University of Alberta

THE LOGICAL DESIGN OF A MULTIMEDIA DATABASE FOR A NEWS-ON-DEMAND APPLICATION

by

Chiradeep Vittal
M. Tamer Özsu
Duane Szafron
Ghada El Medani

DEPARTMENT OF COMPUTING SCIENCE

The University of Alberta
Edmonton, Alberta, Canada

The Logical Design of a Multimedia Database
for a
News-on-Demand Application

Chiradeep Vittal
M. Tamer Özsu
Duane Szafron
G. El Medani

Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2H1
{vittal,ozsu,duane,ghada}@cs.ualberta.ca

Abstract

We describe the design of a multimedia database for a distributed news-on-demand multimedia information system. News–on–demand is an application that utilizes broadband network services to deliver news articles to subscribers in the form of multimedia documents. Different news providers insert articles into the database, which is then accessed by remote users over a broadband network. Multimedia documents are composite objects where the component objects have spatial and temporal relationships which need to be captured in the database. Modeling of multimedia documents involves three issues: (1) modeling of individual document components (i.e., monomedia objects such as text, images, etc), (2) modeling of the document structure, and (3) modeling of the presentation structure. We take an object-oriented approach to dealing with these issues. Within (1), our research has so far concentrated on text. We use an *annotation* based scheme where entire text is stored as a single document and the formatting mark-ups are represented as annotations on the text. To model the structure of multimedia news documents, we follow the SGML/HyTime international standard by designing a document type declaration (DTD) for multimedia news articles. We build an object oriented model of multimedia news documents based on this DTD. Finally, we model the presentational aspects (e.g., fonts, number of columns, playing of audio and video) as objects and store them in the database..

Acknowledgment

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Multimedia information systems integrate diverse media sources such as text, video, speech and images to enable a variety of multimedia applications. Many of the current multimedia information systems do not use database management technology, for two main reasons. Some systems are single-user systems running on personal computers. In other systems, the designers perceive that database management systems (DBMSs) introduce performance penalties that cannot be tolerated due to real-time constraints. The use of DBMSs in these systems has, by and large, been restricted to storing meta-information about the data – clearly a directory service – rather than as the repository of the multimedia data.

We contend that the multimedia database is at the heart of a multimedia information system. A multimedia database stores two types of entities:

1. the monomedia objects such as text, data, audio, video, and still images as well as the documents that contain these objects, and

2. meta-information, such as the temporal relationships among monomedia objects and the quality-of-service (QoS) data, that are used by other system components.

The challenge is to provide an integrated repository for all multimedia objects so that they can be accessed by all system components. In this report we describe an object-oriented database design of a multimedia database for a news-on-demand application. The issues that we address range from logical database modeling to the development of an objectbase management system (OBMS)[1] with application-specific query languages. Here, we concentrate only on the logical database modeling aspects.

There are three characterizing features of our work: (1) the central use of DBMS technology, (2) the reliance on object-oriented systems, and (3) strict adherence to international standards. Each of these features need to be justified.

Despite the existence of a number of "multimedia file systems," traditional file systems are not appropriate for this task. One reason for this is the standard argument in favor 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. The size of multimedia objects have been discussed extensively in literature; for example, each of the following takes 1 Mbytes of storage in uncompressed form [Fox91]: six seconds of CD-quality audio, single 640x480 color image with 24 bits/pixel, single frame (1/30 sec-

---

[1] We prefer the term "objectbase management system" to the more common "object-oriented database management system" since what is stored and managed in these systems are not only data, but encapsulated "objects" such as images, text, etc, and the operations that can be applied to them.

ond) CIF video, or one digital X-ray image (1024x1024) with 8 bits/pixel. 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. A second important reason is that multimedia objects have temporal and spatial relationships such as synchronization, and display location of information between captioned text and video. These relationships should be modeled explicitly as part of the stored data. Thus, even if the multimedia data is stored in files, their relationships need to be stored as part of the meta-information in some DBMS. As indicated above, 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. Finally, multimedia applications are generally distributed. Both the target application (news-on-demand) and many other multimedia applications require multiple servers to address their storage requirements. Thus, distributed DBMS technology [ÖV91] can be put to use to efficiently and transparently manage data distribution; distributed file systems are no match to distributed DBMSs in their functionality.

We use object technology for a variety of reasons. First, multimedia objects are complex in their structure. The *primitive objects* (monomedia objects) are not only simple strings or numbers (e.g., names, addresses, and salaries of employees), but also include video, digitized voice and images. Multimedia documents are structured complex objects containing a number of these primitive objects. Each component of the document may itself be composite, resulting in combinations of audio and video, image and text, etc. 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 sitting at the root of the tree. As an example, a book is made up of chapters; chapters consist of sections; sections consist of paragraphs and figures, and so on. In other words, there is a distinction between the document content and the structure of the document.

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 [DG92].

Second, multimedia information systems require an extensible data model that allows application designers to define new types as part of the schema. Furthermore, the applications themselves must be able to add and delete new object types dynamically. Therefore, multimedia systems must not have static schemas and the DBMS must be able to handle dynamic schema

changes. Object-oriented systems meet these requirements much better than relational ones.

We also have 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. Consequently, we would like to test this often repeated claim.

The third characterizing feature of our work is our adherence to international standards for multimedia document representation. This is essential in a project such as ours which involves many partners. Furthermore, 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 (*Standard Generalized Markup Language*) [ISO86] has been chosen as the standard to follow because of its better suitability for the target application, its relative power, its widespread use (for example, the Hypertext Markup Language, HTML, that is the basis of World Wide Web is an application of SGML) and its role as the basis of the ISO 10744 HyTime [ISO92] hypermedia representation standard. SGML mostly deals with textual documents whereas HyTime adds support for hypermedia documents (e.g., links, video, etc.). We provide an overview of SGML and HyTime concepts in Sections 4.1 and 5.1, respectively.

The platform we use to implement our design is the ObjectStore database management system [LLOW91]. ObjectStore uses C++ as its programming language interface. The development environment is the C Set++ package on IBM/RS6000 machines running AIX.

The rest of this report is organized as follows. In Section 2, we describe the application environment, highlighting the important characteristics of the target application and a running example that is used in this report. Sections 3 to 5 present the main contributions of the report and address the three main design issues: how to represent and store individual multimedia objects in the database, how to represent and store the multimedia document structure in the database, and how to represent and store spatio-temporal relationships in the database. Section 6 presents an example that ties in the concepts introduced in sections 3 to 5. We provide, in Section 7, a brief overview of a visual querying facility that we have developed for the multimedia database. Section 8 reviews related work reported in literature. We conclude, in Section 9, with a discussion of the current status of this project.

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 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 1).
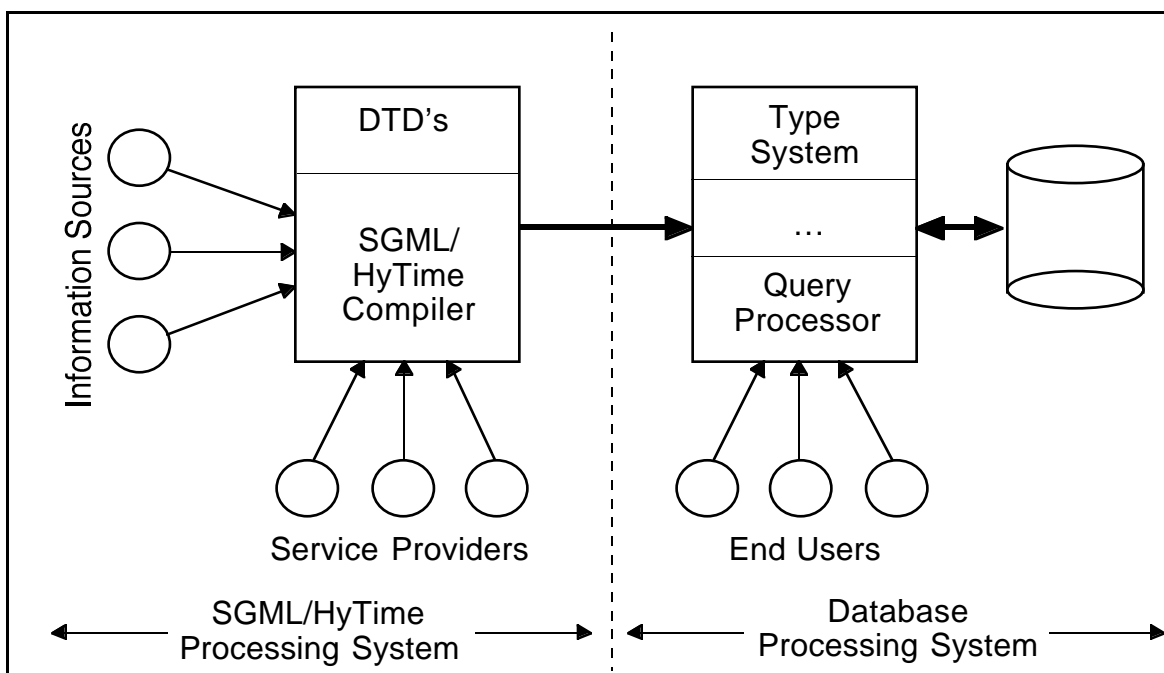
Figure 1. Processing Environment

The scenario for the News-On-Demand application just described brings up two issues:

1. There are several news providers inserting documents into the database from different remote sites, over a network. 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. In other words, the representation of the document 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 1). News providers compose hypermedia articles on their own authoring systems. These articles are then translated to the SGML/HyTime representation. A 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 (the quality-of-service negotiation module, the synchronization module) to obtain relevant meta-information. Our subproject currently focuses on the database processing side of Figure 1.

2. Once inserted into the database, the news article is not updated by either the news provider or the subscriber. Thus, we have a *read–only* model for the database. The news provider may insert newer *versions* of the news article however, as time progresses. The database management system would handle the version management issues.

2.2 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. 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 sitting at the root of the tree. As an example, a book is made up of chapters; chapters consist of sections; sections consist of paragraphs, and so on. In other words, 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.*

2.3 A Sample Multimedia News Article

This section describes a sample multimedia news document that will be used as a running example throughout this report. As mentioned before, a document is a collection of information pieces related to a certain subject. As an example, we use a news article about the recently concluded Commonwealth Games. Since there is so much information, there is more likely a series of articles on the subject. We choose the inauguration of the Games by the Queen as the subject of the article. We will describe the document components in terms of the media present in the document; the full document is depicted in Figure 2.

- The **text** portion consists of the written report on the inauguration cere mony. Included in this is data that may not be shown in the final docu-

Queen opens Games

Victoria, Aug. 18. The Queen today officially inaugurated the 15th Commonwealth Games in Victoria, B.C., at the University of Victoria. The Centennial Stadium was filled to it's capacity of 30,000.

Queen Elizabeth II

Some 3500 athletes from 64 countries are competing in track and field events, swimming, gymnastics, lawn bowling etc. *Lacrosse* is being introduced as an exhibition sport in the Games.

Speech   Video Coverage   More News

— C. Dickens

Figure 2. Sample News Document Presentation

ment presented for viewing, such as the keywords associated with the document. Other textual logical components of the document would be the title, the (optional) subtitle, an (optional) abstract paragraph, the date and location of the news item, the paragraphs that make up the article content, the author, and perhaps the titles of any images appearing in the text.

- The **images** in the document are any pictures related to the subject of the article. For example, since one of the main subject of the article is the Queen, her picture may be an appropriate component of the document. The image can be stored in any format (GIF, TIF, JPEG, etc.). The presentation of the image is also independent of the logical structure, because we may choose to reproduce the image inline with the rest of the document, or display it in a separate window.

- The **sound** or audio component of the document is the recording of the speech given by the Queen at the inauguration. 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 could be the television recording of the speech. 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 TV clip featuring the Queen's speech, the voice of the Queen is synchronized with the video so that the viewer does not find the lip movements out of phase with the sound of the voice being played back. There could be text subtitles displayed along with the video, giving the French translation of the speech.

- 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 documents, or document components where further information can be found, the document enhances its information capacity. Another possibility is that the user may want to make comments, or annotations on the text that would be visible the next time the document is retrieved.

In Figure 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. Again, this is a presentational aspect separate from the logical structure of the document.

It is important to note that Figure 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.

2.4 Querying

As noted before, queries on the document are read-only in nature with no updates after the document is inserted. The following retrieval scenario elaborates on the type of queries the user may perform. A detailed description can be found in [EÖSV95].

- The user wishes to see some articles on the Commonwealth Games. Alternatively, the request may be to view some articles featuring the Queen in them. Therefore, the database is queried for all documents with the *keywords* **Commonwealth Games** in them (or, the **Queen**).

- 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 can be paid for) with the system. Note that each of these additional pieces of information is obtained by the user interface by querying the documents in the list.

- The user selects one particular article (for example, the one described in the previous section), and retrieves the document after negotiating the cost of access.

- The retrieval process itself could trigger additional queries to the document in the database. For example, the physical location of the different media types could be on different servers. This has to be determined by the processes involved in the retrieval (cf. Section 2.5). If the different components of the article need to be synchronized in time, the playback schedule is determined from the synchronization information present in the document. The cost, the schedule, the quality of service parameters, and the physical location of the media, are called *meta-information.* Querying for meta-information is a very important support feature provided by the database.

- Although a keyword based search is the most likely scenario, there are other queries possible that would return a list of documents matching the search criteria. For example:
  - return documents with a particular text string within the text of the article.

- return documents with video, but no text.
- return documents with a certain *location* and *date.*
- return documents by a certain *author* , etc.

- Queries can be performed on the displayed document too. Text string matching is a common example. Following the links within the document could result in more queries by the s   determine the meta-information associated with the new document.
- The database can be queried for presentation information. Presentation information includes synchronization information, information on how the document is spatially laid out on the rendition medium, and formatting of various document components. *Emphasis* elements could be set out in italics (as in *Lacrosse*, in Figure 2), titles in 18 point bold font, and images could be displayed in grayscale, rather than color. The browser displaying the document would query the database to determine this information. The presentation style could either chosen by the user, or a pre-defined presentation style could be used.
- Other complex queries such as "return all documents with quotes by the Queen, on the Commonwealth Games". Note that we only deal with queries on information already in the database. We do not consider queries that require the database to "understand" the document.

2.5 System Architecture and RetrievalProcess

The stored multimedia data is classified as *continuous media*, or *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. Continuous media presents some of the more pressing issues in multimedia information systems and significantly influences the design and the load of the system. Non-continuous media such as text and still images do not have the real–time constraints of audio and video. In our system, continuous media and non-continuous media are stored on different servers.

Figure 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   (CM servers). The distribution of data is transparent to the users since they interact with a querying facility [EÖSV95] at the client, rather than directly accessing individual servers. 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 the client routines to determine the location of a particular piece of multimedia data. 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.

The retrieval of the document involves several system components, each of which must access the database to determine information necessary for the completion of their tasks.
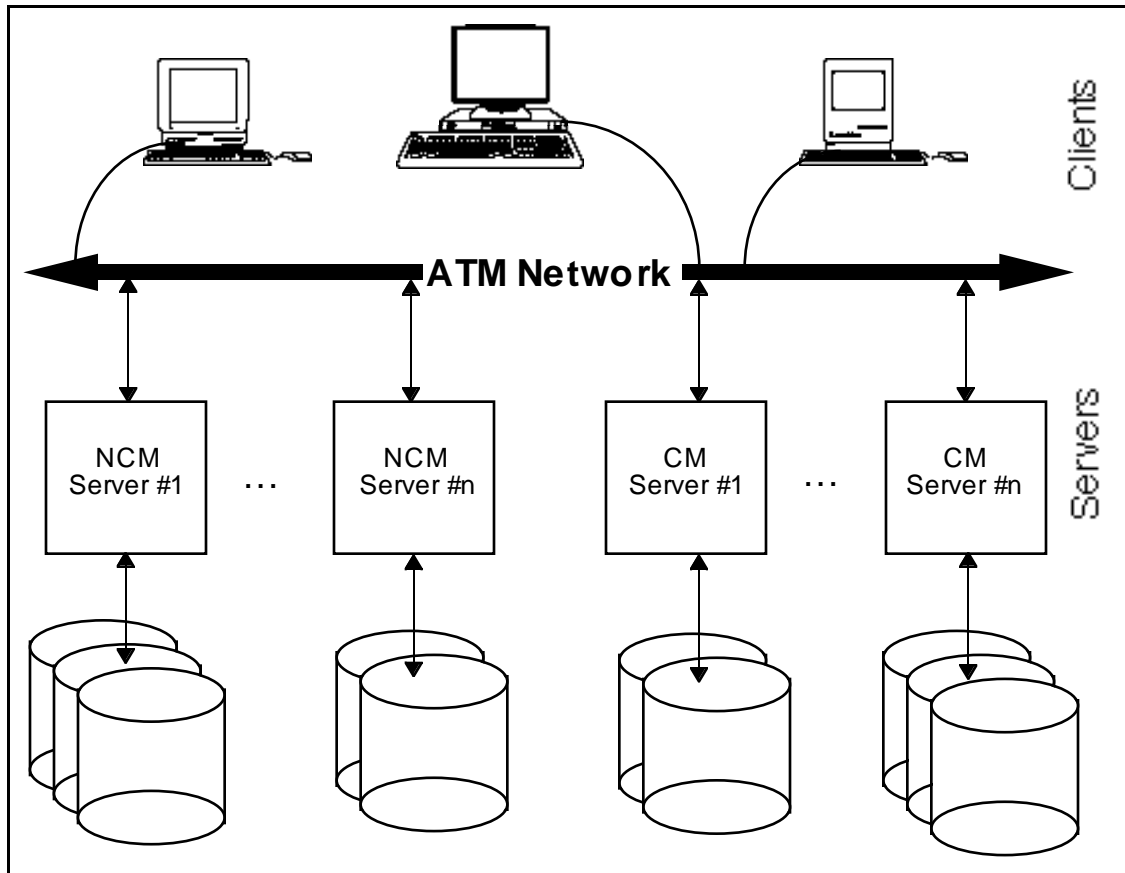
Figure 3. Database System Architecture

Briefly, the user chooses a document to display after having browsed the database through a *Visual Query Interface* [EÖSV95]. The user negotiates through the *Quality of Service Negotiator* [Hafi94] with the distributed system for the desired level of quality and cost of access. Then the *Synchronization* component [LG94] takes over by coordinating the delivery of several streams of monomedia data over the network. To do this, it has to request the *Continuous Media File Server* [NY94] to retrieve the appropriate files and start the streams.

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 it's task. To do this, it queries the database for the first two pieces of information. The synchronization

component needs to know the constraints between the media types in the document in order to determine the schedule of delivery of data over the network. The continuous file server needs to know the names and location of the files it is supposed to retrieve. The database stores the so-called Universal Object Identifier (UOI) for each monomedia object which the continuous media server uses to determine the name and location of the file(s) containing the object.

2.6 Modeling of News-on-Demand Documents

The discussion on the sample multimedia document revealed the three layered view of the document. In addition, the document possesses hyperlinks to other documents and document components. Storage of meta-information and architecture independence are issues arising out of the application requirements. In modeling these requirements there are three issues which need to be addressed:

- The different media components of the document (i.e., text, image, audio, and video) need to be represented in the database. These are called *monomedia objects* and their representation in the database is critical for good performance.

- The second issue that needs to be considered is the representation of the document 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.

- In hypermedia documents, one has to deal with the representation of the spatio-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).

The following three sections presents our approach to addressing these issues. Section 3 presents the type system which is used to model and represent monomedia objects. Section 4 deals with type system used to model document structure and of necessity, some SGML concepts. Section 5 deals with presentation issues, including a discussion of HyTime. The type system is expanded to include HyTime representations of spatio-temporal relationships and link structures. Section 6 describes the composition hierarchy of the document, illustrating how the type system gets instantiated.

11

3. 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. We have not yet started research on these issues. Content-based indexing of images is a problem that we have recently started to investigate. The challenge is to combine indexing techniques with standard methods for storing these media. Since the continuous media file server is not yet integrated with the multimedia database, we only store descriptive information about audio and video in the database.

As mentioned before, the object oriented database chosen for implementation does not provide native support for multimedia data other than text (or strings). These data types are what we call 'atomic types'.

3.1 The Type System for Atomic Types

Figure 4 illustrates the type hierarchy for atomic types. The full description of these types is given in Appendix 3. Instances of atomic types hold the raw (mono) media representation along with other information relevant to the QoS scheduler and synchronization module. Since there is not support for the monomedia types from the database (except for text), we use arrays of characters to store these media.

The continuous media is on a file server not under the control of the DBMS. To access these media files, the database is queried for the list of continuous media type instances in the document. These instances store the location of the file, the host on which the file exists, and also the Universal Object Identifier used by the file server to determine other location information. There is no need to do this for the non-continuous media data types which are managed by the DBMS software.
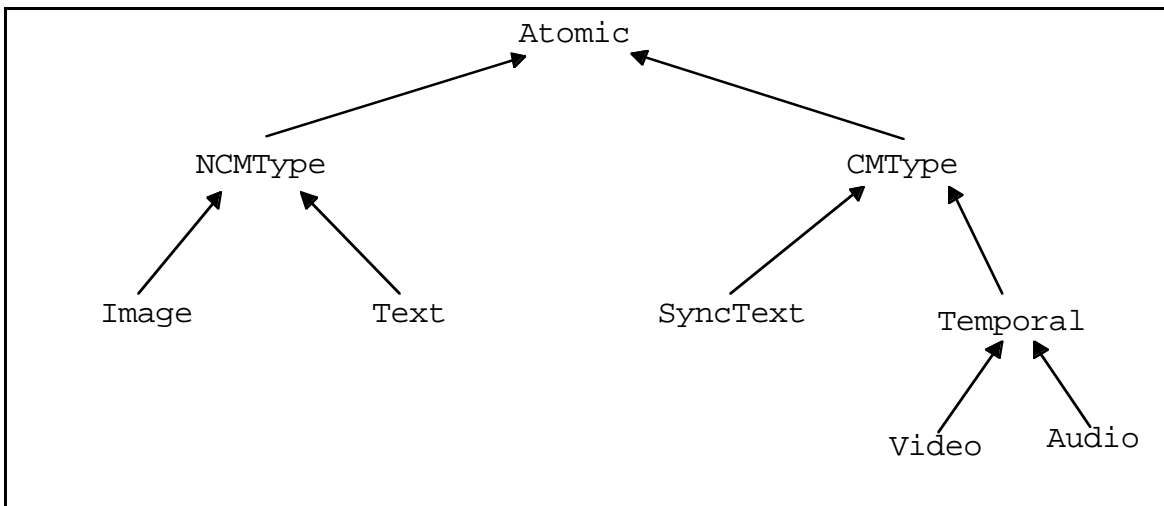
Figure 4. Atomic Types Hierarchy

12

Accordingly, there are two subtypes of atomic media types one for non-continuous media (`NCMType`) and another for continuous media (`CMType`). The attributes and methods which are relevant to both classes of media are abstracted in the `Atomic` type. These are the `length` and generic QoS parameters such as `jitter`, `cost` and `delay` [Hafi94].

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 colors of the image. Both these types have attributes for the QoS parameters specific to the media they model. The `Image` type is further subtyped to reflect the different storage formats possible which entails more attributes.

A similar subtyping scheme is seen on the `CMType` side of the type hierarchy. The `Video` type is subtyped to handle different storage formats. Synchronized text (`SyncText`) is not subtyped from the `NCMType Text`, since it is stored as a file, not as an object in the database. The methods `match`, and `substring` cannot be applied to the synchronized text media since the file is located on a server which may or may not be accessible to the DBMS. The `Temporal` supertype of video and audio is due to the fact that both have a `duration` in the time axis.

3.2 Storage Model for Text

`Text` (a character string) is an atomic type which is supported in the database system. However, in the news articles, the text component of the article is richly structured; i.e., consists of many components (also called *elements*), hierarchically arranged. 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.

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 (Figure 2), we need three fragments – 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 involved, 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 re-assemble the entire text component of the document which 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 residing 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 `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, we store the entire text content as a single string. To associate a particular instance of an element with its text content we store the first and last character locations of that portion of text in the entire text content. We call pairs of integers such as these, **annotations**.

Using this model the text content of the sample news document is :

Queen opens GamesVictoria, Aug. 18. The Queen today officially inaugurated the 15th Commonwealth Games in Victoria, B.C., at the University of Victoria. The Centennial Stadium was filled to its capacity of 30,000. Queen Elizabeth IISome 3500 athletes from 64 countries are competing in track and field events, swimming, gymnastics, lawn bowling etc. Lacrosse is being introduced as an exhibition sport in the Games.

In this example, the second paragraph instance has the annotation [235, 420]. The subelements of the paragraph have the annotations [258, 272] (`link`), and [352, 360] (`emphasis`). The graphical representation of the annotations approach is depicted in Figure 5.

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, and 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. We map this representation to a type system by defining a type, `Text`, whose instances store entire text documents as represented in Figure 5. We also define a type to correspond to every allowable annotation, as specified in the document DTD.

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

- Displaying the text becomes faster, and more efficient because multiple accesses to persistent store are avoided.

Link 2

Begin

End

Emphasis

Begin

End

Queen opens GamesVictoria, Aug. 18. The Queen today officially inaugurated the 15th Commonwealth Games in Victoria, B.C., at the University of Victoria. The Centennial Stadium was filled to it's capacity of 30,000. Queen Elizabeth II some 3500 athletes from 64 countries are competing in track and field events, swimming, gymnastics, lawn bowling etc. Lacrosse is being introduced as an exhibition sport in the Games.
C. Dickens

Figure

Begin

End

Paragraph 2

Begin

End

Figure 5. Annotations to Mark-Up Text Documents

- Indexes can be built on these annotation objects which can aid searches for element instances. For example, it is possible to search for all empha- sized text 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 the values of annotations to change. This can be avoided to a certain extent by specifying annotations relative to some enclosing structure, say with respect to a paragraph. Then the annotations of the paragraphs change after updates but the annotations of the paragraph's sub-elements remain safe.

4. MODELING DOCUMENT STRUCTURE

The logical structure of the document and the presentation information associated with it are necessary for the contents of a document to be understood. Certain queries (cf. Section 2.4), and the presentation of the document also rely on the logical structure of the document. Hyperlinks are made to logical document components.

Document representation formats store one or both of these pieces of information. Markups are used frequently to represent this information. Descriptive markups are used to mark off logical components, and also represent presentation information. As an example, the popular document formatter, LaTeX uses `\section` to delimit sections and `\bf` to indicate bold font. Other proprietary document representation formats such as Postscript and RTF only use markups to represent presentation information; the logical structure is not stored. The markup in LaTeX is also descriptive only; there are no means of specifying the hierarchical relationships between document elements (for example that subsections can only occur within sections).

The *International Organization for Standardization* (ISO) has published two standards for document representation – *Office document Architecture (ODA) and Interchange Format*, and the *Standard Generalized Markup Language (SGML)* (numbers ISO 8613 and ISO 8879 respectively). Both representations are used to model document structure. Both allow internal storage formats, and are architecture independent. However, SGML is especially attractive for the following reasons:

> "SGML can be used for publishing in its broadest definition, ranging from single medium conventional publishing to multimedia database publishing. SGML can also be used in office document processing where the benefits of human readability and interchange with publishing systems are required." [ISO86]

In addition, the ISO standard for hypermedia representation, called *HyTime* (ISO 10744), uses SGML representation syntax and is 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. HyTime also provides a way to represent scheduling and rendering information.

One of the reasons for the success of the World Wide Web is the use of a standard *Hypertext Markup Language* (HTML) for document representation. HTML is an *application* of SGML, and therefore a subset 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].

SGML describes the logical structure of the document by using *markups* to mark the boundaries of logical elements of the document. Although SGML is called a language, it is more of a *meta-language*, which is the key to its flexibility.

4.1 SGML Principles

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, etc. This is known as *procedural markup* in which the presentation information was mixed with document structure and document. In the following description, bold fonts denote SGML terms; italicized words are important concepts which are not restricted to SGML.

The *generalized markup* [ISO86] 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 elements (e.g., paragraph, section, section title, etc.) 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>` Queen Opens Games `</hdline>`

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

F   the markup in SGML is *rigorous* [ISO86] in that elements can contain other elements to form a hierarchy. Thus, `chapter` elements can contain `title` and `section` elements; `section` elements can contain `paragraph` elements and so on; the instances form a tree, enabling manipulation of subtrees. In other words, an SGML document consists of instances of document elements arranged in a hierarchical structure.

SGML does not specify what these elements should be, or their hierarchy. 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. In our case, we need to write a DTD for news articles, but there could be DTDs for books, letters, technical manuals etc. Therefore, each DTD defines a new markup language with which documents belonging to the category of documents the DTD defines can be marked up. In this sense, SGML is a meta-language.

4.2 Document Type Declaration

A DTD specifies *element types*, the hierarchical relationships among elements, and *attributes* associated with them. Attributes contain information that is not part of the document content. For example, if there is a `security level` attribute associated with `chapter` elements, then users without the same level of security clearance could be denied access to the subtree starting at that instance of a `chapter` element. In the example multimedia news document of Figure 2, the following element types can be identified: article, headline, location, date, paragraph, figure, figure caption, emphasis, author, linkarticle, headline, location, date, paragraph, figure, figure caption, emphasis, author, link.

Note that the article itself is considered as an element. There are other elements possible, such as `list`, `quote`, `table`, `emphasis2` (for a different type of emphasis), `keywords`, `source`, etc., which we won't consider, except for `keywords`. If we omit the audio and video elements, the marked-up example news article looks like the following:

```
<article>
<hdline> Queen Opens Games </hdline>
<keywords> Queen Elizabeth, Royalty, Sports </keywords>
<date> 18/8/94 </date>
<location> Victoria, B.C. </location>
<paragraph>The Queen today officially inaugurated the 15th
<link linkend=cwealth.sgml>Commonwealth Games </link>
in Victoria, B.C., at the University of Victoria. The Centennial Sta-
dium was filled to it's capacity of 30,000.
</paragraph>
<figure filename=queen.gif>
<figcaption>Queen Elizabeth II</figcaption>
</figure>
<paragraph> Some 3500 athletes from
<link linkend=list.sgml>64 countries </link>
are competing in track and field events, swimming, gymnastics, lawn
bowling etc. <emphasis> Lacrosse </emphasis> is being intro-
duced as an exhibition sport.
</paragraph>
</article>
```

Note that the start tags are ended by their corresponding end tags. Also, there are certain elements which may not be displayed, such as the `keywords` element. From this example markup, we could conclude that a news article consists of a headline, followed by a date, followed by keywords, followed by the authors name, followed by paragraphs, interspersed with figures. The other elements – links and emphasis always occur within other elements. Therefore, our first entry into the DTD would be :

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

This definition means precisely what we just described. The data on the left side of the dashes names the element, the right side defines its **content model.** The **connector ','** implies 'followed by', the connector '*' denotes zero or more occurrences, and '|' stands for 'or'. Note that the content model breaks up neatly into two components: the front matter, and the body of the article. Therefore, we rewrite the above declaration to look like:

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

The markup in the document should be changed to reflect this extra level of hierarchy.

Next, we try to define the content models of the elements occurring on the right hand side of these element declarations. Note that the elements `keywords, hdline, date, emphasis, link,` and `author` do not have other elements within them; they only contain text strings. In SGML syntax, these strings are called `#PCDATA`. This means that these elements have identical content models, and only the right hand side of the element type declaration will differ. If we combine all these declarations into one, we get :

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

In the sample document, `paragraph` elements contain strings of text interspersed with `emphasis` and `link` elements. Both of these subelements are optional. The `figure` element contains one subelement, the figure caption element, `figcaption`. The `figcaption` element just contains text strings, or `#PCDATA`. We assume this subelement to be always present. Using these inferences and assumptions, we come up with :

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

The second 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 of a slightly different nature. The additional information provides values for attributes of these elements. Thus, the markup

```
<link linkend=cwealth.sgm> Commonwealth Games</link>
```

implies that the `link` element has an attribute called `linkend`, whose value is `cwealth.sgml` in this particular instance of a link type. 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 (string in this case). The `#REQUIRED` means that any instance of the `link` element *has* to have a value associated with the `linkend` attribute. Other options are `#IMPLIED` (attribute need not be instantiated), and `#FIXED` (attribute can be instantiated to only one value).

In our DTD, we chose to make `figcaption` a separate element, rather than an attribute of the element type `figure`. This enables a search on the collection of all figure captions. This also allows us to redefine the `figcaption` element to have a more complex content model, in future versions of the DTD.

There is a third type of entry in the DTD, called *entity*. Entities are useful to include files and non-SGML data in the document and also as short forms for frequently used content models in the DTD. We do not consider entities in our design. We assume that the SGML document to be inserted does not contain any ENTITY entries.

To summarize, the three types of entries in a DTD are element type declarations, attribute lists, and entities. There are several optional features of SGML not mentioned here; they will not be used. Attributes and entities are features used extensively by HyTime, the hypermedia markup language. HyTime is discussed in the next section. The DTD 'design' process outlined here is only a toy example – it is meaningless to try and derive a description for a whole category of documents from a single instance. However, since we are designing the application from scratch, without any extensive set of real documents to analyze, we have to start with a concept of a multimedia news article and design our DTD based on this. The complete DTD based on this concept is presented in Appendix 1.

The DTD is our main basis for our logical structure that models the database design for multimedia news documents. The DTD does not (yet) give any presentation information. While the philosophy of SGML is to leave out the presentation information, the HyTime markup language allows us to specify temporal and spatial relationships between document components. This is described in Section 5.

Each element type in the DTD (including HyTime elements) is modeled as a type in our type hierarchy. The type hierarchy then incorporates the logical structure and presentation information. It should be possible to map the instances of all element types to their content[2]. The contents are 'primitive' data types, or atomic types.

---

[2]The elements with EMPTY content models (for example HyTime elements conforming to the `axis` AF) of course may not have any data content associated with them. If they have attrib-

4.3 Type System for Elements

The type system includes over fifty types for various document element types (including HyTime elements, 'normal' elements and style sheet elements), and their supertypes. Although the type system was built up more or less bottom up, a top down description of the inheritance hierarchy will make things clearer.

4.3.1 Element Types

Figures 6 to 8 show the type hierarchies for logical document elements. The supertype of all elements is the `Element type`. This models the fact that *all* elements need to maintain a reference to their parent element in the document instance hierarchy, so that the hierarchy can be navigated starting from any element. When links are made to arbitrary elements in different documents, or when searches are performed over several documents, it is often useful to know the article these element instances belong to. Therefore, all elements maintain references to the article they belong to. `Element` is subtyped into `TextElement, Structured` and `HyElement`.

Looking at the DTD for news articles, we note that the article has been divided into two components – one called `async`, and the other called `sync`.
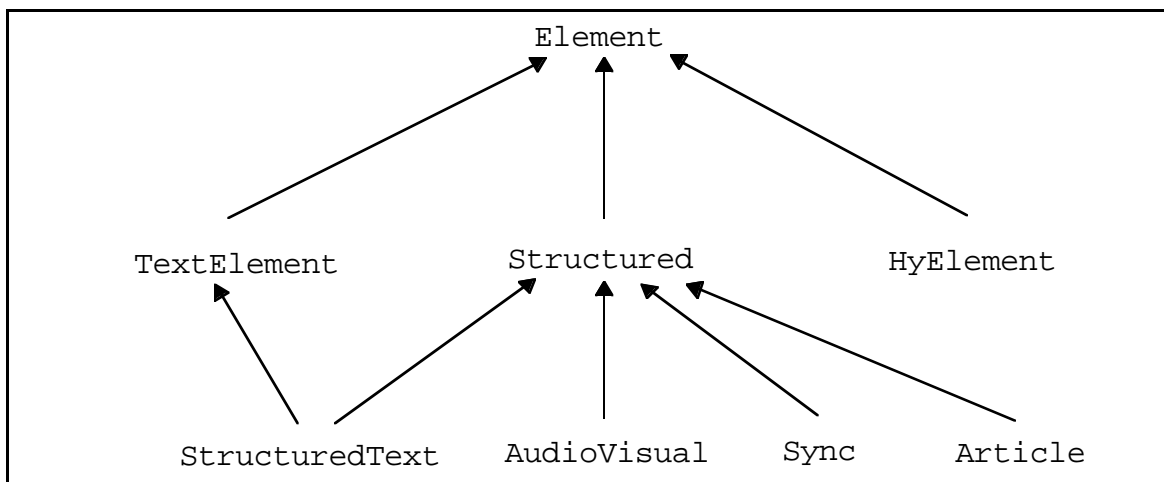
Figure 6. Simplified Element Type Hierarchy

This reflects the fact that continuous media (i.e., media which have synchronization constraints need to be handled by HyTime elements, and normal SGML elements are adequate to deal with text and image data. The supertype `HyElement` encompasses all the HyTime elements used in the DTD. We delay a discussion of these types to until after we have described the HyTime standard in Section 5.

---

utes which specify their content, then it is possible to map them to the instances of the atomic media types (cf. Section 6.3).

Due to the annotation based storage model, elements defined for textual data in the DTD have an attribute whose value is the annotation of the element in the article instance. Their supertype is the `TextElement` type. This supertype has methods to manipulate these annotation values. The other method is `getString` which returns the string value captured by the annotation. The type hierarchy (other than `StructuredText` which is described in Section 4.3.3) rooted at this type is illustrated in Figure 8, and discussed in Section 4.3.4.

4.3.2 Structured Elements

The type `Structured` is a supertype for elements in the DTD with complex content models. By complex we mean, not `EMPTY`, or (`#PCDATA`). That is, structured elements can have child elements in the document instance and need to maintain references to them. Correspondingly these elements have the method `getNth` which returns a reference to their nth child element. Since the types of these child elements are so diverse, the only common supertype is `Element`, which is the return type of this method. This method gets redefined by the subtypes so that the return type becomes more specific.

Elements which are both structured and based on text have a supertype `StructuredText`. The subtypes of this type includes all text elements with complex content models, like `list`, `section`, `figure`, `frontmatter`, etc. The type system rooted at this type is described in Section 4.3.3 and shown in Figure 7.

Instances of the `Article` type are at the root of the composition hierarchy. According to the DTD, they should have references to instances of the `Frontmatter`, `Async` and `Sync` types. In addition, the *date, source, subject,* and *author* are attributes (type `String`) of `Article`. Although these values are already stored (by means of annotations) as instances of `Date`, `Source`, `Subject`, and `Author` types respectively (and are child elements of instances of `Edinfo`). We would like to index the collection of `Article` instances on the values of these attributes, since queries predicated on these are likely to be frequent [EÖSV95]. ObjectStore collections can only be indexed on attributes. The string value of instances of `Date`, `Source`, `Subject`, and `Author` can only be obtained by the application of the method `getString()`. Hence, although we could have methods `getDate`, `getSource`, `getSubject`, and `getAuthor` for the `Article` type, it would not have been possible to build indices on these methods.

`AudioVisual` and `Sync` are the other two subtypes of `Structured`. In the DTD, the element `audio-visual` models one set of logically related HyTime elements such as `axis`, `fcs`, `extlist`, etc. For instance, if the document was one hour of a television broadcast, there would be one `audio-visual` each for the news, for the commercial segments, etc. The whole broadcast would be modeled by the `sync` element, and captured by the `Sync` type. `Sync` instances are collections of `AudioVisual` instances.

Structured elements have complex content models and pose peculiar problems while modeling the content models. The first problem is due to the 'or' connector ('|')in the content model. For example, the `Async` element has the content model:

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

If we have three fields 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, then we lose the relative orderings between say, `Section` instances and `Figure` instances which are the children of the `Async` instance. The other solution is to have just one list of references of the common supertype of `Section, Figure, and Link`; this is `StructuredElement` in this case. This leads to type checking problems since even references to `Paragraph` elements can now be inserted into the list.

The 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; 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., the user has to do some type checking). The solution we adopt is to create an abstract supertype of `Section, Figure,` and `Link`. The parameter of the list is then this supertype. There are no type checking problems now. The drawback is that it creates an explosion of types in the system. We call abstract supertypes created for this purpose *pseudo-union* types.

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 in our type system, we need a mechanism to order the attributes of the type `Frontmatter`. Again, this feature is not present in the data model of ObjectStore. We assume an implicit ordering of attributes in this case. The behavior of the Frontmatter type is such that it enforces the ordering. Thus, when the method `getNth(3)` is applied to an instance of `Frontmatter`, the result is a reference to an instance of the type `Subhdline`.

4.3.3 Structured Text Elements

Instances of `StructuredText` type are text elements (i.e. those whose content is represented by annotations) which have subelements. There are ten such elements, and since their child elements are so diverse in their types, the return type of the `getNth` method is `TextElement`.

Figure 7 shows the hierarchy. The subtypes of `StructuredText` redefine the return type of the `getNth` method, and also have additional methods.
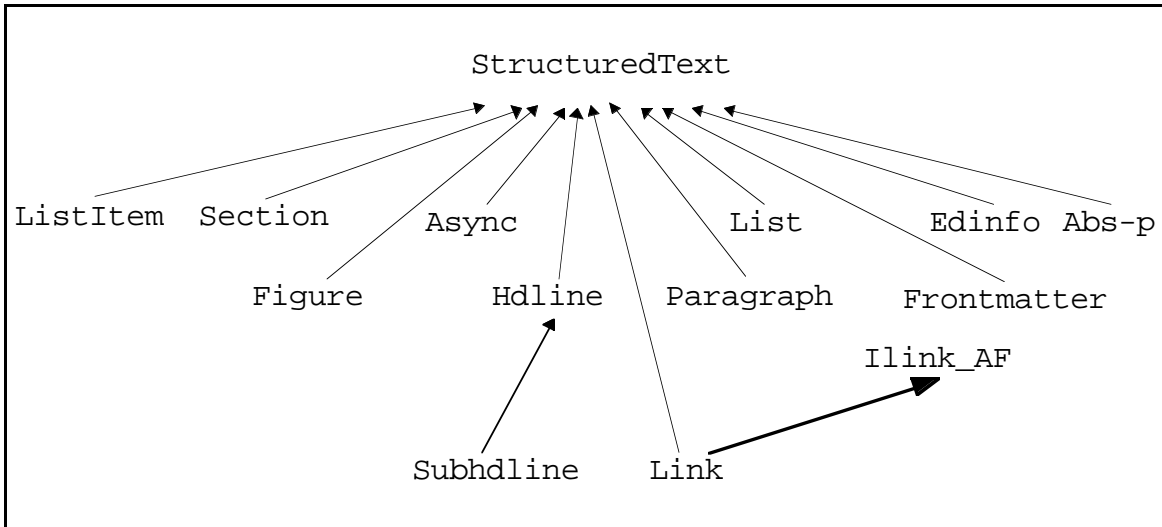
Figure 7. Type System for Structured Text Elements

For example, the `List` type has methods to get and set its title. The type `Section` has the same method (and so the two have a common abstract supertype, which is not shown here).

The type `Figure` illustrates the relationship between atomic types and element types. One of the attributes of `Figure` is `Image`, which is a reference to an instance of a subtype of the type `Image`. The other attribute of `Figure` is `float,` a string valued attribute which indicates the position at which the image has to be displayed relative to the text.

The type `Paragraph` has a `children` attribute which is a list of the subelements of `paragraph` in the DTD. These are emphasis elements, list elements, figure elements, link elements and quotes. Paragraphs can also contain plain text not wrapped in any other element. Therefore the parameter of the child list is the pseudo-union type formed by creating the supertype of `Emphasis`, `List`, `Figure`, `Link`, `String`, and `Quote` types. Similarly, the `Async` type has a child list as an attribute which is a list whose parameter is the pseudo-union supertype of `Section`, `Figure`, and `Link` types. The `Hdline` type is quite similar to `Paragraph` in that the parameter of the list of children is `Emphasis`, `Quote`, `Link`, and `String`. `Hdline` is subtyped to `Subhdline`; this is a classification step only.

`Abs-p` is the type whose instances store the abstract paragraph of the article – this is just one paragraph, according to the DTD. The `Section` type has a `title` attribute; the other children are a list whose parameter is the pseudo-union supertype of `Paragraph` and `List`. The `Edinfo` type which models the editorial information of the article has one attribute each for the location (`Loc`), date (`Date`), source (`Source`), author list (list of `Author`), keywords (`Keywords`), and subject (`Subject`). The `Frontmatter` type models the headlines and other matter usually found at the beginning of news articles. It has

24

as its children `Edinfo`, `Hdline`, `Subhdline`, and `Abs-P` element types – these are the attributes of `Frontmatter`. We delay the discussion of `Link` to the section on HyTime element types.

4.3.4 Other Text Elements

The other text elements consist of text only with no subelements. Most of the types here do not have any additional methods or attributes other than those present in `TextElement`; they have been created as subtypes purely as a classification step, and to retain the uniform approach of modeling all element types as types in the type system. Figure 8 illustrates the hierarchy.

The exceptions are `quote`, `author` and `date` which have additional methods and attributes, as a result of additional attributes being defined in the DTD. The attributes of these elements and the attribute list found in the DTD have a one-to-one mapping; `Quote` has a `source` attribute, `Author` has a `designation` attribute, `Date` has integer attributes for the day, month and year components.
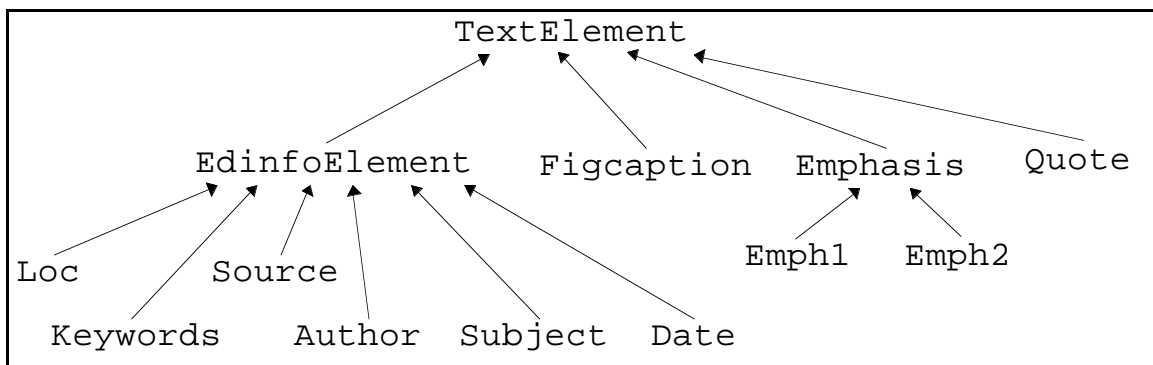
Figure 8. Type Hierarchy for Other Text Elements

25

5. PRESENTATION INFORMATION

This section first presents an overview of HyTime concepts and how they can be used to represent links and temporal/spatial information. Then the approach to representing presentation information for SGML elements is discussed.

5.1 HyTime Overview

The formal name for HyTime is *ISO/IEC 10744: Hypermedia/Time–Based Structuring Language.* This indicates that HyTime is based on (a) links, (b) document structure, and (c) representation of temporal information. For the document structure aspect, HyTime simply uses SGML representation of document structure. This means that HyTime uses DTDs to represent document categories. The other two features are discussed in the following subsections. First, we define a few important HyTime concepts.

5.1.1 Architectural Forms

The fact that HyTime also uses DTDs to represent document categories leads us to the idea that we could define one catch-all DTD 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 (recall that SGML is a *meta*–language, and DTDs define languages), plus the semantics for the special elements, would be our hypermedia 'language'. This approach is too restrictive for a number of reasons – people 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.

The solution adopted by HyTime is to call these special elements **architectural forms (AF)**. 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 our HyTime document instances, we first define element types which **conform** to the specification of the architectural form. Then we use instances of these conforming element types. If we want to use the clink architectural form (AF) as the link element in our news article DTD, we would have the following declarations :

```
<!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 H   parser that the element is supposed to conform to the `clink` archi-

tectural form. To conform to an AF, an element declaration (or instance) should 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 our `link` element should also define that attribute in our DTD.

The list of architectural forms is a major portion of the HyTime standard – there are 69 of them. This list forms a *meta–DTD* for HyTime. The architectural forms are *abstract* elements types which get instantiated to *concrete* element types in the DTD [DD94]. The **meta-declarations** in the HyTime standards also include the **meta-content** models. Conforming element instances have to adhere to the meta-content model of the AF to which they conform.

5.1.2 HyTime Modules

The HyTime standard is divided into **modules**, each of which describes a group of concepts and architectural forms. These modules are:

- <u>Base Module</u> which describes some basic concepts in HyTime, including the concept of Architectural Forms. A few AFs are also defined.

- <u>Measurement Module</u> which describes the basic units of measurement in time and space and other domains.

- <u>Location Address Module</u> which gives architectural forms for locating pieces of data (so that they can be linked from and to). It uses some features of the measurement module.

- <u>Hyperlinks Module</u> which gives two architectural forms for the definition and processing of hyperlinks.

- <u>Scheduling Module</u> which defines some architectural forms used to schedule events in space and time.

- <u>Rendition Module</u> which defines AF's used to represent presentational information about HyTime documents.

Each module may use certain features of other modules lower down in the hierarchy; thus the location address module does define AF's which use AF's defined in the rendition module. Each HyTime DTD would declare the names of the modules it requires support for.

In our DTD for news articles, we use certain features of the base module (as do all HyTime documents), some of the location address module, some of the hyperlinks module, and some of the scheduling module. We skip the description of all these modules, except for the scheduling module. Concepts needed from other modules will be defined where required.

5.2 Finite Coordinate Spaces

To represent relatively simple spatial and temporal constraints between document elements, we use the finite coordinate space (`fcs`) architectural form defined in the scheduling module. 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 abstracted. 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.

5.2.1 HyTime Measurements

HyTime measurements are expressed in integer terms, and they are made between fixed maximum and minimum values. 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. For example, if we create an axis with character quanta for the sentence: "The Queen today officially inaugurated", then the following markup, adapted from [DD94] (using various architectural forms), picks out the word 'Queen'.

```
<my_dimspec>
<my_marklist>5 5 </my_marklist>
</my_dimspec>
```

The dimspec AF allow us to specify a dimension or a range of quanta. The 5th quantum on the axis, is the character 'Q', counting five quanta starting from this point gives the word 'Queen'.

5.2.2 Axes and Finite Coordinate Spaces

HyTime models space and time using axes of finite dimensions. A *finite coordinate space* is a set of such axes. 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" >
```

Recall that axes are measured in integer terms in quanta. The quantum here 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 9 describes the various concepts used. The fi-
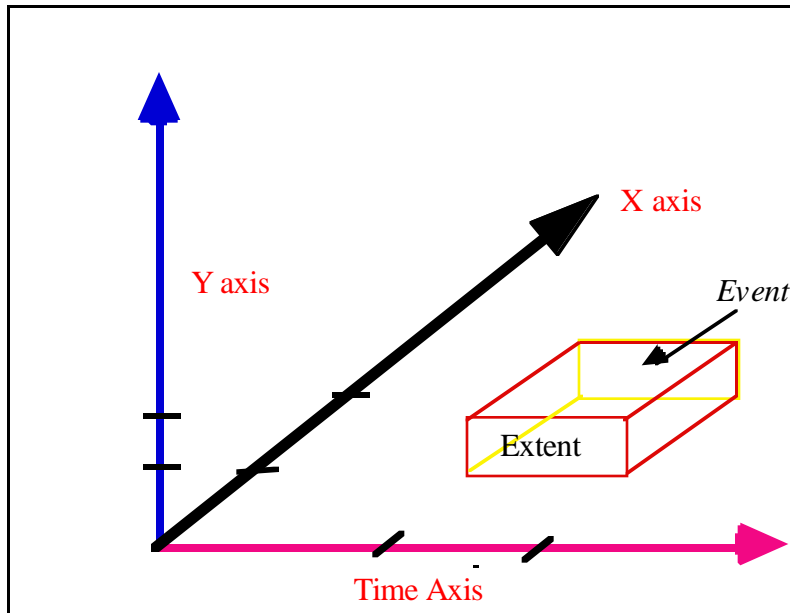
Figure 9. Axes, Events, and Extents (adapted from [DD94])

nite coordinate space shown here has three axes: two spatial, and one tempo-ral.

In HyTime, an *event* is simply an *extent* in the FCS. An extent is a set of ranges along the various axes defining the FCS.

Figure 9 shows the extents marked on all three axes for the event denoted by the box. 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 docu-m   ce associates a data object with the event. The semantics and the manner in which the events are rendered are defined by the application. The (meta) element type declarations 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, we 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 conform-ing to the `extlist` architectural form. These `extlist` elements give the ex-tents of the event along the axes of the FCS.

5.2.3 A DTD Fragment for Closed Captioned Video

In the sample document shown in Figure 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 the Queen's speech, along with French subtitles displayed at intervals at the bottom edge of the screen. We call this closed captioned video (CC Video). Using various HyTime architectural forms, we now write the DTD fragment corresponding to the CC Video concept.

In the CC Video document, we have three types of events, which roughly correspond to the three types of media present – audio, video, and (synchronized) text. Figure 10 shows only the time axis to display the extents of these events, for the first 65 seconds of the presentation. We see that there are five events of type text (because we assume the number of subtitles to be five, in the 65 seconds), and one each of the audio and video types. There are spatial extents also – we create two axes to represent the X and Y coordinates on the workstation screen (we presented our time axis in the last 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
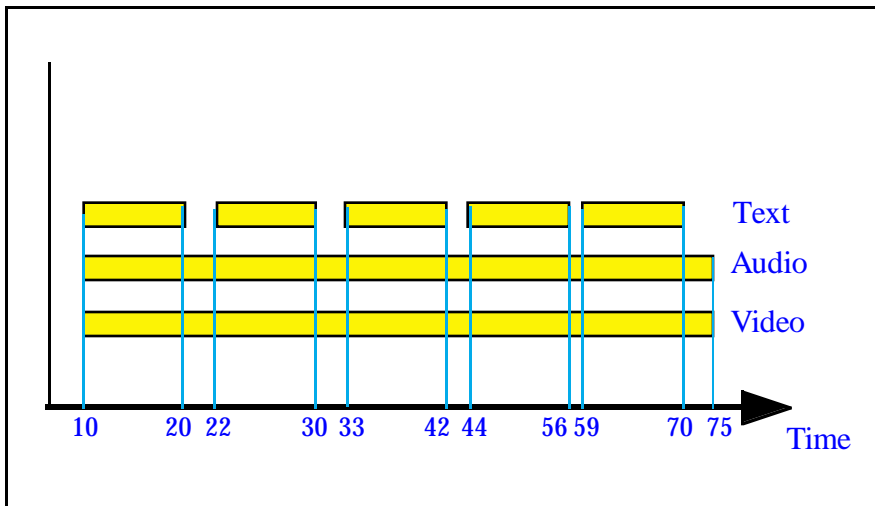
Figure 10. Extents Along the Time Axis for Events in CC Video

a workstation screen (assumed to be 1024 x 900).

As mentioned before, we have 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, we choose to associate a file with each event. This could be 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.

We define the event schedule which can represent the timeline shown in Figure 10, which consists of one audio, one video, and several text events:

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

For a complete DTD, including attribute lists, refer to the Appendix 1. What remains is the declaration of the FCS :

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

And finally, we declare our CC Video document (which we call `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.

5.3 Formatting Instructions

The SGML philosophy is to bind the processing instructions to the logical elements of the document as late as possible, i.e., only at formatting time. However, we need to include this information in our database. The list of associations between logical elements and their processing instructions is known as a *style sheet.* The alternatives are to use the LINK feature of SGML (quite different from hyperlinks), or the international standard being defined for specifying formatting instructions called DSSSL [Herw94]. This is incomplete, and therefore is not a candidate for representation in our database. Therefore, what we need to store is a representation of the style sheet.

To represent the style sheet in the database as a separate piece of information from the document instance we extend our type system to include a DTD for style sheets. An example DTD for a style sheet specification would be (adapted from [Gold90]) :

```
<!ELEMENT rule    - - (source, spec+)>
```

```
<!ELEMENT source  - - (#PCDATA) >
<!ELEMENT spec    - - (#PCDATA, value)>
<!ELEMENT value   - - (#PCDATA) >
```

An example instance of a style sheet [Gold90] would be :

```
<rule>
  <source>listitem</source>
    <spec>bullets<value>square</value> </spec>
   <spec>indent<value> 7 </value></spec>
   <spec>font<value> courier bold </value></spec>
</rule>
```

This specifies that the logical element list item should have square bullets marking the item, and should be indented by 7 spaces, and should be set in courier bold font.

It should be noted that style sheets are inadequate to specify the entire range of processing instructions. One example is context sensitive processing – the processing of an emphasis element may depend on whether it occurs in the abstract paragraph or in the main body of text. Another aspect is the layout of text – for example in two or three column formats. The first can be handled using the LINK option of SGML [Gold90]. For the second problem, we can associate this information as processing instructions for the root of the document instance tree; in this case the instance of an `article` element.

5.4 Type System for Presentation Information

HyTime uses SGML syntax for hypermedia representation. Since we use HyTime to model temporal and spatial information, the same concept of a document can be extended to include the presentation layout as well. To represent processing instructions, we have another category of documents – the DTD for style sheets. This too is a collection of elements with hierarchical relationships.

5.4.1 HyTime Elements

The type `HyElement` in Figure 6 is the supertype for all HyTime elements in the type system. Its immediate subtypes are those modeling the architectural forms used in the DTD. The attributes of `HyElement` are its ID (assigned by the author of the document, or by the document authoring software), and the string representing the name of the architectural form. This models the

---

[3]The elements with EMPTY content models (for example HyTime elements conforming to the `axis` AF) of course may not have any content associated with them. If they have attributes which specify their content, then it is possible to map them to the instances of the atomic media types (cf. Section 6.3).

assumption that every HyTime element can be linked to, and should answer the architectural form it conforms to. Appendix 3 contains a full description of the `HyElement` type system which is illustrated in Figure 11.

Of the nine HyTime architectural forms used in the DTD, the most important are the **fcs** and the **ilink** AFs. The **ilink** AF has a `%HyBrid;` content model, therefore it could be a `Structured` element depending upon the DTD designer. We create a type for this AF, called `Ilink_AF`, as a subtype of the `HyElement` type. In the DTD for news articles, the `link` element has a complex content model and conforms to the **ilink** AF. Therefore, the `Link` type is a subtype of both `Ilink_AF` and `Structured`. According to the HyTime standard, the **ilink** AF has to have the attributes `linkends` and `anchrole` (anchor role). The **ilink** AF can be used to specify multiple destinations per link, and can link any element to any other element. The `linkends` attribute is therefore a list of `Element` references. The anchor role is a `(#PCDATA)` valued attribute, and is therefore of the type `String`. The `Ilink` type has the pure virtual method `traverse` which takes the object ID of a destination element (present in the `linkends` attribute), and performs a traversal according to the applications semantics (hence `Ilink` is an abstract type, like most other types representing architectural forms). This method is defined in the `Link` subtype.
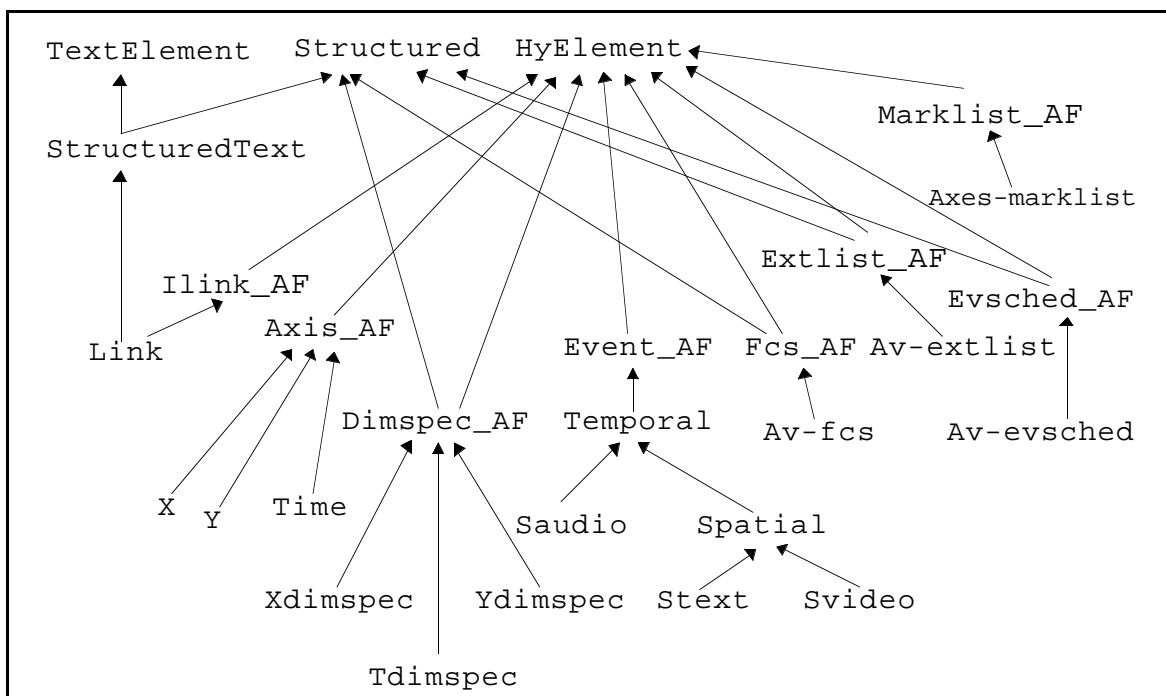

Figure 11. Type Hierarchy for HyTime Elements

The **fcs** element is important because it provides the interface to the other system components to (a) determine the types of media objects present in the

continuous media, (b) to determine the playout schedule of the media objects which are a part of the **fcs.** The attributes and methods of the `Av-fcs` type illustrate how this information can be obtained. It has a method `GetSchedule` which returns an object of type `TimeFlowGraph` which contains the schedule of the objects. The method `GetVideoObjects` returns a list of references to objects of type `Video` (an atomic type). These atomic objects can be queried for location and QoS information.

The other HyTime elements (Figure 11) are architectural forms used in the DTD. The **axis** architectural form has a `%HyBrid;` content model; the axes used in the DTD all have `EMPTY` content models, and hence do not have `Structured` as a supertype. All three axes (`x`, `y`, and `time`) declared in the DTD are similar, except for the dimensions, measurement units, and measurement granularity, which are reflected in the values of the `axisdim`, `axismeas`, and `axismdu` attributes. However they have different semantics in the DTD; thus they are separate subtypes of `Axis_AF`.

The **event** architectural form also has a '`%HyBrid;`' content model in HyTime, but the events in the DTD all have `EMPTY` content models. 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 `Temporal` reflect the fact that `Saudio` has a purely temporal dimension, 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` attribute reference the `Extlist` instances which hold the values of the extents of these elements along the three axes.

The **extlist** architectural form has the concrete element type `Av-extlist`. The children of this element are the three elements conforming to the **dimspec** architectural form. Therefore the `Av-extlist type` is a subtype of the `Structured` type. The three subtypes of `Dimspec_AF` (not shown in the diagram) are exactly the same, but are separate for classification purposes. They contain elements conforming to the **marklist** AF, and are hence `Structured` elements.

5.4.2 Other Elements

T   the style sheet DTD are shown in Figure 12. There are 7 elements in that DTD, of which only 3 are structured elements (`stylesheet`, `rule`, and `spec`). All the elements consist of strings. It is preferable not to use the annotation model to store these text elements. This is because the size of the style sheet is small (no large objects, and a few lines of text). Therefore the types modeling the elements of this DTD are either subtypes of `Structured,` or are direct subtypes of `Element`.

34
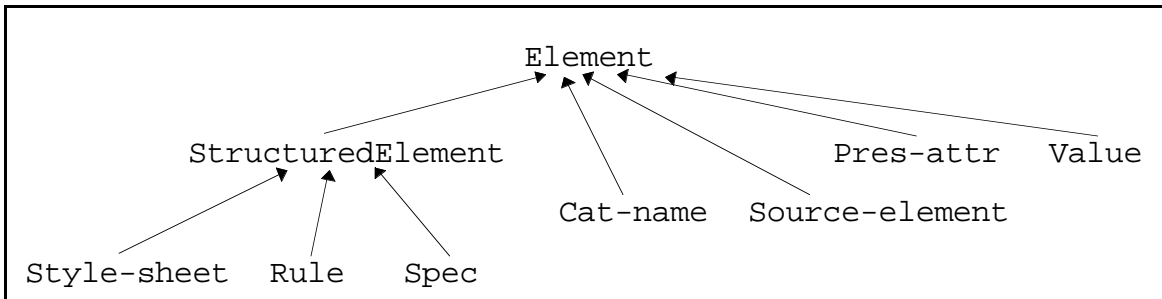
Figure 12. Style Sheet Element Types

5.5 Other Types in the System

When the concept of annotations representing text content was discussed, it was said that an object in the system held the lists of annotations present in a document instance. We call the type of this object 'article_root'. The attributes and methods of this type are described in the appendix.

Instances of article_root have references to the instance of the article type which is at the root of the hierarchy in the document instance. They also have a reference to the Text object which holds the string content of the article. When the user poses a query to the database to retrieve documents matching a certain keyword, a reference to the instance of the article_root is returned. This object has all the information needed to render the text portion of the document, and also can efficiently access the content of individual elements instances, for example keywords.

There is also a type to model annotations, called Annotation. This has the two integer location values as attributes, and methods to manipulate these values.

6. COMPOSITION HIERARCHY – AN EXAMPLE

The discussion in the previous section concentrates on the type system without looking at 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, we will demonstrate how the structure of the example document is mapped to a composition hierarchy as objects are instantiated and their attribute values set. This discussion refers to Figures 13 and 14, 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 13) 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, 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 1, each document is separated into sections first. In our example, we assume that the figure which consist of the Queen'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
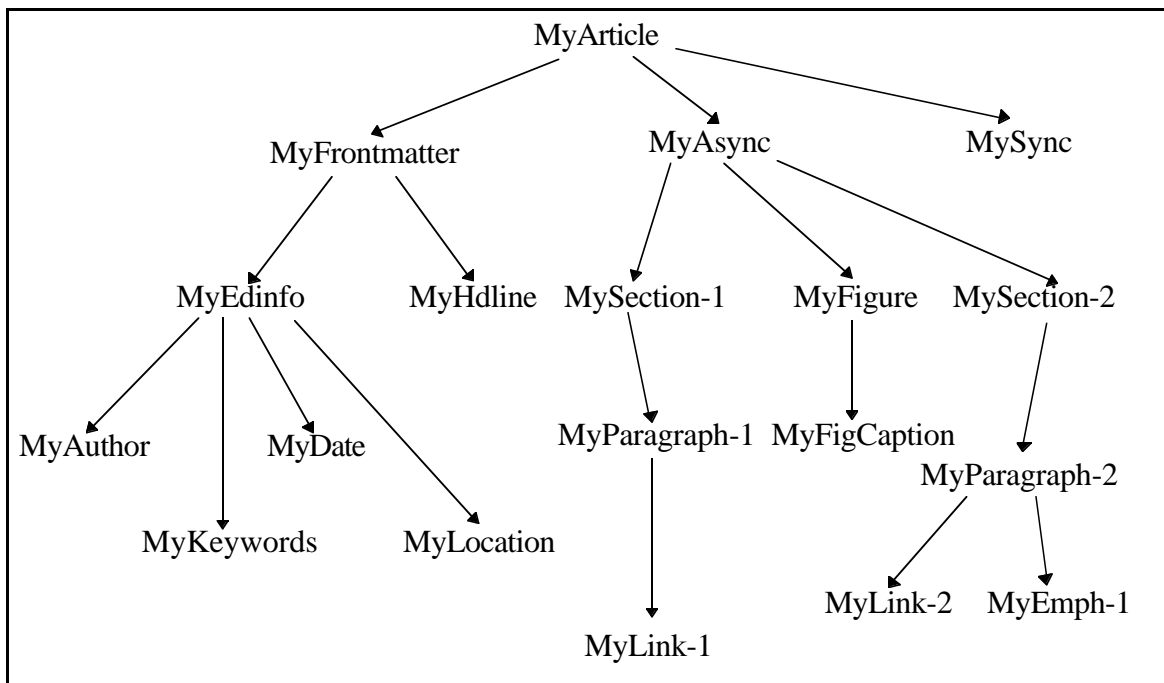
Figure 13. Partial Object Composition Hierarchy

36

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). For example, MyFigure has a link to an object of type `Image` (or one of its subtypes depending on the type of the Image) for the Queen's picture.

The synchronous part of the document that corresponds to the audio and video is shown in Figure 14. In the sample news document of Figure 2, it is assumed that a closed captioned video of the Queen's speech is associated with the article.
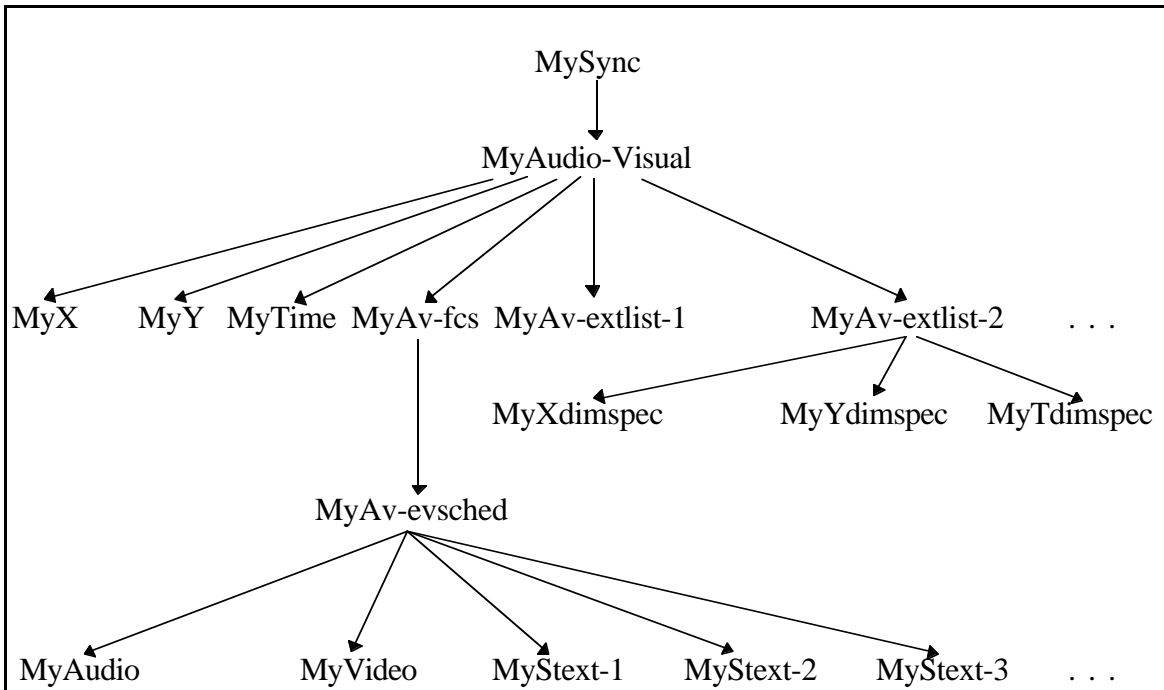
Figure 14. Composition Hierarchy for the Synchronous Portion of the Example Document

This consists of the video, synchronous with the speech (audio), along with captions which appear periodically, giving the French translation of the speech. 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 14, 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; for example if the speech 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.

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. We omit the marker list since it is too involved to display in one figure.

Not shown in the composition hierarchy is the occurrence of instances of atomic types. In Figure 13, MyFigure has a reference to an instance of `Image`. In Figure 14, 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`.

7. A VISUAL QUERYING FACILITY

Many of the tools that access multimedia information systems are based on browsing. In the case of hypermedia documents, these browsing tools may become sophisticated enough to allow navigation via links, playing of audio and video components, etc. Many tools ignore the equally important query facility which allows ad hoc querying of the multimedia news database. Our research deals with this aspect of hypermedia document access.

We are ultimately interested in the development of query languages, access primitives, and visual query facilities that would allow sophisticated querying of these databases, including content-based querying of all types of data. However, we are a long way from reaching that goal. What we have developed at this point is a visual query facility that allows elaborate searching of textual parts of documents and provides means for accessing other monomedia objects by means of keywords. The full description of this facility is given elsewhere [EÖSV95]; what we will provide here is a brief overview of a few important aspects relevant to the topic of this report.

An important aspect of the visual query facility is its tight integration with the OBMS. Each of the allowed functions has a corresponding ObjectStore query specification. Thus, each user request is translated into an invocation of a query on the database and the translation of the result to the visual form that is requested.

A second important feature is its adherence to the SGML/HyTime principle of separating the storage from the presentation. As mentioned earlier, the stored text, for example, may indicate that part of it is emphasized. The emphasized text can be displayed in various formats (e.g., italics, bold, reverse video). These presentation specifications are stored in the database as *style sheets* (cf. Section 5.3) which are then accessed by the visual query facility. Our current implementation is sophisticated in its handling of textual data, but relatively naive in dealing with continuous media (for example, we always open a new window for video with its own controls).To represent the style sheet in the database as a separate piece of information from the document instance, we extend our type system to include the DTD for style sheets.

We also separate certain presentational features, such as window size, control panel for various displays, etc., from the style sheet and consider these as the *user profile*. This is because styles are associated with individual logical elements, but there are other presentational features that are independent of the types of element being displayed. For example, one user may specify that clicking a video button in a document should start playing the video immediately while another user's preference may be to open up a video control window with VCR-type controls. These choices are made more frequently than the choices of styles, and may require a different storage model than the one adopted for the document content which assumes no updates to the content.

8. 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 involves (a) defining a model for multimedia documents, and (b) defining models for multimedia data. Documents, multimedia documents in particular, are richly structured. With the addition of hyperlinks (leading to *hypermedia*), the information capacity of the document is increased dramatically. Document models try to capture the structure of documents and 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, we have defined *Atomic* types 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. This structure is usually explicitly made visible by the document authoring system. Current technology is not sophisticated enough to do the same for other media types such as images, video, and audio. These structures involve spatial and temporal constraints. Nevertheless, it is foreseeable in the future that these media will be just as richly structured (if not more) as text.

The utility of object oriented database systems for hypermedia applications (*vis a vis* relational systems) is highlighted in [Bala94]. Perhaps the earliest object oriented approach is [WKL86] which discusses building a multimedia database on top of Orion. In [BCKL+94], the task of incorporating support for 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 different SGML Document Type Declarations (DTD)) are fields of a new data type called TEXT. Each TEXT field consists of the contiguous text content of the document along with the parse tree. Elements in the DTD can be part of the query; it is also possible to pose queries about the DTD itself. The EXPAND operator can be used to convert parts of the parse tree into fields of a relation. Updates to the TEXT field are not handled. The obvious drawbacks are listed in Section 1.1. Including multimedia data and HyTime elements would not be possible in this design. The approach of storing the text content contiguously and not fragmenting it is similar to our approach. However, we store the locations (or annotations) of the start and end of element instances; the parse tree is implicit in the composition hierarchy (cf. Section 6). Other approaches to object oriented models for multimedia data include [CAF+91].

A novel object oriented model for a video database is proposed in [OT93]. The model is *schemaless,* and includes *inheritance by inclusion* as a inherit 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. Incorporating *structured* video data will be a future extension to our design (through an extension to the DTD). However in the above model, 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?

Querying of SGML documents is also the focus of [CACS94], where extensions of two OBMS query languages are proposed. The paper highlights the issues associated with the object oriented modeling of SGML document structure. In particular, two extensions to the data model of an object oriented database system (O_2) are proposed. They are: (a) *ordered tuples*, or the ordering of attributes of a type, and (b) union types. The extensions to the query language are (a) the *contains* predicate to handle querying on strings, (b) *implicit selectors* to handle queries over union types, and (c) two new sorts to query text without exact knowledge of it's structure.

Types representing unstructured document elements 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/storage overhead. However, the authors believe that the improved access flexibility makes the new language particularly suitable for extensions to SGML, such as HyTime. Structured document elements have types associated with them; however there are no inheritance relationships shown between them. The issue of dynamic additions of new DTDs to the schema is not addressed here.

Our approach to handling union types is described in Section 4.3.2. The ordering of attributes is visible through the behavior of the types. We don't handle queries with inexact knowledge of the document structure in our model. Querying on strings is handled by the method `match` of the `Text` type. Since we do not implement union types in our system, we do not deal with selectors (queries predicated on the discriminant method). The inheritance relationships described in Section 4.3 and Section 5.4, and the composition hierarchies described in Section 6 illustrate our approach: every element is an `Element`, and may have an instance of an atomic type. If the element is a `TextElement`, then it's content is described by it's `annotation` attribute.

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, and enables queries on the structure of the document. The paper does not describe the querying facilities, but describes in detail how dynamic DTD handling is implemented by means of meta-classes.

The design follows a layered approach by separating out the DTD specific features and DTD independent features into two separate layers of classes[4]. Document type-specific classes are *specializations* of the document type- inde-

---

[4] There is a third layer, the HyTime layer, which is under development.

41

pendent classes. This means that features present in all SGML documents (methods to navigate the document tree for example) can be abstracted out in the classes of the document type-independent layer. Furthermore, there are two meta-classes: *terminal* and *nonterminal*. Classes in the document type-specific layer are instances of either of these meta-classes. The nonterminal class has a method to create a new document-specific class at run time. The content model of the new class can be set using another method. Finally, instances of the document type-specific class can be created at run time using the inherited method `createElem()`. In this manner, DTDs can be dynamically created and inserted into the database.

Our approach has similarities to this design. The supertypes `TextElement`, `Structured`, and `StructuredText` can be said to be document type-independent types. We have not considered dynamic additions of DTDs to the database in our design (although ObjectStore has metatypes in it's data model and allows the dynamic addition of classes to the ObjectStore database schema).

The implementation of a persistent object oriented system for HyTime documents forms part of the paper [KRRK94]. The database (implemented on ObjectStore) forms part of a HyTime engine 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 again 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 documents. 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) used 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.

The types representing HyTime elements in our design are all subtypes of `HyElement`. This could be considered to be a separate 'HyTime Layer'. However, the application specific HyTime elements (for example `Stext`, `Saudio`) are subtypes, of types representing architectural forms, and not a separate layer. We assume no updates to the instances in the databases.

A document model based on the Office Document Architecture (ODA) is described in [MRT91] and [BRG88]. 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 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 forms most of the paper [BRG88].

Conceptual structures do not form part of the SGML or ODA standards; our approach is characterized by a strict adherence to the SGML/HyTime standard.

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". This model performs almost all of the functions of a HyTime engine.

Others have focused on the temporal aspects of multimedia data, and their synchronization. [HR93] describes a model for composing multimedia objects and the playback of the composite objects. [LG93] describes a temporal model to capture the timing relationships between objects in composite multimedia objects, and maps it to a relational database.

9. CONCLUSIONS AND DISCUSSIONS

In this report we describe an object-oriented database design of a multimedia database for a news-on-demand application. There are three characterizing features of our work: (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 preparation. The other novel aspects of this work are the following:

1. 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.

2. The development of a complete type system that is in complete harmony with the news article DTD that was designed (Appendix 1).

3. The explicit representation of the spatio-temporal relationships in multimedia documents that enables the separation of the presentational considerations from the document structure and content.

This database is implemented on top of ObjectStore running under AIX. The implementation language is xlC, which is IBM's implementation of C++ for the AIX environment.

In the long-run, we are developing an extensible OBMS that has inherent support for multimedia formation systems. We intend to use our own system, called TIGUKAT[5] [ÖPS+95], to eventually replace ObjectStore. We may not be able to achieve the same performance, but there will be opportunities to expand on the functionality and investigate the feasibility of various issues. The issues that we have started to investigate or will investigate in the future include (1) the full support for the representation of spatial and temporal relationships, (2) the development of application specific query languages and primitives, and the optimization of these queries, (3) content-based indexing and querying of multimedia objects, and (4) the incorporation of the database with various storage systems that are fine-tuned for the storage of particular types of objects (e.g., video file servers, image file systems).

It is difficult, if not impossible, to investigate all of these issues with 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

---

[5] TIGUKAT (tee-goo-kat) is a term in the language of Canadian Inuit people meaning "objects." The Canadian Inuits, commonly knowns as Eskimos, are native to Canada with an ancestry originating in the Arctic regions of the country.

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.

A purely behavioral approach offers several benefits including consistency, understandability and portability. TIGUKAT's object model is uniform. Everything in the system, including types, classes, collections, behaviors, functions as well as meta-information, is a first-class object with well-defined behavior. Thus, there is no separation between objects and values. The schema information is a natural part of the database that can be queried like other objects. The uniformity of the model eliminates the separation between the schema and the objects and provides reflective capabilities to the system [PÖ93]. This allows the schema objects (i.e., types, behaviors, classes, collections) to be accessed using the query language just like other objects, simplifying dynamic schema evolution management.

We have included "time" as an inherent feature of the object model [GÖ93] by defining time as a type which can then be specialized in various forms (e.g., discrete time intervals, continuous time). The inclusion of time as a basic object management feature enables the direct representation of synchronization of the presentation of multimedia objects (i.e., temporal relationships between objects). There is a complete query model for TIGUKAT, including an SQL-like query language, a powerful object calculus and a set of object algebra operations [PLÖS93] which can be extended with application-specific constructs and primitives.

REFERENCES

[BAH93] K. Böhm, K. Aberer, C. Hürer. Extending the scope of document handling: The design of an OODBMS application framework for SGML document storage. Technical Report P-93-24, GMD-IPSI, Germany, 1993.

[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. et. al, "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 Engineering*, 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. Int. Workshop on Multimedia Information Systems*, pages 1–20, February 1992.

[EÖSV95] G. El Medani, M.T. Özsu, D. Szafron and C. Vittal. "A visual query facility for multimedia databases," submitted to *2nd International Conference on Multimedia Computing and Systems*, May 1995.

[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 Intl. Conf. on Management of Data,* pages 91–102, May 1994.

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

[GÖ93] I. Goralwalla and M.T. Özsu. "Temporal extensions to a uniform behavioral object model," In *Proc. 12th Int. Conf. on Entity-Relationship Approach,* pages 115–127, Dallas, Texas, December 1993.

[Hafi94]   A. Hafid et. al. On news-on-demand service implementation. Publication #928, Département d'Informatique et de Recherche Opérationnelle, Université de Montrèal, September 1994.

[Herw94] E. van Herwijnen. *Practical SGML,* Kluwer Publishers, 1994.

[HR93]   R. Hamakawa and J. Reikmoto. "Object composition and playback models for handling multimedia data", In *Proc. First ACM Intl. Conf. Multimedia*, 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.

[KRRK94] J. F. Koegel, L. W. Rutledge, J. L. Rutledge, C. Keskin. "HyOctane: A HyTime engine for an MMIS", In *Proc. First ACM Intl. Conf. Multimedia,* October 1993.

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

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

[MRT91] C. Meghini, F. Rabitti, and C. Thanos. "Conceptual modeling of multimedia documents," *IEEE 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 Int. Conf. on Very Large Data Bases*, pages 451-462, 1994.

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

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

47

[PLÖS93] R. Peters, A. Lipka, M.T. Özsu and D. Szafron. "An extensible query model and its languages for a uniform behavioral object management system", In *Proc. 2nd Int. Conf. on Information and Knowledge Management*, pages 403–412, November 1993, Washington, D.C.

[PÖ93] R.J. Peters and M.T. Özsu. "Reflection in a uniform behavioral object model,'' In *Proc. 12th Int. Conf. on Entity-Relationship Approach,* pages 37–49, Dallas, Texas, December 1993.

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

DOCUMENT TYPE DECLARATION 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)   - -
                   (emph1|emph2|quote|link|#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 2. DTD FOR STYLE SHEETS

```
<!DOCTYPE style-sheet SYSTEM "style-sheet.dtd" [
<!ELEMENT style-sheet    - -  (cat-name, rule+)>
<!-- Document category name, e.g. "article" -->
<!ELEMENT cat-name  - -  (#PCDATA)>
<!-- rule which maps element (source) to style (spec)-->
<!ELEMENT rule - -  (source, spec*)>
<!ELEMENT source    - -  (#PCDATA)>
<!-- each presentation attribute has a value -->
<!ELEMENT spec - -  (pres-attr, value)>
<!ELEMENT pres-attr - -  (#PCDATA)>
<!ELEMENT value- -  (#PCDATA)>


<!-- No Attributes for any of the style sheet elements -->
]>
```

APPENDIX 3. TYPE SYSTEM FOR DTD ARTICLE

Note1: All data members (attributes) are protected, which means that they are visible only to their type's subtypes. Corresponding get/set methods are implicitly assumed in most cases.
Note2: Pointer types are indicated by the suffix '_oid'. For example, a pointer to an object of type `Element` is shown as `Element_oid`.

Note3: The notation for methods is as follows:
 	
    methodA: `TypeA`
means that the return type of methodA is `TypeA`.
 	
    methodB: `TypeA × TypeB → TypeC`
means that `methodB` takes two arguments of types `TypeA` and `TypeB` and returns an object of type `TypeC`.

1. ATOMIC TYPES

Atomic

supertypes :   root
subtypes: NCMtype, CMtype

    attributes

        length   :   `Integer`
        QosParam   :   `QosParameterClass_oid`

`length` gives the size of object in characters
`QosParam` is a list of general parameters such as the jitter, delay, cost, throughput, and guarantee class

    methods

NCMType

supertypes: Atomic
subtypes: Image, Text

    attributes

        content   :   `String`

content is an array of characters (char [] in C++)

    methods

        compress   :
        uncompress   :

`compress` and `uncompress` take no arguments, and compress/uncompress the character stream

Text

<u>supertypes</u>:   NCMType
<u>subtypes</u>:   none

    <u>attributes</u>

        TextQoS   :   `QosTextClass_oid`

TextQoS stores the QoS parameters relevant to (non-synchronized) text

    <u>methods</u>

        match   :   `String` $\rightarrow$ `SearchResult`

        subString   :   `Integer` x `Integer` $\rightarrow$ `String`

`match` returns the result of matching its argument against the text. SearchResult contains the location as well as the surrounding text.
`substring` returns the substring located by the two integer arguments

Image

<u>supertypes</u>:   NCMType
<u>subtypes</u>:   (GIF_Image, JPEG_Image, etc will be subtyped when required) none

    <u>attributes</u>

        width   :   `Integer`
        height   :   `Integer`
        ImageQoS   :   `QosImageClass_oid`

width and height of image, common to all storage formats
ImageQoS stores the QoS parameters relevant to Images only

    <u>methods</u>

CMType

<u>supertypes</u>:   Atomic
<u>subtypes</u>:   SyncText, Temporal

    <u>attributes</u>

        filename   :   `FilePathClass_oid`
        location   :   `SiteClass_oid`
        UOI   :   `128 byte value`

`filename` and `location` give information on the location of the file on the particular server.
`UOI` is the universal object identifier which gives additional site-independent information. The SiteClass gives additional information on the transport protocol which is used by the QoS module

    <u>methods</u>

SyncText
<u>supertypes</u>:   CMtype
<u>subtypes</u>:   none

    <u>attributes</u>
        SyncTextQos :   `QoSSyncTextClass_oid`

    <u>methods</u>

Temporal
<u>supertypes</u>:   CMtype
<u>subtypes</u>:   Video, Audio
    <u>attributes</u>
        duration   :   `Integer`

duration of playback in seconds

    <u>methods</u>
        scale   :   `real` $\rightarrow$ `Temporal_oid`

scale the media on the time axis

Video
<u>supertypes</u>:   Temporal
<u>subtypes</u>:   none (MPEGVideo, MJPEGVideo, etc will be subtyped when required)
    <u>attributes</u>
        width   :   `Integer`
        height   :   `Integer`
        frameRate :   `Integer`
        bitRate   :   `Integer`
        videoQos   :   `QoSVideoClass_oid`

`width, height, frameRate, and bitRate` are attributes presumably common to all video storage formats
`videoQos` gives the QoS parameter list specific to video media, such as packetsize, and color.

    <u>methods</u>

Audio
<u>supertypes</u>:   Temporal
<u>subtypes</u>:

  <u>attributes</u>
        numChannels :   `Integer`
        codingLaw   :   `String`
        sampleRate :   `Integer`
        bitsperSample :   `Integer`
        AudioQos   :   `QoSAudioClass_oid`

`AudiQos` gives the QoS parameter list specific to audio media, such as packetsize

    <u>methods</u>

2. ELEMENT TYPES

Note1: To implement the heterogeneous collections that are required to store the child elements of structured elements, several abstract supertypes which are the 'union' types of the child elements have been created. These supertypes are subtypes of `Element`, `HyElement`, `TextElement`, `Structured`, or `StructuredText`. Consequently, most types listed here have more supertypes, which are these 'union' types. To avoid the explosion in the name space, these supertypes are not listed in the following text. Only one such type is shown: the `phrase` type. The `children` attribute of the `paragraph` type illustrates its use.

Note2: The notation `List<A>` means a list of objects of type `A`.

Element

| supertypes | : | root |
| subtypes | : | Text Element , Structured, HyElement ___ |

> attributes
>
> | parent | : | `Element _oid` |
> | ArticleElement | : | `Article_oid` |
>
> > `parent` is the oid of the element's parent in the hierarchy of the document instance
> >
> > `ArticleElement` is the article to which the element belongs to
>
> methods
>
> | GetParent | : | `Element _oid` |
> | SetParent | : | `Element _oid` |
>
> > These methods get refined by all subtypes

TextElement

All elements which can be represented by annotations

| supertypes | : | Element |
| subtypes | : | EdinfoElement, Figcaption, Emphasis, Quote, StructuredText |

> attributes
>
> | absoluteAnnote | : | `Annotation_oid` |
> | relativeAnnote | : | `Annotation_oid` |
>
> > `TextElements` are annotations by definition . The absolute annotation is with respect to the articles text content. The relative annotation is w.r.t the immediate parent element of the text element.
>
> methods
>
> | getString | : | `String;` |
>
> > returns the string enclosed in the annotation

Structured

SGML/HyTime elements with a complex content model.

<u>supertypes</u>:   Element
<u>subtypes</u>:   StructuredText, sync, article, audio-video, body, fcs_AF, Evsched_AF, dimspec_AF

     <u>attributes</u>

         children   :   `List<Element_oid>`

> Structured elements have children by definition. The least common supertype of all possible children of all `Structured` elements is `Element`. This attribute is *private*, so that it can be specialized by the subtypes of `Structured`.

     <u>methods</u>

         getNth   :   `Integer` → `Element_oid`

> returns the oid of the Nth child of the element where N is the argument. Return type is specialized in the subtypes.

StructuredText

This type and all its subtypes model text elements (`TextElement`) with a hierarchical structure.

<u>supertypes</u>:   Text Element, Structured __
<u>subtypes</u>:   section, figure, async, frontmatter, article, edinfo, list, listitem, hdline, paragraph

     <u>attributes</u>

         children   :   `List < TextElement_oid >`

> The actual implementation of the collection of child elements could be different, for example a number of lists. A `private` data member, again.

     <u>methods</u>

         getNth   :   `Integer` → `TextElement_oid`

> returns the oid of the Nth child of the element where N is the argument. Refined from `Structured`

HyElement

HyTime architectural forms have this as a supertype.

<u>supertypes</u>:   Element
<u>subtypes</u>:   Ilink_AF, Fcs_AF, Evsched_AF, Event_AF, Axis_AF, Extlist_AF, Marklist_AF, Dimspec_AF

     <u>attributes</u>

         id   :   `String`
         AForm   :   `String`

> All HyTime elements are required to have IDs in the News article DTD. HyTime elements should also be able to answer the name of the architectural form they conform to.

     <u>methods</u>

         getAForm   :   `String`

Fcs_AF

<u>supertypes</u>:   HyElement, Structured
<u>subtypes</u>:   Av-fcs

    <u>attributes</u>

        evsched   :   `List <Evsched_AF_oid>`

> The content model of `fcs` is `(evsched)+`. Hence the child elements are only of type `Evsched_AF`

    <u>methods</u>

        getNth   `:Integer` $\rightarrow$ `Evsched_AF_oid`

> Refined from `Structured` type

Av-fcs

<u>supertypes</u>:   Fcs_AF
<u>subtypes</u>:   (none)

    <u>attributes</u>

        t_axis   :   `Time_oid`
        x_axis   :   `X_oid`
        y_axis   :   `Y_oid`
        t_fcsmdu   :   `String`
        x_fcsmdu   :   `String`
        y_fcsmdu   :   `String`

> These attributes store the knowledge about the axes and the measurement data units (fcsmdu) along various axes.

    <u>methods</u>

        GetTimeSchedule   :   `TimeFlowGraph`
        GetSpatialSchedule :   `SpatialScheduleInfo`
        GetVideoObjects   :   `List  <Video_oid>`
        GetAudioObjects   :   `List  <Audio_oid>`
        GetSyncTextObjects :   `List  <Text_oid>`
        GetImageObjects   :   `List  <Image_oid>`

> `GetTimeSchedule` returns the time flow graph representing the playback schedule.

> `GetSpatialSchedule` returns a representation of the spatial ordering of the events in the fcs.

> GetXXXObjects return a list of object references to atomic types representing these media. These references can then be used to retrieve QoS and location information.

Ilink_AF

<u>supertypes</u>:   HyElement
<u>subtypes</u>:   link

    <u>attributes</u>

        linkends   :   `List <Element_oid>`
        anchrole   :   `String`

> `Ilink` can link to multiple destination, and can link to any element. Hence `linkends` is a list of `Element` references

> `anchrole` is fixed in the DTD, according to the HyTime standard.

    methods

        traverse   :   `Element_oid` → *null*

> `traverse` defines some semantics for traversing a link.
> This is refined in each subtype.

Link

supertypes:	Ilink_AF, Structured
subtypes:	none

    attributes

        children   :   `List of Phrase_oid`

> the content model of the link element is a phrase.

    methods

        getNth   :   Integer → Phrase_oid

        traverse   :   TextElement_oid → *null*

        traverse   :   StructuredElement_oid → *null*

        traverse   :   TextElement → *null*

> `getNth` is a refined method of `Structured`.

> `traverse` is polymorphic. Depending upon the type of
> the destination element supplied as an argument, there are
> different implementations.

Evsched_AF

supertypes:	HyElement, Structured
subtypes:	Av_evsched

    attributes

        axisord   :   `String`
        apporder   :   `String`
        sorted   :   `String`
        basegran   :   `String`
        gran2hmu   :   `List<Integer>`
        pls2gran   :   `List<Integer>`
        children   :   `List<Event_AFoid>`

    methods

> All the attributes, except for `children` are attributes of the
> architectural form listed in the HyTime standard.

Av_evsched

supertypes:	Evsched_AF
subtypes:	none

    attributes

        textEvents   :   `List<Stext>`
        audioEvents :   `List<Saudio>`
        videoEvents :   `List<Svideo>`

> the text, audio and video events are stored in individual lists
> so that the implementation of `GetTimeSchedule` in the
> `Av-fcs` type becomes simpler.

    methods

Event_AF

supertypes: HyElement
subtypes: temporal

    attributes

        exspec   :   `Extlist_AF_oid`
        pls2gran   :   `List<Integer>`

> All the attributes are attributes of the architectural form listed in the HyTime standard.

    methods

Temporal

supertypes: Event_AF
subtypes: Spatial, Saudio

    attributes
        taxis   : `Time_oid`
    methods
        getTimeExtents   :   `List<Extents>`

> returns the portions of the time axis the event occupies

Spatial

supertypes: Temporal
subtypes: Stext, Svideo

    attributes
        x_axis   :   `X_oid`
        y_axis   :   `Y_oid`
    methods
        getXExtents   :   `List<extents>`
        getYExtents   :   `List<extents>`

> These methods return the extents of the event object along the spatial axes.

Stext

supertypes: spatial
subtypes: none
    attributes
        content   :   `SyncText_oid`
    methods

Saudio

supertypes: temporal
subtypes: none
    attributes
        content   :   `Audio_oid`
    methods

Svideo
supertypes: spatial
subtypes: none
> attributes
>> content   :   `Video_oid`
> methods

Axis_AF
supertypes: HyElement
subtypes: X, Y, Time

> attributes
>> axismeas   :   `String`
>> axismdu   :   `String`
>> axisdim   :   `String`

All are HyTime Standard attributes

> methods

X, Y, Time
subtypes: none
supertypes: Axis_AF
> attributes

> methods

subtyped for classification purposes

Extlist_AF
subtypes: none
supertypes: HyElement, Structured

> attributes
>> children   :   `List<Dimspec_oid>`

The Standard specifies a more complex content model which includes a number of architectural forms which are not going to be used in this implementation.

> methods

Av-extlist
supertypes: Extlist_AF
subtypes: none
> attributes
>> time_spec   :   Tdimspec_oid
>> x_spec   :   Xxdimspec_oid
>> y_spec   :   Ydimspec_oid

children of `Av-extlist` are `Dimspec` elements, which are stored as attributes, rather than a list. The dimspec elements are used to mark off portions of axes.

> methods

Dimspec_AF

supertypes: StructuredText
subtypes: Tdimspec, Xdimspec, Ydimspec

      attributes

mark1	:	Integer
mark2	:	Integer

> dimspec is a pair of integers marking off positions along an axis.

      methods

Xdimspec, Ydimspec, Tdimspec

supertypes: Dimspec_AF
subtypes: none

> subtyped for classification purpose only. They also represent elements in the DTD

Figure

supertypes: StructuredText
subtypes: none

      attributes

image	:	Image_oid
caption	:	Figcaption_oid
float	:	String

> image refers to the raw image associated with the figure

> float has a value which indicates where the figure is to be displayed relative to the surrounding text (here, top, bottom)

      methods

getCaption	:	String

> returns the caption associated with the figure (null if there is no caption).

Phrase

supertypes: none
subtypes: Emphasis, Quote, String

      attributes

      methods

> abstract supertype for emphasis and quote elements. An example of an 'union' type used to implement heterogeneous collections for the children attribute of Structured elements.

Paragraph

supertypes: StructuredText
subtypes: none

    attributes
        children   :   `List<(Phrase,List,Link,Figure)_oid>`

    methods
        getNth   :   `Integer → (Phrase,List,Link,Figure)_oid`

List

supertypes: StructuredText
subtypes: none

    attributes
        title   :   `String`
        number   :   `Integer`
        children   :   `List <Listitem_oid>`

> title is the list header or title (which can be null).
> number is the number of list items in the list.

    methods
        getNthListitem: `Integer → Listitem_oid`

Listitem

supertypes: StructuredText
subtypes: none

    attributes
        children   :   `List <Paragraph_oid>`

    methods
        getNth   :   `Integer → Paragraph_oid`

> `getNth` refined from `StructuredText` to reflect the fact that the only children of `listitem` are `paragraph` instances.

Section

supertypes: StructuredText
subtypes: none

    attributes
        title   :   `String`
        children   :   `List <(Paragraph, List)_oid>`
    methods
        getNth   :   `Integer → (Paragraph, List)_oid`

> the return type of `getNth` for this type is the supertype of `paragraph` and `list` types.

Async

<u>supertypes:</u>   StructuredText
<u>subtypes:</u>   none
    <u>attributes</u>
        children   :   `List <Section, Figure, Link oids>`
    <u>methods</u>
        getNth   :   `Integer` → `(Section,Figure,Link)_oid`

> the return type of `getNth` for this type is the supertype of `Section, Figure` and `Link` types.

Edinfo

<u>supertypes:</u>   StructuredText
<u>subtypes:</u>   none
    <u>attributes</u>
        location   :   `Loc`
        date   :   `Date`
        source   :   `Source`
        authors   :   `List <Author >`
        subject   :   `Subject`
        keywords   :   `Keywords`
        abstract   :   `Abs-p`
    <u>methods</u>

> the return type of `getNth` for this type is `EdinfoEle-ment_oid`

Frontmatter

<u>supertypes:</u>   StructuredText
<u>subtypes:</u>   none
    <u>attributes</u>
        children   :   `List<(Edinfo,Hdline,Subhdline,`
                                       `Abs-p)_oid>`
    <u>methods</u>
        getNth   :   `Integer` →
                         `(Edinfo,Hdline,Subhdline,Abs-p)_oid`

> the return type of `getNth` for this type is the supertype of `EdinfoElement, Hdline,` and `Abs-p` types.

Hdline

<u>supertypes:</u>   StructuredText
<u>subtypes:</u>   Subhdline
    <u>attributes</u>
        children   :   `List<(Phrase,Link)_oid>`
    <u>methods</u>
        getNth   :   `Integer` → `(Phrase,Link)_oid`

Subhdline

<u>supertypes:</u>   Hdline
<u>subtypes:</u>

    <u>attributes</u>

        children   :   `List<(Phrase,Link)_oid>`

    <u>methods</u>

        getNth   :   `Integer → (Phrase,Link)_oid`

Abs-p

<u>supertypes:</u>   StructuredText
<u>subtypes:</u>   Subhdline

    <u>attributes</u>

        paragraph   :   `Paragraph_oid`

    <u>methods</u>

EdinfoElement

<u>supertypes:</u>   TextElement
<u>subtypes:</u>   Loc, Source, Author,Subject,Date

    <u>attributes</u>

        parent   :   `Edinfo_oid`

    <u>methods</u>

        GetParent   :   `Edinfo_oid`

Keywords

<u>supertypes:</u>EdinfoElement
<u>subtypes:</u>   none

    <u>attributes</u>

        number   :   `Integer`

number of keywords

    <u>methods</u>

Loc, Source, Subject

<u>supertypes:</u>   EdinfoElement
<u>subtypes:</u>   none

    <u>attributes</u>

    <u>methods</u>

Subtyped only for classification purposes

Date

<u>supertypes:</u>   EdinfoElement
<u>subtypes:</u>   none

    <u>attributes</u>

        day   :   `Integer`
        month   :   `Integer`
        year   :   `Integer`

    <u>methods</u>

Author

supertypes: EdinfoElement
subtypes: none

attributes

bio	:	Paragraph
photo	:	Image
designation	:	String
affiliation	:	String

methods

> all attributes are derived from attribute values found in the markup. They could be null, or empty.

Emph1, Emph2

supertypes: Emphasis
subtypes: none

attributes
(none)
methods
(none)

> subtyped only for classification purposes

Figcaption

supertypes: TextElement
subtypes: none

attributes
(none)
methods
(none)

> subtyped only for classification purposes

Quote

supertypes: TextElement
subtypes: none
attributes

source	:	String
style	:	String
date	:	Date

> source is who the quote is attributed to, and style is the either block, or embedded. date is the date (if any) associated with the quote. All these values are obtained from the attributes given in the markup for the quote element.

methods

Sync

<u>supertypes</u>:   Structured
<u>subtypes</u>:   none

    <u>attributes</u>

children	:	`List<AudioVisual_oid>`

    <u>methods</u>

getNth	:	`AudioVisual_oid`

> return type of `getNth` is `AudioVisual_oid`

AudioVisual

<u>supertypes</u>:   Structured
<u>subtypes</u>:   none

    <u>attributes</u>

x_axis	:	`X_oid`
y_axis	:	`Y_oid`
t_axis	:	`Time_oid`
fcs	:	`Av-fcs_oid`
extlists	:	`List<Av_extlist_oid>`

    <u>methods</u>

getNth	:	`(X, Y, Time, Av-fcs,Av-extlist)_oid`

Article

<u>supertypes</u>:   Structured
<u>subtypes</u>:   none

    <u>attributes</u>

location	:	`String`
date	:	`String`
source	:	`String`
subject	:	`String`
authors	:	`List<String>`
keywords	:	`List<String>`

> attributes replicated from the `edinfo` type, so that indexes can be built on them for faster searching.

front	:	`Frontmatter_oid`
async	:	`Async_oid`
sync	:	`Sync_oid`

> child elements stored as attributes instead of a list.

base	:	`Article_root_oid`

> reference to the object which has the text content, the lists of annotations, and the collection of all elements.

    <u>methods</u>

Article_root

<u>supertypes:</u>   Root
<u>subtypes:</u>   none (yet)

    <u>attributes</u>

textblock	:	`Text_oid`
imageList	:	`List<Image_oid>`
audioList	:	`List<Audio_oid>`
videoList	:	`List<Video_oid>`
stextList	:	`List<SyncText_oid>`

> The list of atomic objects in the article. Annotations are defined on the text contained in the textblock attribute.

article	:	`Article_oid`

> The instance of the `article` element which is at the root of the document hierarchy.

loc	:	`Annotation_oid`
keywords	:	`Annotation_oid`
source	:	`Annotation_oid`
author	:	`Annotation_oid`
subject	:	`Annotation_oid`
date	:	`Annotation_oid`
figcaptions	:	`List<Annotation_oid>`
Emph1	:	`List<Annotation_oid>`
Emph2	:	`List<Annotation_oid>`
quote	:	`List<Annotation_oid>`
list	:	`List<Annotation_oid>`
paragraph	:	`List<Annotation_oid>`
figure	:	`List<Annotation_oid>`
section	:	`List<Annotation_oid>`
frontmatter	:	`List<Annotation_oid>`
async	:	`List<Annotation_oid>`
listitem	:	`List<Annotation_oid>`
link	:	`List<Annotation_oid>`
edinfo	:	`List<Annotation_oid>`
hdline	:	`List<Annotation_oid>`
subhdline	:	`List<Annotation_oid>`
abs-p	:	`List<Annotation_oid>`
title	:	`List<Annotation_oid>`

> Annotation lists for text elements

    <u>methods</u>

Annotation

<u>supertype</u> :   Root
<u>subtype</u>:   none

    <u>attributes</u>

first	:	`Integer`
last	:	`Integer`

    <u>methods</u>