse which takes the object ID (a reference to an Element instance) of a destination element and performs a traversal according to the applications semantics. This method is defined in the Link subtype.

The **fcs** element is important because it provides the interface to the other system components. This allows the types of media objects present in the continuous media, as well as the presentation schedule of the media objects which are a part of the FCS, to be determined. The **fcs** AF is represented by the type Fcs_AF. The actual type used in the document is the Av_fcs type. In the composition hierarchy of a multimedia news article instance, all the continuous media objects are descendants of the Av_fcs instance. Therefore, the Av_fcs type is entrusted with the interfaces to other system components.

As explained in Section 2.5, the synchronization component requires a representation of the temporal constraints between the monomedia (atomic type) instances in the document instance. The QoS negotiation component requires the list of monomedia instances in the document to be able to negotiate their quality of access. The method GetSchedule of type Av_fcs returns an instance of a data structure called scenario_str. This is a data type defined by the synchronization component design group. Its instances contain a representation of the temporal constraints. The playback schedule for the Atomic instances in the FCS is derived from this representation. The methods GetVideoObjects, GetAudioObjects and GetSyncTextObjects return lists of references to objects of type Video, Audio and SyncText respectively (which are atomic types).

Instances of the Article type are at the root of the composition hierarchy. The `article` element has to conform to the **hydoc** architectural form. Hence, type Article is a subtype of HyDoc_AF. According to the DTD, Article instances should have references to instances of Frontmatter, Async and Sync types. This means that Article is also a Structured type. In addition, the date, source, subject, and author are attributes (type String) of Article, even though these values are already stored (by means of annotations) as instances of Date, Source, Subject, and Author types respectively (Figure 4.4). This replication is done for performance reasons explained in the next chapter.

The other HyTime types (Figure 4.6) are architectural forms and the conforming elements used in the DTD. The **axis** architectural form has a '`%HyBrid;`' meta-content model. The actual conforming axes used in the DTD (`x`, `y`, and `time`) have `EMPTY` content models. Hence, type Axis_AF does not have Structured as a supertype. The three document axes declared in the DTD have their representative

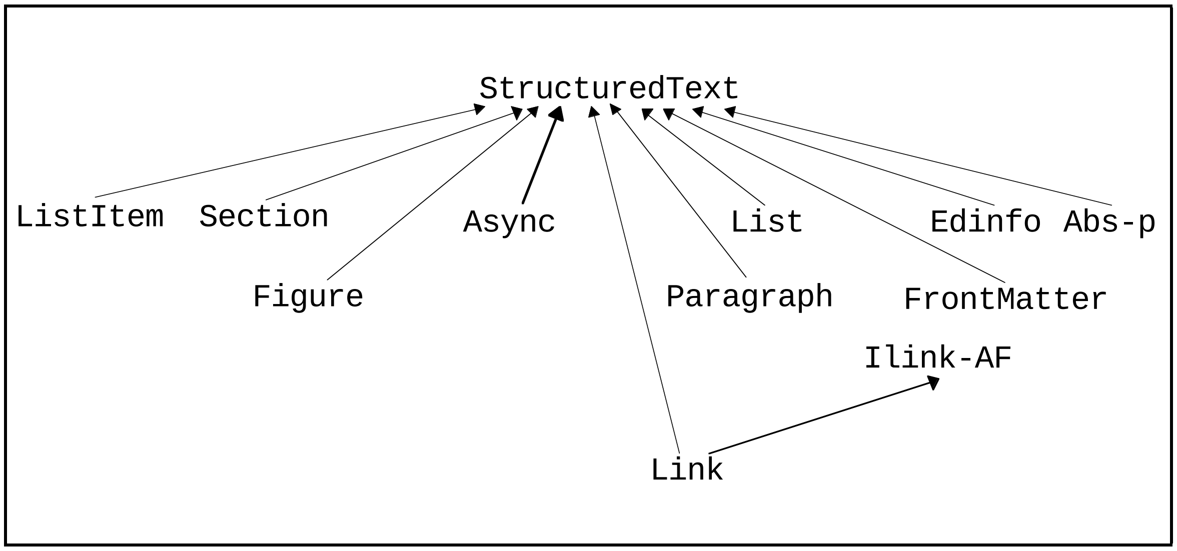


Figure 4.5: Type System for Structured Text Elements

types (X, Y, and Time) subtyped from Axis_AF. They do not declare additional methods or attributes. However, they have different semantics in the DTD. The values of the attributes for the dimensions, measurement units, and measurement granularity (axisdim, axismeas, and axismdu) are different for each axis.

The **event** architectural form also has a '%HyBrid;' meta-content model in the HyTime standard, but the events in the DTD all have EMPTY content models and hence, type Event_AF representing this architectural form is not a subtype of type Structured. The Event_AF type has been subtyped to represent the three different types of events possible in the finite coordinate space — text, video and audio (SText, SVideo, and SAudio). The intermediate supertypes Spatial and Spatio-Temporal model the temporal dimension of SAudio, while SVideo and SText have both spatial and temporal dimensions. These types have attributes which reference the atomic type instances which store the media associated with these objects. For instance, an SText type instance will have a reference to an instance of SyncText. The exspec attributes have references to the Extlist instances which hold the values of the extents of these elements along the three axes.

The **extlist** architectural form is represented in the DTD by the element type av-extlist. The content model of this element type is defined to consist of one instance of xdimspec, one instance of type ydimspec, and one instance of tdimspec. These elements represent extents on the three different axes (x, y and time respectively). They conform to the **dimspec** architectural form. This AF is represented in the type system by type Dimspec_AF. The three subtypes of Dimspec_AF are XDimspec, YDimspec, and TDimspec, which model the respective DTD elements. The content models of the three DTD elements are identical — they consist of two integers, representing the start and length of the extent. Hence, av-extlist instances are not leaf nodes in the document composition hierarchy. Therefore, the Av_extlist type is a subtype of the Structured type.

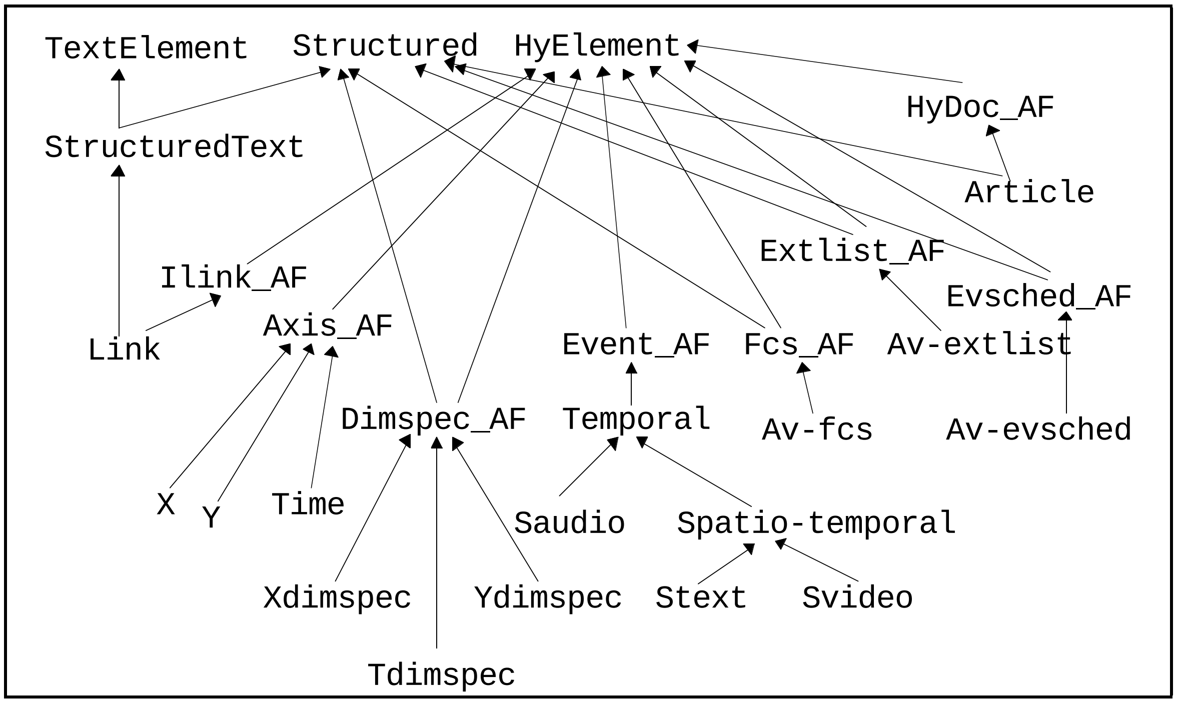


Figure 4.6: Type Hierarchy for HyTime Elements

4.2.6 Other Types

AudioVisual and Sync are the remaining subtypes of Structured which have not been described. In the DTD, the element `audio-visual` models one set of logically related HyTime components. For instance, if the document was one hour of a television broadcast, there would be one `audio-visual` each for the news, the commercial segments, etc. The whole broadcast would be modeled by the `sync` element. Sync instances hence have collections of AudioVisual instances as child elements.

Chapter 5

Implementation Issues

The design presented in Chapter 4 is general enough to be implemented on most object-oriented systems. The implementation, described here, is constrained by the implementation environment. These constraints, in turn, affect the design of the type system. In the following, the composition hierarchy which emerges when the sample multimedia document shown in Figure 2.2 is inserted in the database, is described. Then the constraints and the design decisions taken in their context are described.

5.1 Example Design

In this section, the sample document (Figure 2.2) is used to demonstrate how the type system can be exercised. This discussion concentrates on the composition hierarchy that emerges among objects according to the document structure. The composition hierarchy is based on the attributes of each type. Instead of presenting the attributes abstractly, it will be demonstrated how the structure of the sample document is mapped to a composition hierarchy as objects are instantiated and their attribute values set. This discussion refers to Figures 5.1 and 5.2, where object instances of type X are denoted as *MyX* and the arrows are from objects to their component objects.

The root of the composition hierarchy (Figure 5.1) is one instance of the Article type object, called *MyArticle*. *MyArticle* has three attributes, among others, that point to a Frontmatter type object, called *MyFrontmatter*, an Async type object, called *MyAsync*, and a Sync type object, called *MySync*. *MyFrontmatter*, holds the information in the document that is delimited by the markup `<front>` and `</front>`. As discussed in Section 4.2.1, the body of the document is separated into an asynchronous part (*MyAsync*) and a synchronous part (*MySync*). The asynchronous part describes the text and image part of the document.

According to the DTD of Appendix A, each document is separated into sections first. In this example, it is assumed that the figure which consist of the building's picture and the text before it is one section (even though it is only one paragraph) and the part after the figure is a second section. Thus, there are two Section type objects (*MySection-1* and *MySection-2*), as well as one Figure type object *MyFigure*, which are components of *MyAsync*.

The rest of the hierarchy should be obvious. Note that there are composition paths from some of these objects to instances of atomic types (Figure 4.1). For example, *MyFigure* has a link to an

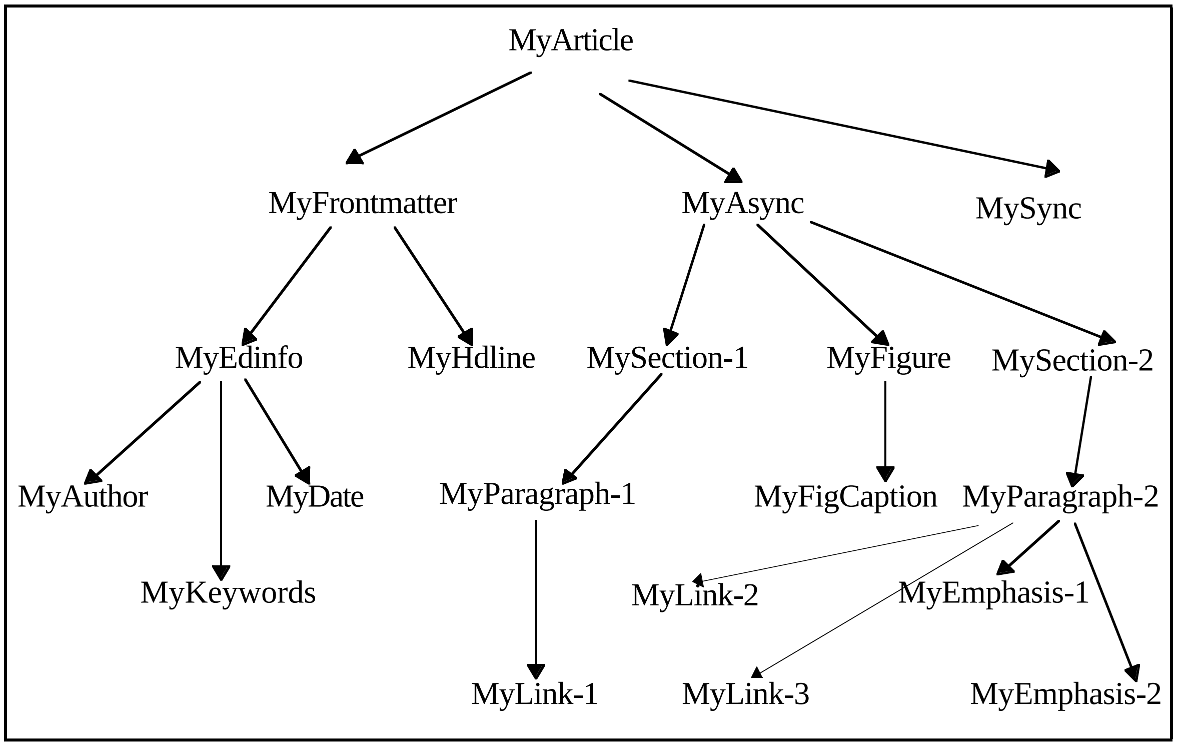


Figure 5.1: Partial Object Composition Hierarchy

object of type Image (or one of its subtypes depending on the type of the Image) for the picture of the building.

The synchronous part of the document that corresponds to the audio and video is shown in Figure 5.2. In the sample news document of Figure 2.2, it is assumed that a closed captioned video of the Guided Tour is associated with the article.

The closed caption video consists of the video, synchronous with the commentary (audio), along with captions which appear periodically, giving the French translation of the commentary. The three media are modeled as events in the finite coordinate space described in the DTD. The whole "audio visual" therefore consists of the two spatial axes (the time axes), the finite coordinate space, and the list of event extents along the axes.

Since there is only one closed captioned video, there is only one instance of the AudioVisual element in Figure 5.2, which has as its children the instances of the axes, the instance of the Av_fcs, and multiple instances of extent lists (*MyAv-extlist*).

The Av_fcs instance itself contains just one event schedule (there could be several if the commentary had been partitioned into logical segments). The event schedule is just the collection of the events occurring in the FCS. Since the audio and video data are not segmented, there is just one audio event, one video event; there are however several synchronized text (SText) event instances, one for each caption.

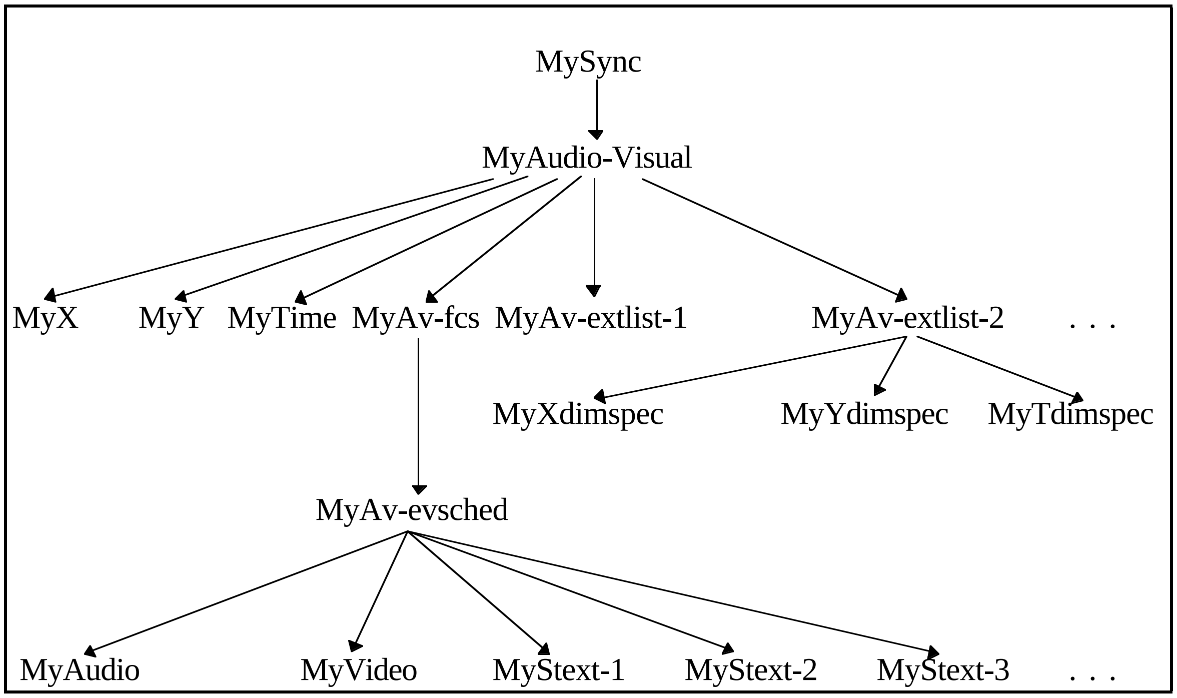


Figure 5.2: Remaining Composition Hierarchy (Synchronous Portion)

According to the DTD, each extent list consists of dimension specifications (dimspec), which in turn consist of marker lists (list of positions along the axes). The first two instances of the Av_extlist type are shown in the figure; the contained dimspec instances are shown for the second. The marker list is omitted since it is too involved to display in one figure.

Not shown in the composition hierarchy are the occurrences of instances of atomic types. In Figure 5.1, *MyFigure* has a reference to an instance of Image. In Figure 5.2, *My-Audio* has a reference to an instance of Audio, *MyVideo* to an instance of Video, and *MySText-1*, etc. have references to instances of SyncText.

5.2 Implementing the Type System

The implementation language is C++ [Stro91]. The specific implementation of the language used is the xlC product from IBM. The OBMS used is ObjectStore. Methods of a type are called (member) *functions* in C++. The term *interface* refers to a set of *methods*. Attributes are also called *data members*. Types have two special function members — a *constructor* and a *destructor*. Constructors usually involve memory allocation and initialization for a new instance of the type[1]. Destructors involve cleanup and deallocation of memory for that object. If no constructor or destructor is

[1]Actually, classes, and not types, have instances. However, all objects of a class have the same type.

provided, the compiler provides default implementations for them. Pointers to instances are also called references in the following discussion. For a type T, the type of the reference to an instance of type T is called pointer-to-T. In C++, the syntax is T*.

ObjectStore provides an extensive set of parameterized collection types which can store references to instances. The parameter is the type of the references of instances which can be inserted into the database.

5.2.1 Designing for Persistence

The ObjectStore data model provides persistence independent of type. Any C++ object can be made persistent and handled the same way as non-persistent objects. Once persistent, an object can be accessed either by navigation from other persistent objects, or by giving it a persistent name (a character string). These names are called *database roots*, or entry points.

ObjectStore does not maintain type extents automatically. Extents should be maintained by the user programs, if needed. Extents are essential for queries which search over a particular type of element. For example, the query may be: "select paragraphs which contain the word 'Computer'", or "select structured text elements containing the word 'Canada'". In this implementation, type extents are automatically maintained as persistent parameterized sets with the type as a parameter. These sets are database roots. For example, the extent declaration for the type Paragraph looks like:

```
os_Set<Paragraph*>* Paragraph_extent =
                        &os_Set<Paragraph*>::create(db,
                        os_collection::pick_from_empty_returns_null
                        );
```

This defines the extent of the Paragraph type to be a set of pointer to Paragraph. The default behavior of the set is modified so that a query over an empty set returns null, and does not signal an exception.

When a type is instantiated, the type's constructor ensures that a reference to the object is inserted into that type's extent. Similarly, the destructor of the type deletes the reference from the type's extent. When an object is constructed, its supertype's constructors are called before the object itself is created. This conveniently ensures that extents for all supertypes of an object are automatically maintained. For instance, when a Paragraph instance is created, references to this object are inserted (in sequence) in the extents for types Element, Structured, TextElement, StructuredText and Paragraph. This sequence implies that the constructor for type Element would be called twice: once each by the Structured and TextElement types. The result would be an inconsistent type extent containing duplicate pointers. By using the *virtual* inheritance mechanism of C++, this is avoided[2]. Destroying an object has the opposite effect to that of the constructor; it removes the references to the object from all the extents in the opposite sequence.

5.2.2 Abstract Types and Inheritance

An abstract type is a type which specifies interfaces without implementing at least one of them. Abstract types cannot be instantiated. In C++, a type is made abstract by declaring at least one of its methods (function members) to be *pure virtual*. A virtual function is a mechanism in C++

[2]Extents are sets; duplicate elements are not allowed. Therefore, even if a particular constructor was called multiple times during a single instantiation, the state of the extent would not change.

to support dynamic binding. A derived type can re-implement functions declared to be virtual in the supertype and hence override the implementation in the supertype. A pure virtual function is usually left un-implemented in the abstract type.

The purpose of an abstract type is to represent a general concept. In the type system for elements, types in the top level hierarchy are all abstract types. All types representing HyTime architectural forms are also abstract types. In the type system for atomic media, types Atomic, CMType, NCMType, and Temporal are abstract. A non-abstract type is referred to as a *concrete* type.

Implementing Abstract Types

An issue in implementing abstract types is whether abstract types should merely specify interfaces, at least one of which is not implemented. This design paradigm has an appeal to it. A type implementor who is implementing a type for a document element creates it as a subtype of one of the abstract types. The implementor then implements the interfaces specified in the abstract types for that subtype (and any additional methods defined for the subtype). But there could be methods that are implementable in the abstract supertypes. This implementation could then be inherited by the concrete subtypes. An example is the method GetArticle in the Element type. This can be implemented by having an attribute which is a reference to the Article instance which the element belongs to. GetArticle simply returns this reference. This method now needn't be re-implemented for every concrete type in the type system (there are 40 of them).

The disadvantage of the latter approach is that a constructor with a non-zero number of parameters needs to be implemented for type Element. This is because the attributes of Element need to be initialized. Since the constructor of a type calls the constructor of its supertypes, the type implementor needs to know about the constructors of all the supertypes and make explicit calls to them. In the first approach, there would be no need to write a constructor for the abstract type. The compiler would supply a default constructor.

In this implementation, the second approach is adopted — every method is implemented at the highest possible level in the type hierarchy. This approach tends to make certain portions of the implementation complex (the constructor for type Paragraph needs to call four constructors with three parameters each, for instance), but it cuts down on overall implementation time.

Return Types

In the last chapter, some of the methods of the types in the top level type hierarchy for document elements were described. These methods are implemented as virtual functions. The full C++ specification is given in Appendix B. The return types of these methods are the most general types. For example, method GetParent of type Element has a return type Element. However, in most cases, the type of the parent element of an element instance is constrained by the DTD. For example, instances of type Frontmatter can only have instances of type Article as a parent. Therefore, GetParent for Frontmatter would have its return type as reference to Article. Unlike the case for GetArticle, type Element does not have an attribute which is a reference to the parent instance. GetParent is a pure virtual function which is implemented in type Frontmatter. This type would have an attribute parent which is of type pointer-to-Article. GetParent would be implemented in Frontmatter, and not Element. The same reasoning applies for the declarations and definitions of GetNthChild in type Structured. The (simplified) declarations for types Element and Frontmatter are given below:

```
    class Element {
```

```
private:
      Article* articleElement;
public :
      virtual Element* GetParent() = 0;//pure virtual function
      virtual Article* GetArticle();//can be re-implemented by subtypes
};

class Frontmatter : public StructuredText{
private:
        Article* parent;
public:
        Edinfo* edinfo;
        Hdline* hdline;
        Subhdline* subhdline;
        AbsP* absP;
        Article* GetParent(); //implemented in this type
        TextElement* GetNthChild(int index); //inhrtd from Structured
};
```

C++ allows the return type of an overriding virtual function to differ from that of the overridden function provided that (a) the overridden function returns a reference to an instance of type T, and (b) the overriding virtual function returns a reference to a type derived from T. In the example above, Article is (indirectly) derived from Element.

5.2.3 Modeling DTD Constraints

The DTD specifies constraints on the type, the number, and order of the sub-elements of elements with complex content models (called structured elements). Modeling these constraints is non-trivial since there is little or no support from the C++/ObjectStore data model for these constraints. In the following, the specific problems and the solutions employed are discussed.

Union Types

Structured types model elements which have complex content models. These are non-leaf elements in the composition hierarchy. Since the document tree is ordered, their children are ordered, which gives the view of a tree as a nested list. Any structured element with an undetermined number of children will have a childList attribute. This attribute is a parameterized list whose parameter is the type of the child instances, as an attribute. For example, the type Sync has the content model and simplified C++ type declarations:

```
<!ELEMENT   sync  - -   (audio-visual+)>

class Sync : public Structured {
private:
    Article* parent;
public:
    os_List<AudioVisual*>* childList;
    Article* GetParent();
    AudioVisual* GetNth(int index);
};
```

The problem arises out of the use of the 'or' connector ('|') in conjunction with the '∗' or '+' connectors in the content model. For example, the Async element has the content model:

```
<!ELEMENT   async    - -  (section|figure|link)*>
```

There could have been three attributes for the Async type each of which is list of references of the type of one of the three elements listed on the right hand side (i.e, attributes sectionList, figureList and linkList). However, with this representation, the relative orderings between say, Section instances and Figure instances are lost. One solution to this problem is to have just one list of references of the existing common supertype of Section, Figure, and Link; this is Structured in this case. However, this leads to type checking problems since references to any subtypes of Structured (say Paragraph elements) could now be inserted into childList.

A second solution is to use union types: the parameter of the list of children is the union type of the three types: Section, Figure, and Link. Unions are present in the C++ data model and ObjectStore allows named union types to be made persistent. However, a discriminant method has to be provided to differentiate between the types in the union, and the user has to ensure that the right type is being accessed (i.e., there is run-time type checking). The third solution (the one adopted in this implementation) is to create an abstract supertype of Section, Figure, and Link. The convention adopted is to concatenate the first three letters from each type name to obtain the name of the new type. The parameter of the childList is then this supertype and there are no type checking problems. The drawback is that it creates an explosion of types in the system. Abstract supertypes created for this purpose are called *pseudo-union* types in this model. The declaration for type async now is:

```
class Async : public StructuredText{
private:
    Article* parent;
public:
    os_List<Fig_Lin_Sec*>* childList;
    Article* GetParent();
    Fig_Lin_Sec* GetNthChild(int index);
};
```

Pseudo-union types are subtyped from types already present in the element hierarchy. For example, the type Fig_Lin_Sec is a subtype of StructuredText. Pseudo-union types are abstract. In some cases, it is possible that the pseudo-union type is indeed a genuine supertype of the types involved. The modification to the type system after the inclusion of union types is a difficult task. The pseudo-union types defined in the type system are listed in Appendix B.4.

Ordering of Sub-elements

The second problem occurs in the use of the 'follows' connector (','). For example the element frontmatter has the content model:

```
<!ELEMENT frontmatter --(edinfo,hdline,subhdline,abs-p)>
```

This means that instances of Edinfo, Hdline, Subhdline, and Abs-p must follow each other in any document instance. To capture this constraint in the type system, a mechanism is needed to order the attributes of the type Frontmatter. This feature is not present in the data model of ObjectStore[3]. An implicit ordering of attributes is assumed in this case. The behavior of the Frontmatter type is such that it enforces the ordering. Thus, when the method GetNthChild with the value of parameter 'n' equal to 3, is applied to an instance of Frontmatter, the result is a reference to an instance of the type Subhdline.

Number of Sub-elements

The DTD declaration for the element `frontmatter` (given above) gives the constraint that there is exactly one `edinfo`, one `hdline`, one `hdline`, one `subhdline` and one `abs-p` sub-elements. None of the elements can be omitted. A wide variety of constraints on the number of sub-elements can be specified with the '∗', '+', and '?' occurrence operators.

There is no direct support from the data model for this constraint. There is no way of specifying, for instance, that a particular attribute cannot have a null value. The constraint can be enforced through the type's constructor. For example, the constructor for type Frontmatter would ensure that no Frontmatter instance is created unless it is supplied references to existing Edinfo, Hdline, Subhdline and Abs_p instances.

However, this is not such a clear-cut issue. Is it possible to build a database loader which could use such constructors? Is it efficient? Is it necessary for the constraints to be satisfied all the time? Could the database provide a consistent view of the document only after the whole document has been instantiated? Some of the responsibility for maintaining this DTD constraint also lies with the parser/database loader which ensures that the document being inserted conforms to the DTD for multimedia news articles.

In this implementation, since the database loader has not yet been designed, the issue is left open. However, the rudimentary instantiation program built to test the type system does try to enforce the number constraints.

5.2.4 Performance Issues

In the section describing the design of atomic types, a storage model for text which avoided fragmentation of the text component of the document was presented. This was done to optimize the performance of the database. It was also mentioned that there was a single type, type ArticleRoot which stored, as attributes, the lists of annotations of the logical elements. A complete listing of this type is given in Appendix B. This feature also enhances the performance of the browser in displaying text.

A related design goal is to minimize complex navigations of the hierarchy. By maintaining references to objects frequently accessed as attributes of ArticleRoot, access cost is reduced. For the multimedia news document, a frequent request from other system components is for the list of the atomic media instances present in the document. The references to these media instances are stored in lists which are attributes of the ArticleRoot type. A similar strategy is seen in the declaration of the GetArticle method in type Element. Any element can navigate through the parent elements

[3]Product types, which are not present in the data model of ObjectStore, can model attribute ordering.

to determine the article it belongs to. By providing an attribute whose value is the reference to the Article instance, this navigation is avoided.

To enhance the performance of the database, there are two key ways by which the *implementation* can help the DBMS. The first is by *clustering* related objects, and the second is by providing indices on collections to help optimize queries.

Because ObjectStore transfers data from the server to the client a page at a time[4], clustering related objects in the same object cluster optimizes performance. Clustering can be done at various granularities. For the hierarchical structure of the document instance, it is desirable to store the child instances of an element instance in the same cluster or segment as that of the element instance. As noted in the preceding section, an element instance may not have control over the instantiation of its child instances. It is left to the instantiation routines to carry out this optimization. However, all constructors which allocate memory for attributes of a type's instance, try to ensure that the allocation occurs within the same segment of the instance.

ObjectStore collections can be indexed on data members (attributes) only. Except for attributes which are are integer, character, or string valued, the type implementor has to provide the rank and hash functions which enable the index to be built. It is the responsibility of the type implementor to indicate which attributes are indexable. Once this is done, indexes can be automatically maintained (under updates to the attribute) by the DBMS. However, for attributes which are references to user defined types, or are character strings valued, user controlled index maintenance must be used. This involves calling index maintenance functions within the methods of the type which modify the attribute being indexed. With the read-only assumption, this usually involves modifying the constructor and destructor of the type. The user of the type must explicitly request ObjectStore to add or drop an index to a collection based on that attribute.

In Section 4.2.5, it was stated that the date, source, subject, and author are attributes (type String) of Article, even though these values are already stored (by means of annotations) as instances of Date, Source, Subject, and Author types respectively The reason for this replication becomes clear now. It is desirable to index the collection of Article instances on the values of these attributes, since queries predicated on these are likely to be frequent (Section 2.6). However, ObjectStore collections cannot be indexed on methods. The string value of instances of Date, Source, Subject, and Author can only be obtained by the application of the method GetString. Hence, although there could have been methods GetDate, GetSource, GetSubject, and GetAuthor for the Article type, it would not have been possible to build indices on these methods.

More details on indexes and queries can be found in [EM95].

[4]This data transfer policy can be changed.

Chapter 6

Related Work

The issue of database design for multimedia data has been tackled from the relational as well as the object-oriented data modeling perspectives. The design usually involves (a) defining a model for multimedia documents, and (b) defining models for multimedia data. Documents, multimedia documents in particular, are richly structured. Document models try to capture the structure of documents and in the case of hypermedia, the functionality of hyperlinks. Since multimedia data (specifically time-based media such as audio and video) differ from traditional data in their synchronization and temporal requirements, they require a data model different from conventional models. These are usually object-oriented models. Thus, for example, Atomic types have been defined to model these data.

Of the various media types which make up a multimedia document, the text component is by far the most richly structured. In the news-on-demand database design, the SGML standard has been adopted to describe this structure. Structures for video, image, and audio data involve spatial and temporal constraints. It is foreseeable that in the future these media will be just as richly structured as text [OT93].

In this chapter, database models of structured documents (specifically SGML/HyTime documents) are reviewed first. Later, other implementations of multimedia databases reported in literature are examined. Both relational and object-oriented models are considered, although the emphasis is on object-oriented models.

6.1 Database Models for Structured Documents

A comprehensive description of various database models for SGML documents proposed in literature is given in [SAZ94]. These models either employ (a) a direct representation of the DTD in the schema, or (b) the schema is defined external to the DTD (i.e., DTD-independent schema). Our database model is an example of the former.

Relational Models

An example of the latter approach is found in [BCK+94], where the task of incorporating support for structured text in a relational DBMS is tackled. To enable queries on structured text documents in

SGML format, extensions to SQL are proposed. Instances of document categories (defined by their DTDs) are fields with data type TEXT, of relations. Each TEXT field consists of the contiguous text content of the document along with the parse tree which represents the structure of the document. A schema creation for an article would look like:

```
    CREATE TABLE (aid INTEGER,
                  insert DATE,
                  document TEXT GRAMMAR article_dtd,
                  PRIMARY KEY (aid));
```

Here `article_dtd` is a reference to the article's DTD, `aid` is the article id and `insert` is the date the article was inserted. The EXPAND operator can be used to convert parts of the parse tree into fields of a relation. In this way, specific elements of the document can be extracted by selecting nodes in the parse tree. Updates to the TEXT field are not handled. The aim of the model is to integrate a relational DBMS with a text retrieval engine. It would be extremely difficult to incorporate support for multimedia and HyTime elements in this model. The idea of storing the text content contiguously and not fragmenting it, is found in the news-on-demand database design. However, the locations of the start and end of text element instances (annotations) are stored; the parse tree is implicit in the composition hierarchy (Section 5.1). A similar approach to storing SGML documents is found in [Macl90]. The DTD is external to the schema and the parse tree is stored as combinations of links to the parent elements and offsets from the start of the text. A full text retrieval system is integrated with this model.

Object-Oriented Models

The advantages of object-oriented database systems for multimedia/hypermedia applications over relational systems is highlighted in [Thur92] and [Bala93]. A case for using object-oriented databases for storage and retrieval of structured multimedia documents is made in [BA94]. Perhaps the earliest object-oriented approach is in [WKL86] which discusses the logical modeling of structured multimedia documents. In addition to standard DBMS functions, the authors identify other functional requirements of multimedia applications. These include aggregation, generalization, dynamic schema modification, modeling of presentation information, versioning, large data volume, content based access, and the ability to specify constraints between logical document elements. It is proposed to represent the constraints as methods of the objects representing the document elements. Since the work does not adopt any standard for document representation, there is very little a priori knowledge of the structure, and any object in the composition hierarchy can have a relationship with any other object, regardless of type or position. In the news-on-demand database design, the heterogeneity of the objects involved in the relationships is constrained by the document's DTD. For example, a `section` element is not allowed to be the sub-element of a `paragraph` element.

Querying of SGML documents is the focus of [CACS94], where extensions of two OBMS query languages are proposed. The DTD is mapped into an object-oriented schema implemented on the O_2 OBMS. Two extensions to the data model of O_2 are proposed. They are: (a) ordered tuples, or the ordering of attributes of a type, and (b) marked union types. Union types are introduced to handle the 'or' connector. For example a DTD entry using the '|' connector and the corresponding O_2 declarations are:

```
<!ELEMENT section -- ((title,body+)|(title,body*,subsectn+))>
```

```
class Section public type union(
                  a1:tuple(title:Title, bodies:list(Body)),
                  a2:tuple(title:Title, bodies:list(Body),
                          subsectns:list(Subsectn)))
                  constraint: (a1.title !=nil, a1.bodies !=list())
                             (a2.title !=nil, a2.subsectns !=list())
```

Here, a1 and a2 mark the union. The extensions to the query language are (a) the *contains* predicate to handle querying on strings, (b) implicit selectors to select the correct path while handling queries over union types, and (c) two new sorts to query text without exact knowledge of its structure (Section 2.6). Types representing unstructured document elements (with a #PCDATA content model, for example) are inherited from basic (atomic) types such as Text and Bitmap. This means that textual document elements are fragments of the text content of the document, which imposes a performance overhead while fetching the entire document. There are no inheritances relationships between classes.

Union types have been introduced here to handle alternative structures for the same document element (such as the section shown in the example above). In general, union types are needed to constrain the heterogeneity of the elements which can occur at a particular position in the composition hierarchy. Our approach for handling union types is described in Section 5.2.3. The ordering of attributes is visible through the behavior of the types. Queries with inexact knowledge of the document structure are not considered in our model. Querying on text content is handled by the method Match of the Text type. Since marked union types are not used in the news-on-demand type system, implicit selectors are not dealt with. The inheritance hierarchy described in Chapter 4, and the composition hierarchies described in Section 5.1 illustrate our approach: every element is an Element, and may have an instance of an Atomic type. If the element is a TextElement, then its content is obtained by the method GetString.

The design of an OBMS application to handle the storage of SGML documents is described in [BAH93]. This design also fragments documents according to the document's SGML type definition. The paper does not describe the querying facilities, but describes in detail how dynamic DTD handling is implemented by means of meta-classes.

The application, called D-STREAT, follows a layered approach by separating the DTD specific features and DTD independent features into two separate layers of classes. Document type-specific classes are specializations of the document type-independent classes. This means that features present in all SGML documents (methods to navigate the document tree for example) can be located in the DTD-independent layer which contains only one class called Document_Element. Furthermore, there are two meta-classes: TERMINAL and NONTERMINAL. The TERMINAL class models leaf nodes in the document hierarchy (i.e., #PCDATA elements), while the NONTERMINAL class models elements which are structured elements. Classes in the DTD-specific layer are instances of either of these meta-classes. The meta-classes have methods to create new document-specific classes at run time (createElemType). The content model of the new class can be set using another method, setContentModel. Instances of the DTD-specific class can be created at run time using the method createElem inherited from either meta-class. In this manner, DTDs can be dynamically created and inserted into the database. Finally, the content of the element can be set using the setContent method.

The schema is essentially a flat class hierarchy, with depth one. Every nonterminal element is a list of document elements. This implies that schema does not enforce the DTD constraints (i.e., the content model of the individual elements) on the heterogeneity of the elements comprising the list. It is argued that, since the SGML standard and the DTD do not define semantics for each document element type, they do not have any semantic relationships between them. Therefore, the classes representing the DTD elements are only specializations of the Document_Element class. The value of the attribute elementTypeName of the TERMINAL and NONTERMINAL meta-classes is the name of the logical document element the instance represents. Unions are not needed because there is no heterogeneity in the composition hierarchy. The authors envisage that the setContent method would check the insertion of a new element instance against the content model. The creation of a new element-type class is illustrated here:

```
<!ELEMENT    authlist  - - (author)*>

//representative classes are capitalized and
//first letter of instance names are capitalized
AUTHLIST := NONTERMINAL->createElemType("authlist");
AUTHLIST->setContentModel("(author)*");
Authlist1 := AUTHLIST->createElem();
Authlist->setContent(author1, author2);
//author1 and author2 are pre-existing instances of AUTHOR
```

The disadvantage of this approach is that a significant amount of type checking is done at document insertion time. In the news-on-demand database design, types are checked at compile time. There are similar implications for the processing of queries. The assertion that the element types never have semantics is debatable. For example, both `table` and `figure` element types may have a `float` attribute which specifies their position relative to the surrounding text. Conceivably, the two may have a common supertype because of this attribute. In addition, HyTime elements have semantics well defined by the HyTime standard.

Nevertheless, our approach has features similar to those of D-STREAT. The supertypes TextElement, Structured, and StructuredText can be said to be document type-independent types. However, our type hierarchy is not flat — the schema ensures that the document being inserted conforms to the DTD. In [BAH93], it is pointed out that the fact that the document structure is always a tree is factored into the DTD-independent layer. Thus, optimization algorithms for queries on tree structures could be integrated into the OBMSs query optimizer. Dynamic additions of DTDs to the database have not been considered in the news-on-demand database design[1].

The extension of D-STREAT to handle HyTime-encoded documents is described in [BMN94]. Since a document element conforming to an architectural form has both SGML and HyTime semantics, there is an additional HyTime layer in the model. Every architectural form has a meta-class modeling its semantics. For every HyTime conforming element instance in the document, there are two objects in the database. For example, `link` instances would have two classes (and hence two instances) representing them. The first class would be an instance of the NONTERMINAL meta-class, and the second would be an instance of the ILINK meta-class. The second class is also a specialization of the first. In our model, type Link is derived from both StructuredText and Ilink_AF types (Figure 4.6).

A document model based on the Office Document Architecture (ODA) is described in [BRG88] and [MRT91]. ODA is similar to SGML in that it allows for the specification of the logical structure of the document. In addition, it allows the specification of a layout structure, or the presentation information associated with the document. The papers mention object-oriented models as candidates

[1]This is possible. ObjectStore has metatypes in its data model and allows the dynamic addition of types to the ObjectStore database schema.

to model these structures. They define an additional layer, called the conceptual structure which is used to capture the semantics of the components of the logical structure. In [MRT91], it is recognized that support for *multimediality* is required; this is achieved by providing primitive classes for each media type. Querying this document model, and the optimization of such queries are described in detail in [BRG88].

6.2 Other Multimedia Databases

Presentational applications sometimes use OBMSs to manage persistent multimedia data and perhaps manage data capture and playback devices. These applications have common requirements such as composition, synchronization and playback control. Models and implementations exist which provide support for these requirements either by integrating an OBMS, or building the model around the OBMS [GBT93, SW94]. The intent is to provide a generic object-oriented toolkit which can be used by application developers to implement various multimedia applications. Other database models focus on a particular monomedia (e.g., video, or image), or on providing support for distribution, or on merely storing meta-information.

6.2.1 Presentation-Oriented Multimedia Databases

An object-oriented framework for modeling composite multimedia objects (such as multimedia documents) is proposed by the Object Systems Group at the University of Geneva in [GBT94] and [GBT93]. The focus is on providing a high level interface for multimedia programming. In particular, [GBT94] deals with data models for time-based media, and [GBT93] deals with so-called audio/video (AV) databases. These databases are collections of digital audio/video data and processes which can compose and aggregate these data. An AV database, therefore, not only stores data, but is also "involved with the capture, presentation and scheduling of complex objects, managing access and allocation of devices and channel bandwidths, and notifying the application of presentation-related events".

A similar all-encompassing approach is seen in [SW94] where a layered data model for multimedia applications is described. There are four layers: a data manipulation layer (DML), a data presentation layer (DPL) and a data control layer (DCL). The DDL is similar to the definition of Atomic types. The DML provides services to group DDL objects into so-called *MM events* and specifies an *event calculus* to allow construction of complex events. The DPL provides descriptions of how data is to be presented to the user and adds information to events from the DML (e.g. information on spatial layout, output format, icons) to produce a set of instructions for communicating the data to the user. The DCL manages the presentation - including playback control, I/O device control, and navigation. These models perform almost all of the functions of a HyTime engine [KRRK93, Rutl93].

The implementation of a persistent object-oriented system for HyTime documents is described in [KRRK93] and [Rutl93]. The database (implemented on ObjectStore) forms part of a HyTime engine (called HyOctane) which is used to process and display hypermedia documents represented using the HyTime standard. This design also fragments the document according to the element types in the DTD. The design is layered: there is an SGML layer, a HyTime layer, and an application layer.

There are only three classes in the SGML layer: the document class, the element class, and the attribute class. When a document is inserted into the database, an instance of the document class is created, with its fields as the collection of all instances of the elements of the document. The element instances in turn have references to their attributes which are instances of the attribute class. In the HyTime layer, each architectural form (AF) has a class associated with it. Instances of these AFs get inserted at document insertion time. The application layer has a class for each element type in the DTD. These get instantiated by the application process, which obtains information on them by querying the HyTime and SGML layers. The application then works from this layer. Updates to these objects get propagated down to the appropriate HyTime and/or SGML layers.

6.2.2 Miscellaneous Multimedia Databases

A novel object-oriented model for a video database is proposed in [OT93]. The model is schemaless, and includes inheritance by inclusion as an inheritance mechanism. This means that instances, not types, inherit attributes. Therefore, the hierarchical structure of a video object would be described by a series of derivations, and not by composition. However, it is not clear as to how one can navigate the structure — how does one get to the third scene of a movie, for instance? Other approaches to object-oriented models for multimedia data include [CAF$^+$91]. Incorporating structured video data will be a future extension to our design (through an extension to the DTD).

Others have focussed on the temporal aspects of multimedia data, and their synchronization. [HR93] describes a model for temporally composing multimedia objects and the playback of the composite objects. In [LG91], Little and Ghafoor present a procedure for the spatial and temporal composition of distributed multimedia objects. The environment is a distributed multimedia system (like the one described in Section 2.5). Object Composition Petri-nets (an augmented model of Petri net with logic of time intervals for Petri net execution) are used to specify the temporal constraints of compound multimedia objects. The destination workstation retrieves the temporal relationships from the server database, evaluates the Petri net, and derives the playback schedule. In [LG93] the same authors present a conceptual temporal model based on intervals to model the timing relationships present in compound multimedia objects. This model forms a basis for a hierarchical data model and for temporal access control algorithms to allow VCR-like capabilities. They show how it can be mapped to a relational database and derive the playback algorithm.

HyTime also adopts an interval based approach to modeling timing relationships, and establishes a composition hierarchy (Figure 5.2) using the document structuring feature of SGML. There is a single time axis for the whole document and all time intervals are defined on this axis. In our implementation, this representation is mapped to an instance of a Time Flow Graph [LG94] (a playback schedule)[2]. The Time Flow Graph is used by the synchronization component of the distributed multimedia system to synchronize data streams retrieved from the servers.

[2]Type Av_fcs, which models a Finite Coordinate Space has a method GetTimeSchedule, which returns a data structure representing the binary temporal relationships between the events in the FCS. From this data structure, the synchronization module derives the Time Flow Graph.

Chapter 7

Conclusions and Future Work

In this thesis an object-oriented multimedia DBMS design for a news-on-demand application has been described. The focus of the work reported here is the development of a type system that supports multimedia documents. There are three characterizing features of the type system design: (1) the central use of DBMS technology, (2) the reliance on object-oriented systems, and (3) strict adherence to international standards. The database is designed to accommodate actual multimedia objects as well as meta-information about them. The database schema consists of an object type system which follows the SGML/HyTime standard for document representation.

The other features of this work are:

1. The choice of a document representation standard. Three document representation standards were examined: Office Document Architecture (ODA), SGML, and HyTime. Based on their relative modeling powers and suitability to the application, the SGML and HyTime standards were chosen. The architecture of the processing environment for the database system was designed to support the choice of the SGML and HyTime standards.

2. The development of a Document Type Declaration (DTD) for multimedia news articles.

3. The development of a complete type system that is in complete harmony with the news-article DTD.

4. The annotation-based storage of text, which allows for efficient storage of documents as well as for fast search according to any of the document markups.

5. The successful implementation of the design on a commercial object oriented database management system (ObjectStore) and subsequent integration with other system components (including a visual query interface [EM95]) by other project participants.

The type system design implements a DTD definition for multimedia news documents. Populating the database with instances of other document types is not possible. Dynamic insertion of new DTDs needs to be investigated to allow insertion of different document types. Inserting new types at run time could involve using meta-types. An interesting question in the context for support for multiple DTDs is whether there are inheritance relationships between the types representing document elements in different DTDs. For example, there could be a slightly different article DTD, which does not have, say, quotes, but has bibliographic references.

A related goal which is important to consider in future, is modeling the whole DTD as an object in the database. This would support dynamic additions of new document types in the database.

The type system also needs to be made richer to support other HyTime architectural forms not used in the multimedia news DTD. In particular, the location AFs given in the location module need to be implemented. This includes supporting the functionalities of the HyQ query language.

This work has concentrated on the right-hand-side of Figure 2.1. Consequently, a multimedia document is currently entered into the database by creating the object instances according to the types defined in the database. The document being inserted is assumed to conform to its DTD. A database loader needs to be written which can validate multimedia documents and load the database with the type instances automatically. This can be done by coupling the multimedia DBMS with an SGML compiler which is retrofitted to instantiate object instances at the "code generation" step of compilation. For document elements which provide support for continuous media, HyTime capability needs to be added to this front-end. In this fashion, documents marked-up according to the SGML/HyTime standard can be automatically inserted into the database.

An important extension to the atomic types would be the ability to index media objects other than text, based on content. At this point, the only way to query images, for example, is to define a number of attributes and search on the values of these attributes. It is important to provide the facility to pose queries that refer to the content of the images rather than the attributes defined on them and be able to deal with such queries. The initial step in providing this facility would be to work on indexing of image contents.

In the long-run, an extensible OBMS that has inherent support for multimedia information systems is being developed. The intention is to use this system, called TIGUKAT [ÖPS+95], to eventually replace ObjectStore. Although it may not be possible to achieve the same performance, there will be opportunities to expand on the functionality and investigate the feasibility of various issues. It is difficult, if not impossible, to investigate all of the issues related to multimedia DBMS design by building a layer on top of a closed system such as ObjectStore. TIGUKAT is currently being prototyped at the Laboratory for Database Systems Research of the University of Alberta. It has a purely behavioral object model where the users interact with the system by applying behaviors to objects. In this way, full abstraction of modeled entities is accomplished since users do not have to differentiate between attributes and methods.

Bibliography

[BA94] K. Böhm and K. Aberer. "Storing HyTime documents in an object-oriented database," In *Proc. of CIKM '94*, pages 26–33, 1994.

[BAH93] K. Böhm, K. Aberer and C. Hürer. "Extending the scope of document handling: The design of an OODBMS application framework for SGML document storage," *Arbeitspapiere der GMD No. 811*, GMD-IPSI, Germany, 1993.

[BMN94] K. Böhm, A. Müller and E. Neuhold. "Structured document handling — a case for integrating databases and information retrieval," In *Proc. of CIKM '94*, pages 147–154, 1994.

[Bala93] V. Balasubramaniam. "State of the art review on hypermedia issues and applications," Internal document, Graduate School of Management, Rutgers University, Newark, New Jersey, 1993.

[BCK$^+$94] G. E. Blake, M. P. Consens, P. Kilpeläinen, P.-A. Larson, T. Snider and F. W. Tompa. "Text/relational database management systems: Harmonizing SQL and SGML," In *Proc. First Intl. Conf. Appl. of Databases*, pages 267–280, June 1994.

[BRG88] E. Bertino, F. Rabitti, and S. Gibbs. "Query processing in a multimedia document system," *ACM Trans. Office Information Systems*, **6**(1):1–41, January 1988.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet and M. Scholl. "From structured documents to novel query facilities," In *Proc. ACM SIGMOD Intl. Conf. Management of Data*, pages 313–324, May 1994.

[CAF$^+$91] S. Christodoulakis, N. Ailamaki, M. Fragonikolakis, Y. Kapetanakis, and L. Koveos. "An object-oriented architecture for multimedia information systems," *Q. Bull. of IEEE Tech. Comm. on Data Eng.*, **14**(3): 4-15, September 1991.

[DD94] S. J. DeRose and D. G. Durand. *Making Hypermedia Work — A User's Guide to HyTime*, Kluwer Publishers, 1994.

[DG92] N. Dimitrova and G. Golshani. "EVA: A query language for multimedia information systems," In *Proc. Intl. Workshop on Multimedia Information Systems*, pages 1–20, February 1992.

[DÖBS94] A. Dogac, M. T. Özsu, A. Biliris, and T. Selis. *Advances in Object-Oriented Database Systems*, Springer-Verlag, 1994.

[EM95] G. El-Medani. A Visual Query Facility for a News-on-Demand Multimedia Database. Master's Thesis, University of Alberta, Department of Computing Science, 1995 (forthcoming).

[Fox91] E. A. Fox. "Advances in interactive digital multimedia systems," *Computer*, **24**(10): 9–21, October 1991.

[GBT93] S. Gibbs, C. Breiteneder and D. Tsichritzis, "Audio/video databases: An object-oriented approach," In *Proc. 9th Intl. Conf. on Data Engineering*, pages 381–390, 1993.

[GBT94] S. Gibbs, C. Breiteneder and D. Tsichritzis. "Data modeling of time-based media," In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 91–102, May 1994.

[GDC$^+$92] C. Goble, M. Docherty, P. Crowther, M. Ireton, J. Oakley, and C. Xydeas. "The Manchester Multimedia Information System." In *Proc. Conf. on Extending Database Technology*, pages 39–55, 1992.

[Gold90] C. F. Goldfarb. *The SGML Handbook*, Oxford University Press, 1990.

[HBB$^+$94] A. Hafid, G. v. Bochmann, T. Burdin, R. Dssouli, J. Gecsei, B. Kerhervé and Q. Vu. "On news-on-demand service implementation," Publication #928, Département d'Informatique et de Recherche Opérationnelle, Universite de Montreal, September 1994.

[HR93] R. Hamakawa and J. Reikmoto. "Object composition and playback models for handling multimedia data," In *Proc. ACM Conf. Multimedia '93*, pages 273–281, October 1993.

[ISO86] International Standards Organization. Information Processing — Text and Office Information Systems — Standard Generalized Markup Language (ISO 8879), 1986.

[ISO89] International Standards Organization. Office Document Architecture (ODA) and Interchange Format (ISO 8613), 1989.

[ISO92] International Standards Organization. Hypermedia/Time-based Structuring Language: HyTime (ISO 10744), 1992.

[KRRK93] J. F. Koegel, L. W. Rutledge, J. L. Rutledge and C. Keskin. "HyOctane: A HyTime engine for an MMIS," In *Proc. ACM Multimedia '93*, pages 129–136, August 1993.

[LG93] T. D. C. Little and A. Ghafoor. "Interval-based conceptual models for time-dependent multimedia data," *IEEE Trans. Knowledge and Data Engineering*, **5**(4):551–663, April 1993.

[LG91] T. D. C. Little and A. Ghafoor. "Spatio-temporal composition of distributed multimedia objects for value added networks," *Computer*, **24**(10):42–50, October 1991.

[LG94] L. Lamont and N. D. Georganas, "Synchronization architecture and protocols for a multimedia news service application," In *Proc. IEEE International Multimedia Computing and Systems Conf.*, 1994.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. "The ObjectStore database system," *Communications of the ACM*, **34**(10): 50–63, October 1991.

[Macl90] I. A. Macleod. "Storage and retrieval of structured documents," *Information Processing and Management*, **26**(2):197–208, February 1990.

[MRT91] C. Meghini, F. Rabitti, and C. Thanos. "Conceptual modeling of multimedia documents," *Computer*, **24**(10): 23–30, October 1991.

[NY94] R. Ng and J. Yang. "Maximizing buffer and disk utilizations for news-on-demand," In *Proc. 20th Intl. Conf. on Very Large Databases*, pages 451–462, 1994.

[ÖPS+95] M. T. Özsu, R. Peters, D. Szafron, B. Irani, A. Lipka, and A. Munoz. "TIGUKAT: A uniform behavioral objectbase management," *VLDB Journal*, 1995 (To appear).

[ÖSEV95] M. T. Özsu, D. Szafron, G. El-Medani and C. Vittal. "An object-oriented multimedia database system for a news-on-demand application," *ACM Multimedia Systems Journal*, 1995 (To appear).

[OT93] E. Oomoto and K. Tanaka. "OVID: Design and implementation of a video-object database system," *IEEE Trans. Knowledge and Data Management*, **5**(4):629–643, August 1993.

[ÖV91] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*, Prentice-Hall, 1991.

[Pric93] R. Price. "An introduction to the future international standard for hypermedia object interchange," In *Proc. ACM Multimedia '93*, pages 121–128, 1993.

[Rutl93] L. Rutledge. A HyTime Engine for Hypermedia Document Presentation. Master's Thesis, University of Massachusetts Lowell, 1993.

[SAZ94] R. Sacks-Davis, T. Arnold-Moore and J. Zobel. "Database systems for structured documents". In *Intl. Symp. Advanced Database Tech. and their Integration (ADTI '94)*, 1994.

[SLVZ95] B. Subramaniam, T. W. Leung, S. L. Vandenberg and S. Zdonik. "The AQUA approach to querying lists and trees in object-oriented databases," In *Proc. of Intl. Conference on Data Engineering '95*, 1995.

[Stev91] S. M. Stevens. "Next generation network and operating system requirements for continuous-time media," In *Proc. Second International Workshop on Network and Operating System Support for Audio and Video*, pages 197–207, 1991.

[Stro91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991.

[SW94] G. A. Schloss and M. J. Wynblatt. "Building temporal structures in a layered multimedia data model," In *Proc. ACM Multimedia '94*, pages 271–278, 1994.

[Thur92] B. Thuraisingham. "On developing multimedia database management systems using the object oriented approach," *Multimedia Review* **3**(2):11–19, Summer 1992.

[WKL86] D. Woelk, W. Kim, and W. Luther. "An object-oriented approach to multimedia databases," In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 311–325, May 1986.

[VÖSE94] C. Vittal, M. T. Ozsu, D. Szafron and G. El-Medani. "The Logical Design of a Multimedia Database for a News-On-Demand Application," Technical Report 94–16, Department of Computing Science, University of Alberta, December 1994.

Appendix A

DTD for Multimedia News Articles

```
DOCTYPE     article     SYSTEM      "article.dtd" [

<!-- HyTime Modules Used -->
<?HyTime support base>
<?HyTime support measure>
<?HyTime support sched manyaxes=3>
<?HyTime support hyperlinks>

<!-- Non-HyTime Notations used   -->
<!NOTATION virspace PUBLIC    -- virtual space unit (vsu)--
      "+//ISO/IEC 10744//NOTATION Virtual Measurement Unit//EN">

<!-- Document Structure -->
<!ELEMENT   article     - -   (frontmatter, async, sync)>
<!ELEMENT   frontmatter - -   (edinfo, hdline, subhdline, abs-p)>
<!ELEMENT   edinfo      - -   (loc & date & source & author+ &
                                          keywords & subject)>
<!ELEMENT   (loc|source|subject)    - -   (#PCDATA)>
<!ELEMENT   (hdline|subhdline)      - - (#PCDATA)>
<!ELEMENT   date    - -   (#PCDATA)>
<!ELEMENT   (author|keywords) - -   (#PCDATA)>
<!ELEMENT   abs-p   - -   (paragraph)>
<!ELEMENT   async   - -   (section|figure|link)*>
<!ELEMENT   section - -   (title?, (paragraph|list)*)>
<!ELEMENT   title   - -   (#PCDATA) >
<!ELEMENT   paragraph - - (emph1|emph2|list|figure|link|quote|#PCDATA)*>
<!ELEMENT   (emph1|emph2|quote) - -   (#PCDATA) >
<!ELEMENT   list         - -   (title?, listitem+)>
<!ELEMENT   listitem     - -   (paragraph)*>
```

```
<!ELEMENT   link         - -   (emph1|emph2|quote|figure|#PCDATA)+>
<!ELEMENT   figure       - -   (figcaption?) >
<!ELEMENT   figcaption   - -   (#PCDATA)>
<!ELEMENT   sync         - -   (audio-visual+)>
<!ELEMENT   audio-visual - -   (x, y, time, av-fcs, av-extlist+)>
<!ELEMENT   (x|y|time)   - -    EMPTY>
<!ELEMENT   av-fcs       - -   (av-evsched+)>
<!ELEMENT   av-evsched   - -   (audio*, video*, stext*)>
<!ELEMENT   (audio|video|stext) - -   EMPTY>
<!ELEMENT   av-extlist   - -   (xdimspec, ydimspec,tdimspec)>
<!ELEMENT   (xdimspec|ydimspec|tdimspec) - - (axes-marklist)>
<!ELEMENT   axes-marklist - - (#PCDATA)>

<!ATTLIST article
            id          ID    #REQUIRED
            HyTime      NAME  #FIXED HyDoc>

<!ATTLIST   quote
            source      CDATA #IMPLIED>
<!ATTLIST   author
            designation CDATA #IMPLIED>

<!ATTLIST   figure
            filename    CDATA #REQURED
            format      CDATA #REQUIRED>
<!ATTLIST   (x|y|time)
            HyTime      NAME  #FIXED axis
            id          ID    #IMPLIED
            axismeas    CDATA #FIXED "virspace"
            axismdu     CDATA #FIXED " "
            axisdim     CDATA #FIXED "virspace">
<!ATTLIST   link
            HyTime      NAME  #FIXED ilink
            id          ID    #REQUIRED
            linkends    IDREFS      #IMPLIED>
<!ATTLIST   av-fcs
            HyTime      NAME  #FIXED fcs
            id          ID    #REQUIRED
            axisdefs    CDATA #FIXED      "x y time">
<!ATTLIST   av-evsched
            HyTime      NAME  evsched
            id          ID    #REQUIRED
            axisord     CDATA #FIXED      "x y time"
            basegran    CDATA #FIXED      "vsu vsu vsu">
<!ATTLIST   (audio|video)
            HyTime      NAME  #FIXED event
            id          ID    #REQUIRED
            filename    CDATA #REQUIRED
            format      CDATA #REQUIRED>
<!ATTLIST   stext
            HyTime      NAME  #FIXED event
            id          ID    #REQUIRED
            filename    CDATA #REQUIRED>
<!ATTLIST   av-extlist
            HyTime      NAME  #FIXED extlist
            id          ID    #REQUIRED>
<!ATTLIST   av-dimspec
            HyTime      NAME  #FIXED dimspec
            id          ID    #REQUIRED>
<!ATTLIST   axes-marklist
            HyTime      NAME  #FIXED marklist
            id          ID    #REQUIRED >

]>
```

Appendix B

Type Declarations

B.1 Atomic Types

```
/* class Atomic : Parent class in the hierarchy for basic media types
 * private members :
 *     length : size of the media object
 *     hostQoSParam: pointer to object holding media independent
 *                   QoS parameters
 * protected members:
 *     Atomic: constructor (protected to make the class abstract)
 *     name  : keep track of multiple instances of the same object at
 *             different hosts.
 * public members  :
 *     ~Atomic    : virtual destructor
 *     GetLength : returns the size of the object in characters;
 *     GetName   : returns the name of the object (temporary feature)
 * subtypes: NCMType, CMType
 */

class Atomic{
private:
    unsigned long length;
    HostQoSParameter *hostQoSParam;
protected:
    char* name;
    Atomic(unsigned long length,
           char *name =0,
           HostQoSParameter *qos =0);        //constructor
public:
    static os_typespec* get_os_typespec();
    virtual ~Atomic();        //destructor
    unsigned long GetLength();
    char* GetName();
    HostQoSParameter* GetHostQoSParameter();
};

/* class NCMType : Parent class of non continuous media types
 * private members :
 *      content :array of characters representing the object
 * protected:
 *      NCMType : constructor (to make it an abstract class)
 * public members :
 *      ~NCMType: destructor
 *      GetContent: return the content of the object as char *
 * subtypes : Text, Image
 */
class NCMType : public Atomic{
private:
    char *content;
protected:
    NCMType(const char* content, unsigned long length,
            char* name =0, HostQoSParameter* qos =0);
public:
    static os_typespec* get_os_typespec();
    ~NCMType();        // destructor virtual (as it is abstract).
    char* GetContent();
};

/* class CMType : Parent class of continuous media types
 * private members :
 *    filename : name of the file containing the object
 *    location : location of the file (site, directory).
 *    uoi        : the Universal Object Identifier
 * protected:
 *      CMType : constructor  (to make it an abstract class)
 * public members :
 *      ~CMType: destructor (virtual since type is abstract)
 *      GetFilename: return the file name
 *      GetLocation: return the location
 *      GetUOI : returns the Universal Object Identifier required by the CMFS
 * subtypes : Temporal, SyncText
 */
class CMType : public Atomic{
```

```
private:
    char *filename;
    Location *location;
    UOI uoi;
protected:
    CMType(const char *filename, Location *location,
             UOI uoi, int length, char *name =0,
             HostQoSParameter *qos =0);
public:
    static os_typespec *get_os_typespec();
    virtual ~CMType();
    char *GetFilename();
    Location *GetLocation();
    UOI GetUOI();
};

/* class Temporal: Parent class of continuous media types
 *                      with temporal attributes
 * private members:
 *     duration: length of playback (in seconds)
 * protected:
 *     Temporal: constructor  (to make it an abstract class)
 * public members :
 *     ~Temporal: destructor (virtual since it is abstract)
 *     GetDuration: returns the value of the duration attribute
 * subtypes: Video, Audio
 */
class Temporal : public CMType{
private:
    unsigned long duration;       //in seconds
protected:
    Temporal(unsigned long duration, const char *fname,
               Location * location, UOI uoi, unsigned long length,
               char *name =0, HostQoSParameter *qos=0);
public:
    static os_typespec *get_os_typespec();
    virtual ~Temporal();
    unsigned long GetDuration();
};
```

B.2 Top-Level Element Hierarchy

```
/* class    Element : Parent class in the hierarchy for element types
 * private members
 *     articleElement : the article to which the element instance belongs
 * protected members
 *     Element: constructor (protected to make the class abstract)
 *              maintains extents
 *              Default constructor to be used when element does not
 *              belong to any article
 * public members
 *     ~Element    : virtual destructor
 *     GetArticle : returns the pointer to the Article instance
 *     GetParent  : pure virtual function - returns the parent of the
 *                    instance in the composition hierarchy
 * subtypes: TextElement, Structured, HyElement
 */
class Element{
private:
        Article *articleElement;
protected:
        Element();
        Element(Article *);
public :
        virtual ~Element();
        virtual Element *GetParent() = 0;
        Article *GetArticle();
};

/*class TextElement: Elements which are text instances and can be re-
 *                    presented by annotations.
 *private members   :
 *protected members:
 *      TextElement : constructor made protected to enforce abstractness
 *                    of the class.
 *public members    :
 *        absoluteAnnotation : annotation of the text element with
 *                             respect to the entire text representing
                               the text content of the article.
 *      GetString   :returns the string value of the TextElement instance.
 *      GetAbsoluteAnnotation : returns the annotation object.
 */

class TextElement : public virtual Element{
```

```
protected:
    TextElement(Annotation *, Article *);
    TextElement();
public :
    Annotation *absoluteAnnotation;
    virtual ~TextElement();
    Annotation *GetAbsoluteAnnotation();
    char *GetString();
};

/*class Structured : Elements which have subelements (children)
 *                    Virtual inheritance from Element so that
 *                    multiply inheriting subtypes will have only
 *                    copy of the Element subobject
 *private members :
 *protected members:
 *    Structured : constructor made protected to enforce abstractness
 *                 of the class. Default constructor to be used when
 *                 there is no article instance.
 *public members   :
 *     ~Structured : destructor ; removes instance from extent
 *      GetNth     : returns nth sub-element (count starts from
 *                     one, not zero). Pure virtual function
 */

class Structured : public virtual Element {
private :
protected:
    Structured();
    Structured(Article *);
public :
    virtual ~Structured();
    virtual Element *GetNth(int) = 0;
};

/* class StructuredText : text elements which are structured.
 * private members : (none)
 * protected members :
 *       StructuredText : constructor, made protected to enforce
 *                 abstractness of type. Maintains extent
 * public members :
 *        ~StructuredText : destructor; removes instance from extent.
 *         GetNth: inherited from Structured. Returns nth child element
 *                    (return type changed to TextElement). Pure virtual
 *                           function
 */

class StructuredText: public virtual TextElement, public virtual Structured{
private :
protected:
    StructuredText(Article *, Annotation*);
    StructuredText();
public :
    virtual ~StructuredText();
    virtual TextElement *GetNth(int)=0;
};
```

B.3 Type ArticleRoot

```
class ArticleRoot{
public:
    static os_typespec* get_os_typespec();
    //list of atomic media
    Text* textBlock;
    os_List<Image*>* imageList;
    os_List<Audio*>* audioList;
    os_List<Video*>* videoList;
    os_List<SyncText*>*stextList;
    //the article instance where all this stuff belongs
    Article* article;
    //Annotations for various singly occurring TextElements
    Annotation* edinfo;
    Annotation* loc;
    Annotation* keywords;
    Annotation* source;
    Annotation* author;
    Annotation* subject;
    Annotation* date;
    Annotation* frontmatter;
    Annotation* hdline;
    Annotation* subhdline;
    Annotation* absP;
    Annotation* title;
    // Annotation lists.
    //the non-Structured TextElements
    os_List<Annotation*>* figcaptionList;
    os_List<Annotation*>* emph1List;
    os_List<Annotation*>* emph2List;
    os_List<Annotation*>* quoteList;
    os_List<Annotation*>* titleList;
    //StructuredTextElements
    os_List<Annotation*>* listList;
    os_List<Annotation*>* paragraphList;
    os_List<Annotation*>* figureList;
    os_List<Annotation*>* sectionList;
    os_List<Annotation*>* listItemList;
    os_List<Annotation*>* linkList;

    ArticleRoot(Text* textBlock);
    ~ArticleRoot();
};
```

B.4 Union Types

```
//for class Async
class Fig_Lin_Sec: public virtual StructuredText{
};

//for class AudioVisual
class Ext_Fcs_Tim_X_Y : public virtual HyElement {
};
//for class Av_evsched
class SAu_STe_SVi : public virtual Event_AF{
public:
        virtual CMType *GetContent() =0;
        virtual char *GetSummary() =0;
        virtual os_List<TDimspec*> *GetTimeExtents() =0;
};

//for class Paragraph
class Emp_Fig_Lin_Quo : public virtual TextElement{
};

//for class Link
class Emp_Fig_Quo : public virtual Emp_Fig_Lin_Quo{
};
```