AN OBJECT-ORIENTED SGML/HYTIME COMPLIANT MULTIMEDIA DATABASE MANAGEMENT SYSTEM*

M. Tamer Özsu, Paul Iglinski, Duane Szafron, Sherine El-Medani†, and Manuela Schöne††
Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

ABSTRACT

We describe the design of an object-oriented multimedia database management system that can store and manage SGML/HyTime compliant multimedia documents. The system is capable of storing, within one database, different types of documents by accommodating multiple document type definitions (DTDs). This is accomplished by dynamically creating object types according to element definitions in each DTD. The system also has tools to automatically insert marked-up documents into the database. We discuss the system architecture, design issues and the system features.

1. INTRODUCTION

Traditionally, multimedia applications have not fully exploited database management system (DBMS) technology. Mostly they have used DBMSs to store meta-information and have stored multimedia objects in flat files. The connection between the DBMSs and the object files is usually non-existent (or very loose), requiring the application programs and users to access both of the repositories themselves. This is not a preferable course of action for three major reasons. First, 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 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), and fault tolerance (transactions, recovery). Second, multimedia objects have temporal and spatial relationships that must be taken into account for synchronization and display of information (e.g., synchronization of an image with its audio annotation). 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. Finally, the size and complexity of multimedia objects require special treatment in storage, retrieval, and transmission. DBMS technology, especially object DBMSs, is particularly well suited for the representation and storage of this type of data.

Recently, a number of multimedia projects that have used DBMSs have been reported in the literature (see [PS97] for a survey). However, there are a number of common deficiencies in many of these attempts. Some of them use the DBMS technology only to store and manage meta-data, while the actual multimedia objects are stored in ordinary files with very little integration between the DBMS and these files. This restricts the role of DBMSs and makes it difficult (if not impossible) to apply regular database access methods to multimedia data. Others store some of the multimedia objects (e.g., text and still images) as binary large objects ("BLOB"s) in relational DBMSs. Even though this puts some of the multimedia objects in a database, it is usually not possible to provide database functionality over these objects. For example, content-based querying of BLOBs is not possible since the DBMS treats them as byte strings whose interpretation is left to the application. Still others develop and implement their own one-of-a-kind models for multimedia data. Unfortunately, this requires everything to be developed from scratch since various tools based on international standards cannot be used together with these one-of-a-kind systems.

In this paper, we report our work in developing an object DBMS to store and manage SGML/HyTime [Gold90, DD94] compliant multimedia documents. Three points are important about our work. First, even though we heavily use DBMS technology to manage multimedia data, we are not arguing for the storage of **all** multimedia data in a database. In fact, most of the multimedia data currently reside in traditional files which will need to be incorporated into multimedia information systems in an interoperable environment. The argument put forth here is that the users' access

* This research is supported by Canadian Institute for Telecommunications Research (CITR), one of the Networks of Centres of Excellence funded by the Government of Canada.

† Current address: Vicom Multimedia Inc., Edmonton, Alberta, Canada.

†† Current address: Intelligent Marketing Systems, Edmonton, Alberta, Canada.

to this data should be managed by a DBMS. We propose to accomplish this by storing and managing the **structure** of **all** multimedia objects, as well as some of the objects themselves in an object database, and providing strong links from this DBMS to the particular servers (e.g., image file systems or video-on-demand systems) that store the remainder of the data. Thus, interoperability, the second important point, is a central aspect of our work. In this paper our emphasis is on the database models and associated tools to store and manage multimedia data; interoperability issues will be reported elsewhere. Finally, we base our work on the SGML/HyTime standard for representing multimedia documents. This allows us to build an entire system by using commonly available tools that work with these standards.

The system described in this paper is part of the larger Broadband Services Project that is being conducted by five institutions in Canada with support from the Canadian Institute for Telecommunications Research. The objective of the larger project is to develop a software infrastructure and to define an API that is suitable for a wide range of broadband distributed multimedia applications [WLE+97]. The multimedia DBMS component of this project covers a broad spectrum of activities from the design and implementation of an appropriate database type system (schema) to the development of application-specific query models and languages that support content-based access to multimedia objects. This paper focuses on a couple of components of the multimedia DBMS; issues such as video and image modeling are reported elsewhere [LÖS96a-c] and other components including multimedia query languages, image and video indexing will be reported in the future.

We assume that the reader is familiar with SGML/HyTime and we do not repeat the basic characteristics of these standards. Readers can refer to the standards themselves [ISO86, ISO92] or to the various books that describe them (e.g., [Gold90, DD94]). The organization of the rest of the paper is as follows. In the next section, we present the architecture of the system that we have developed. Sections 3-5 focus on the core aspects of our system: the type system that we have implemented, the management of multiple document structures, and the automatic insertion of documents into the database. Section 6 contains a review of some of the related work and compares our system with others. Section 7 discusses the current status of the system and reviews some of the improvements that are underway.

2. SYSTEM OVERVIEW

One of the strengths of DBMSs is their ability to manage data on behalf of multiple applications and enable data sharing among them. DBMSs provide this service in a transparent manner, shielding the applications from the particulars of physical data organization, distribution and performance considerations (query optimization). If DBMSs are to support multimedia applications, the same services need to be extended to multimedia data. Within the context of an SGML/HyTime compliant system, this means the following:

1. The system has to be able to model all types of multimedia data (video, audio, text, images), not only simple data such as character strings and numeric values.
2. The system needs to be open and extensible, both in terms of the types of data it can manage and in terms of its architecture so that it can accommodate different servers and applications.
3. The system has to be able to store and manage different types of documents in one database. In SGML parlance, it should be able to deal with multiple DTDs and be able to store documents whose markup conforms to these DTDs.
4. The system should facilitate the automatic insertion of multimedia documents into the database and provide facilities for querying of these documents both with respect to their contents and with respect to their structure.

The system described here satisfies these constraints. It is a distributed (client-server), object-oriented system that allows coupling with other servers (in particular continuous media servers) and handles dynamic creation of object types based on DTD elements.

The system is built as an extension layer on top of a generic[1] object DBMS, ObjectStore [LLOW91]. The extensions provided by the multimedia DBMS include specific support for multimedia information systems. The development of a type system that supports common multimedia types is at the heart of the multimedia extensions. This paper focuses on the extensible kernel type system as well as the supporting tools to facilitate automatic type creation from a given DTD and the automatic insertion of conforming documents into the database.

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

[1] ObjectStore is a generic ODBMS in the sense that it doesn't have native multimedia support in its base product. The latest version (Version 5) of ObjectStore incorporates *object managers* as add-on products providing limited support for multimedia data types. These class libraries provide some partial solutions to multimedia issues, which may benefit our implementation in future releases.

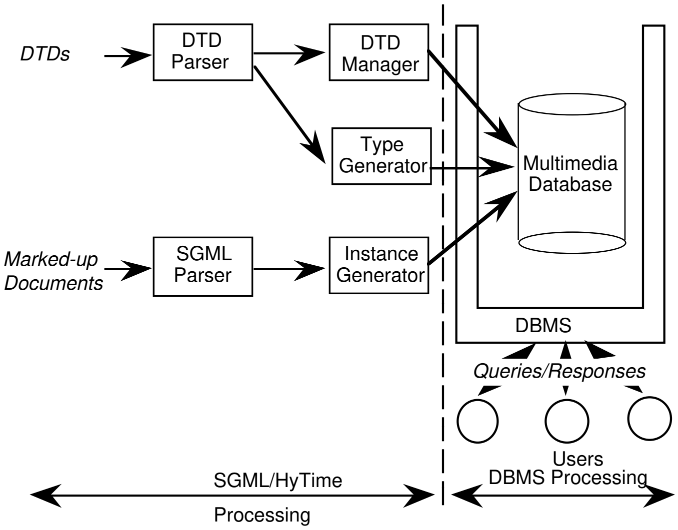

Figure 1. Complete Processing Environment

placed on top of the file system servers and the underlying storage system can be modified accordingly. Alternatively, links can be provided from the multimedia DBMS to the various servers which may be located at different sites of a distributed system. In our prototype configuration, for example, the audio and video objects are stored in a separate continuous media server [MNH96] while meta-data about these objects are stored in our multimedia DBMS.

Another aspect of our system is its integration with tools associated with the management of SGML/HyTime databases. These include DTD parsers and managers, and SGML parsers and instance generators. These are discussed in detail in Sections 5 and 6. Overall, the system deals with both database processing and SGML/HyTime processing issues (Figure 1).

3. MULTIMEDIA TYPE SYSTEM

In designing a type system for an SGML/HyTime compliant DBMS, four issues need to be addressed [ÖSEV95]:

- The different media components of the document (i.e., text, image, audio, and video) need to be modeled and stored in the database. These are called *monomedia objects*. The design of their storage structures in the database is critical for good performance.
- A representation is needed for the document's logical structure. Not every multimedia information system represents the document structure explicitly. For example, a multimedia system using postscript files for documents ignores the hierarchical structure of the document. Explicit representation of the structure facilitates querying and presentation.
- In multimedia documents, one has to deal with the representation of the spatial and temporal relationships between monomedia objects. These relationships are important for presentation purposes – spatial relationships are used to model the placement of the various components on the screen while temporal relationships are essential for the synchronization of monomedia objects during presentation (e.g., audio synchronization with video or captioned text with video).
- Meta information about the DTD elements and instance creation functionality are necessary for document insertion and queries about document structure. A meta-type system can satisfy these requirements.

We discuss below how our system handles each of these issues within the framework of an object DBMS. In this context, "modeling" refers to the design of a type system that supports the representation of various system components.

3.1 Atomic Type System

Our atomic type system consists of the types that are defined to model monomedia objects. This part of the type system establishes the basic types that are used in multimedia applications.

One of the important considerations in the design of the type system is quality-of-service (QoS) requirements. Monomedia objects are associated with specific QoS parameters that are needed for presentation purposes. For example, the QoS parameters of an image can be the format (e.g., JPEG, GIF, TIF), size, resolution, and color depth. For one *logical* monomedia object, there can be a number of concrete objects that can be distinguished only in their QoS parameters. These concrete objects are called *variants* of a logical monomedia object. During QoS negotiation, depending on the available hardware and the desired quality and cost, different variants of the same monomedia object can be retrieved and displayed. Continuous media objects such as audio and video may consist of a number of *data streams*. Some video and image compression technologies, for example, utilize a base stream that can be merged with enhancement streams to obtain higher quality of service levels. Similarly, an audio variant might provide quadraphonic sound with four data streams, one for each channel. It is important to be able to model this relationship between monomedia objects, their variants and the data streams that make up each variant, because multimedia applications need to

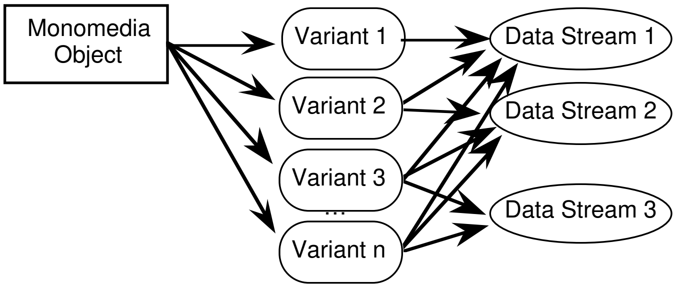

Figure 2. Variants of Monomedia Objects

access each of these objects coherently.

Figure 2 shows a monomedia object with different QoS levels represented by different variants. In Figure 2, Variant 1 consists of Data Stream 1; Variant 2 shares Data Stream 1 with Variant 1, but also contains Data Stream 2. In general, the presentation quality is higher if more data streams are used. Therefore, the QoS level of Variant 2 is higher than of Variant 1. In this example, Variant 3 and Variant n consist of the same streams but differ in other QoS parameters such as maximum jitter, frame loss or delay.

The type system for atomic types is depicted in Figure 3[2]. All atomic types are subtypes of the abstract supertype `Atomic`, which is the root type in the `Atomic` type system. `Atomic` has one data member, the logical identifier `id`, which stores an SGML ID. Each element utilizing an atomic type requires an SGML ID attribute. SGML requires that this ID be unique throughout the document. The details of the type definitions are omitted here due to space limitations—they are provided in [Sch96]; in the remainder of this sub-section, we provide an overview of the semantics of the atomic types.

`DataStream` is an atomic type for streams. A stream identifies a particular file on the continuous media file server. It has data members for meta data related to the data streams, namely size and a universal object identifier.

Instances of `Variant` subtypes hold the monomedia representation and the variant's QoS information. Type `Variant` contains functions to access QoS parameters that are common to all variants. There are two abstract subtypes of `Variant`, `NCMType` for non-continuous media such as text and images, and `CMType` for continuous media such as audio and video. This distinction is made since non-continuous and continuous media are handled differently in the system. The difference between the two types is that

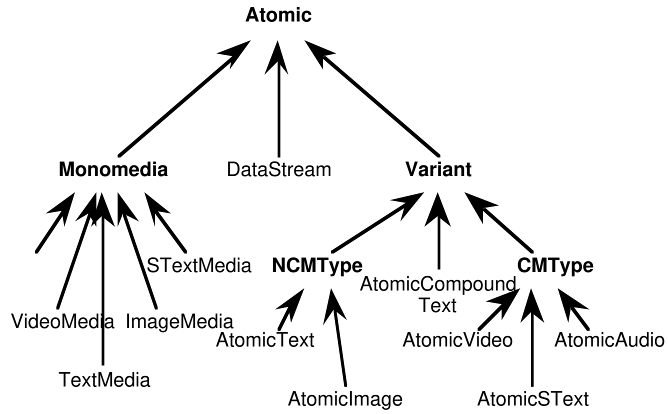

Figure 3. Atomic Type System

instances of types derived from `NCMType` store the raw media in their objects, whereas instances of types derived from `CMType` have only meta-information stored about the continuous media streams, which themselves reside on the continuous media server. `NCMType` and `CMType` have subtypes for the types of monomedia objects that they model (`AtomicSText` corresponds to synchronized text). A composite type, `AtomicCompoundText`, contains a collection of `AtomicText` to represent the textual component of documents whose DTD specifies segmentation of text storage, as described in the following subsection.

The concept of `Monomedia` types has been introduced to facilitate the definition of variants. An instance of a subtype of `Monomedia` type (say `aVideoObject` which is an instance of `VideoMedia`) may have a number of variants, as discussed previously. Instances of monomedia types store references to all of their variants. Thus, `aVideoObject` has references to its variants which are instances of `AtomicVideo`. A monomedia object can have only variants of the same type associated with it; for example, `VideoMedia` can only contain `AtomicVideo` objects.

In this paper, we do not discuss the details of how we model and represent the contents of monomedia objects. Our approach to representing the textual part of multimedia documents is described in [ÖSEV95]; the video modeling approach is discussed in [LÖS96b] and the image model is presented in [LÖS96a]. We have not yet done work on audio representation.

3.2 Element Type System

The element type system is a uniform representation of elements in a DTD and their hierarchical relationships. Each logical element in a DTD is represented by a concrete type in the element type

[2] Abstract supertypes are displayed in bold font, whereas concrete types are displayed in a normal font.

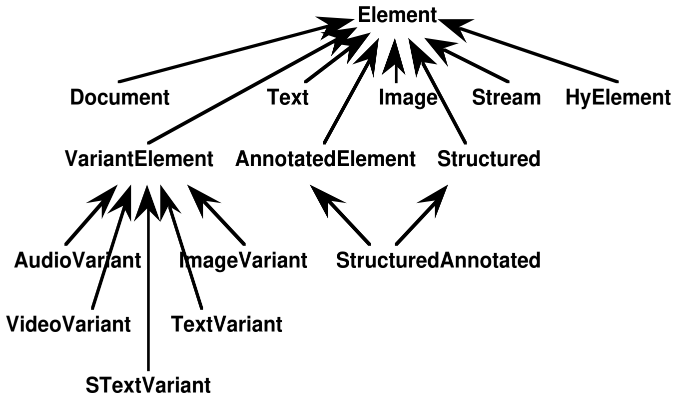

Figure 4. Element Type System

system.

An important design decision relates to the "shape" of the element type system. SGML, as a grammar, is fairly flat but allows free composition of elements. This, coupled with the requirement to handle multiple DTDs within the same database, suggests that the type system also be flat, consisting of collections of types (one collection for each DTD) unrelated by inheritance. This simplifies the dynamic type creation when a new DTD is inserted (to be discussed in Section 5). However, this approach does not take full advantage of object-oriented modeling facilities, most importantly behavioral reuse. Instead of a flat type system, we implement a structured type system where some of the higher-level types are reused through inheritance. This has the advantage of directly mapping the logical document structure to the type system in an effective way — a HyTime element in the DTD becomes a HyTime type in the type system. Furthermore, some of the common data definitions and behaviors for similar types can be reused. The disadvantage is that type creation (as discussed in Section 5) is more difficult. Information such as the characteristics of elements have to be obtained from the DTD to generate the new types as part of the type hierarchy.

As a result of this design decision, the system provides a set of built-in types that constitute the core of the element type system. These types model characteristics that are common to (some or) all DTD elements. This core set of (abstract) element types are depicted in Figure 4. All other element types that may be defined are derived from one of these abstract built-in types.

Type `Element` is the supertype of all element types. It contains the data members `document` and `parent`, together with member functions to access them. Data member `document`, of type `Document`, points to the multimedia document that contains the element. The introduction of this attribute enables each element to know its document. Since SGML documents have a tree structure, each element, except the root element, has a parent. The data member `parent` models this structure by pointing to the element's parent element. The type of `parent` is `Structured` since all parent elements must be structured elements with child nodes (this is further discussed below).

Type `Structured` is a supertype for all elements in the DTD that are potentially non-leaf nodes in the document tree. Such elements have a complex content model, meaning that the content model is not `EMPTY` or `#PCDATA`. Elements with a complex content model may have child elements. That is why structured elements must maintain references to their child elements. Type `Structured` has data members that keep track of these references and member functions that access them.

For efficiency reasons, all textual components of a document are, by default, stored together as one text string [ÖSEV95]. Each textual component (e.g., paragraph, emphasis, etc.) has an *annotation* associated with it that indicates the start and end index of the object's text in the text string. Annotations are not only useful for elements that contain textual data (`#PCDATA`) but also for other elements that have to be located within the text string to display them properly (e.g., figure, link). `AnnotatedElement` type is the abstract type that has the data members that maintain these annotations.

While this default text storage model avoids fragmentation of the document text into numerous objects and facilitates text retrieval and text searching, it leads to expensive document updating and may not be suitable for lengthy documents. Under this model, the modification of text in a document requires modification of all annotations that follow or span (i.e., as a parent element) the changed text. Thus, a small change near the front of a long document requires annotation updates for nearly all the elements as well as (potentially) a string copy of the entire document text.

To address this problem, the text storage model has been augmented by means of a type, `AtomicCompoundText`, and a text segmentation facility that responds to the use of a "text-seg" attribute for elements in the DTD. This allows the DTD writer to specify which elements are to be maintained as separate text segments. The attribute can be given a default token value that can be over-ridden by the document author. Thus, while the DTD writer bears the initial responsibility for determining the desired granularity of text segmentation, the document author can over-ride the default specified in the DTD by specifying an alternative value for the "text-seg" attribute in the particular element's tag. With user-specified text segmentation, annotations are relative, consisting of a start-key and end-key. These keys are associated in a dictionary with a reference to an `AtomicText` (a text segment) object and an offset into the string in that object. The hash table implementation of the dictionary insures rapid lookup. User-specified text segments nested within other user-specified text segments are handled appropriately as a simple sequence of segments. Text that intervenes between user-specified text segments is treated as a separate segment.

Current work involves developing a simple interface for updating documents based on insertion and deletion using HyTime's tree location address (treeloc) for specification of elements in the document tree. Both insertion and deletion will involve validation by the SGML parser.

In order to provide a consistent way of handling DTD elements that refer to atomic objects, we have introduced another special attribute named MM (**M**ono**M**edia) for such elements. The value of the MM attribute is fixed in the DTD to a value that identifies the appropriate atomic type. The mechanism of the MM attribute closely parallels that of the HyTime attribute, described in the following sub-section, which identifies a pre-defined architectural form to which the element must conform. Essentially, this constitutes a set of MM architectural forms for our system which supplement the standard HyTime architectural forms. In fact, some elements can be defined so as to conform to both a HyTime architectural form and an MM architectural form.

For each concrete atomic type there is a pre-defined element type which contains a pointer to an object of the corresponding atomic type. If, for example, the DTD defines an element MyImage with an MM attribute fixed to the value *imageVariant*, a new type will be generated that is subtyped from the pre-defined class `ImageVariant`, which contains a data member that points to an `AtomicImage`. In this case, the element MyImage must have attributes, namely QoS data, and a content model that conform to the MM architectural form for *imageVariant*. The image and its required QoS data are all stored directly in the atomic object.

This way, we can exploit our atomic type system and maintain a one-to-one correspondence between concrete element types and elements defined in the DTD. The advantages are two-fold. First, there is a clear distinction and interface between atomic types and element types. Atomic types model the primitive monomedia objects whereas element types model elements in a DTD. Second, this clear separation simplifies the interaction of other system components with the database. For example, the QoS negotiator module, developed by a partner research group, deals exclusively with the atomic types, completely ignoring the element type system.

For type checking, it is very useful for an object to know its type. To achieve this, the data member `type` has been introduced as a static member of each concrete element class. It is initialized to point to a meta-type object that specifies the type of the element. Each element type has a meta-type associated with it (e.g., type `Article` has the meta-type `ArticleType`). The meta-type object contains the name of the element as it appears in the DTD and other useful information that can be queried. Meta-types are discussed fully in Section 3.4 below.

3.3 Presentation Type System

Presentation of multimedia documents may involve complicated scenarios which require synchronization of various media, the placement of various objects on the screen, and QoS considerations. The algorithms to meet these requirements are outside the scope of the multimedia DBMS. However, it should be possible to store in the database presentation related data for a document that is then used by the presentation tools. In general, the presentation data involves the spatio-temporal relationships between the various objects.

The design challenge is how to model these relationships within the framework of the SGML/HyTime standard. In our system, we make use of several architectural forms that are defined within HyTime. As a background, the HyTime standard is divided into modules, each of which describes a group of concepts that are represented by architectural forms (AFs). The modules include the base module, the measurement module, the location address module, the hyperlinks module, the scheduling module, and the rendition module. Each module may use certain features of other modules lower down in the hierarchy; thus the location address module defines AF's which are used in the rendition module. Each HyTime DTD must declare the names of the modules it requires.

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.

HyTime models space and time using axes of finite dimensions. A finite coordinate space is a set of such axes. All measurements are associated with *axes.* The units of measurement along axes are called *quanta*. An *extent* is a set of ranges along the various axes defining the FCS. An *event* corresponds to an extent in the FCS. An *event schedule* consists of one or more events. Extents are specified using the `extlist` AF,

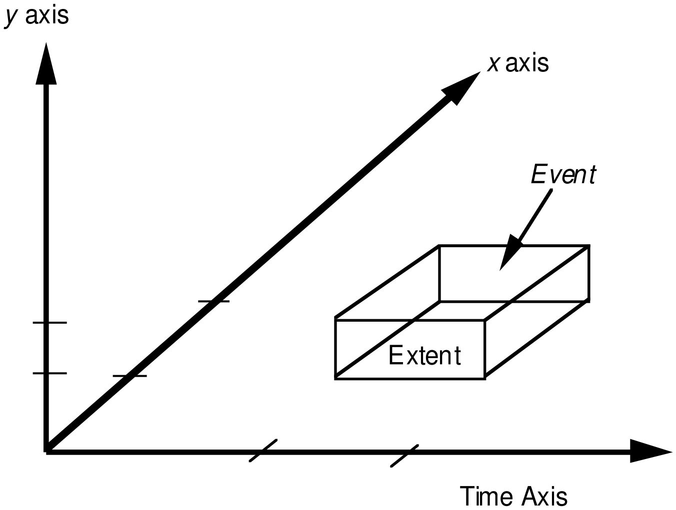

Figure 5. Finite Coordinate Space

events using the `event` AF, and event schedules using the `evsched` AF. The document instance associates a data object with the event. The semantics and the manner in which the events are rendered are defined by the application.

To represent spatio-temporal relationships, we define a FCS consisting of three axes: the *x* and *y* axes for spatial relationships of objects on screen, and the *time* axis for temporal relationships (Figure 5).

The representation of this idea in the database requires the definition of a number of HyTime architectural forms. Of the 69 architectural forms defined as part of the HyTime standard, we have implemented *hydoc*, *dimspec*, *axis*, *event*, *link*, *fcs*,, *exlist*, and *evsched*. `HyElement` is the supertype of all HyTime types in the element type system. Its immediate subtypes correspond to the AFs we have implemented. Following the HyTime standard, all HyTime elements in the DTD must have `ID` and `HyTime` attributes. The `ID` is used as a unique identifier to make element references possible. The `HyTime` attribute specifies the architectural form to which the element belongs. Types `Video`, `Stext`, and `Audio` are directly derived from the HyTime type `Event_AF`. They use HyTime events for synchronization purposes.

3.4 Meta-Types

The meta-type system roughly parallels the element type system. Like the built-in element types, which function as abstract supertypes for the concrete DTD-specific element types, a directed acyclic graph of built-in meta-types provides the supertypes for the concrete meta-types that are generated by the Type Generator.

For each concrete element type that is generated from a DTD, a corresponding meta-type is generated. These meta-types perform two important tasks:

1. They store meta information about the elements in the DTD, such as the element names, their attributes, and the supertypes from which the corresponding element types are derived. This information is necessary for instantiating the appropriate element instances and setting their attributes.
2. They define virtual *create* functions to instantiate persistent objects representing document element instances. These are referred to as *instantiation methods*.

Single instances of these meta-types are created as persistent objects in a database destined for document instances conforming to a particular DTD. The instantiation methods that are implemented in the generated code can create the corresponding element objects, parameterized according to the data extracted by an SGML document parser during the automated document insertion process. Virtual function resolution ensures that any meta-type object will create the appropriate object(s) when its instantiation method is invoked.

The meta-type system is rooted in the base type `ElementType`. This base type contains all the meta information: name, attributes, and supertypes. It also declares and defines member functions that fall into the following categories:

1. **The instantiation methods**. These are virtual functions, mentioned previously, which get redefined in the generated meta-type. There are separate functions for structured elements (with a complex content model) and unstructured elements (with a simple content model).
2. **Functions for setting attributes**. There are non-virtual functions that loop through attribute lists or provide general attribute setting functionalities. A virtual function for setting specific attributes is redefined in the generated element types.
3. **Functions for determining special attributes**. These functions take care of special attributes that are used by the HyTime and MM architectural forms.

The top levels of the built-in meta-type system are shown in Figure 6. In all, there are five levels in this type hierarchy, which utilizes multiple inheritance in parts of the lower levels. This meta-type system is described in detail in [EM96].

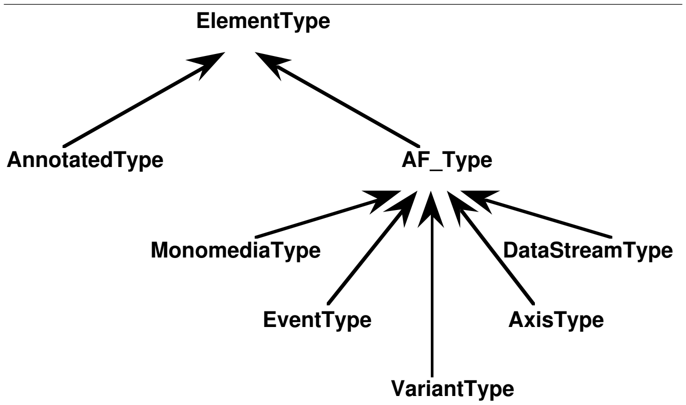

Figure 6. Partial Meta-type System

4. SUPPORT FOR MULTIPLE DOCUMENT STRUCTURES

One fundamental requirement of a multimedia DBMS for SGML/HyTime documents is that it should be able to support multiple DTDs by creating types that are induced by these DTDs. This is essential if the multimedia DBMS is to support a variety of applications. The system must be able to analyze new DTDs and automatically generate the types that correspond to the elements they define. In addition, the DTD must be an object in the database so that users can run queries like "Find all DTDs in which a 'paragraph' element is defined."

The components that have been implemented to support multiple DTDs are depicted in Figure 1. A **DTD Parser** parses each DTD according to the SGML grammar defined for DTDs. While parsing the DTD, a data structure is built consisting of nodes representing each valid SGML element defined in the DTD. Each DTD element node contains information about the element, such as its name, attribute list and content model. If the DTD is valid, a **Type Generator** is used to automatically generate C++ code that defines a new ObjectStore type for each element in the DTD. Additionally, code is generated to define a meta-type for each new element type. Moreover, initialization code is generated and executed to instantiate extents for the new element objects and to create single instances of each meta-type in the specified database. A `Dtd` object is also created in the database. This object contains the DTD name, a string representation of the DTD, and a list of the meta-type objects that can be used to create actual element instances when documents are inserted into the database.

There are two important problems that need to be addressed in this process. Both of these are abstraction problems that can reduce the complexity of the multimedia type system and therefore reduce maintenance time and errors. First, if two or more DTD elements in the same DTD definition share common features, then this feature should, ideally, be automatically extracted and promoted to an abstract superclass. For example, in a prototype news-on-demand type system, the two types, `Video` and `Audio` both shared a common duration attribute, so the abstract supertype `Temporal` was created to promote this feature. If the feature is a common content model, this factoring is straightforward. Otherwise, the problem is harder to solve. Even if attributes of different elements have the same name and specification, they may be semantically unrelated.

Second, common element definitions across different DTDs should be represented by a common type in the type system. However, there is no easy solution to this problem since it leads to the well-known semantic heterogeneity problem, studied extensively within the multidatabase community. Briefly, the problem is one of being able to determine whether two elements are *semantically* equivalent. This problem has also been studied in the programming languages field, where there are many different definitions for type equivalence. For example, two types are name equivalent if they have the same name. However, this would not be a good definition of type equivalence in our model since two different DTDs might use the same name to describe semantically different elements, for example, a `Signature` in a Thesis DTD and a `Signature` in a Symphony DTD. Similarly, programming languages define two types to be structurally equivalent if the components recursively have the same names and types. This may also lead to faulty equivalencies. For example, `Caption` and `Title` could be structurally equivalent, each having a content model that is `#PCDATA`. However, they are semantically different, a difference that may only become clear in the context of what composite objects can contain them. Since this is not a trivial problem, we have chosen to give up some abstraction in favor of a semantically "safe" type system.

This does not mean, however, that we have completely abandoned type re-use across DTDs. We re-use the atomic types such as `AtomicAudio`, `AtomicImage` and `AtomicText`, as well as the high-level abstract supertypes such as `TextElement`, `Structured` and `HyElement` in the element type system. These types are safe to re-use because they have well defined semantics that appear across many document types. For the specific elements in a given DTD, we create new types derived from the abstract supertypes. Name conflicts between elements in different DTDs are resolved automatically by using the DTD name as a prefix during type creation (e.g. `ArticleSection`, `BookSection`).

In addition to re-use through abstract supertypes, we have provided a mechanism for the DTD writer to specify which elements in a DTD are to be treated as reusable across all the DTDs in the system. This is specified by explicitly using a “reusable” attribute in an element's definition. Then, if a reusable type of that name does not exist in the system at the time the DTD is instantiated, it gets created along with reusable types for all its implicitly reusable descendants.

If such a reusable type (or any of its descendants) already exists in the system, the incoming element definition (and its descendants) is validated to ensure that the old and the new type definitions are compatible. If compatible, no new class is created. If not, an error report is generated containing valid definitions for the invalid elements which can later be pasted into the DTD in place of the invalid definitions. A dedicated database maintains information about reusable element definitions, as well as a record of which DTDs utilize a particular reusable element.

Metatype classes are also generated for each reusable element. An instance of a metatype class is capable of creating an object of its associate type during document instantiation. However, for annotated elements (generally elements with textual content) a single reusable metatype instance is not sufficient since the annotation of an object must also be recorded in a `DocumentRoot` object particular to the DTD (e.g. `BookRoot`). In these cases, metatype classes derived from the reusable metatype classes must also be generated in the DTD specific code. An instance of the specific metatype is needed for each DTD in the database using the reusable annotated element. For non-annotated elements, a single instance of a reusable metatype is sufficient for creating objects of reusable types for any DTD.

This facility for reusable types can significantly reduce the amount of generated code in a system that contains multiple DTDs sharing commonly defined elements. Such reusable elements can later be uniformly queried across multiple document types.

The **DTD Manager** in Figure 1 stores the DTD in the database as an enhanced object that can be used for parsing documents and for other purposes. This is done after type creation. As soon as a DTD is stored in the database, SGML documents of that type can be inserted. A DTD object contains the name of the DTD, the DTD as a character string, and a list of meta-type objects used during document instantiation.

5. AUTOMATIC DOCUMENT INSERTION

One of the serious shortcomings of many multimedia DBMS projects is the unavailability of tools for the insertion of documents into the database. Many systems have facilities for querying the database once the documents are inserted in it, but no tools exist to automatically insert documents. This is generally considered to be outside the scope of database work. One of our sub-projects has concentrated on coupling the multimedia database with a retrofitted SGML parser, so that SGML documents can be created, using existing authoring tools, and automatically inserted into the database.

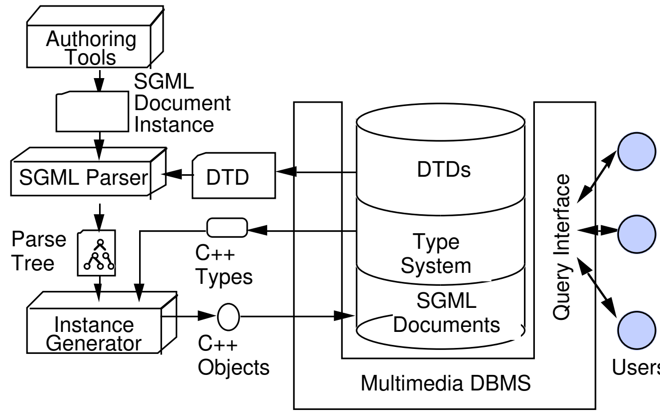

Figure 7. Automatic Document Insertion

The general architecture for this coupling is depicted in Figure 7. The **SGML Parser** accepts an SGML document instance from an **Authoring Tool**, validates it, and forms a parse tree. The **Instance Generator** traverses the parse tree and instantiates the appropriate objects in the database corresponding to the elements in the document. These are persistent objects stored in the database that can be accessed using the query interface.

The parser is based on a freeware application called *nsgmls*[3]. The parser was modified to incorporate the following changes:

1. The DTD used for parsing the document instance is fetched from the multimedia database.
2. The output is passed to the Instance Generator as a parse tree instead of producing parsed text output. This output includes a text string for the document that is stripped of markup, together with a linked list of nodes containing annotations into the string, an attribute list, and pointers to parent and next nodes.
3. The parser does not produce any output unless the document is error-free.

While the parser component is independent of any particular DTD, the Instance Generator is DTD

[3] This parser is developed by James Clark and available on the Internet.

specific. The specific library linked to the parser is the one built from code generated for the specific DTD.

Much of the instance generating code in the system was discussed above in Section 3.4 on meta-type types. Object instances of all the meta-types required by a DTD are persistently stored in the DTD object. For each node in the parse tree, the appropriate meta-type object is found by querying the DTD's meta-type list. The appropriate `CreateObject` method is invoked on this object, which in turn invokes a method for setting attributes according to the data in the parse tree node.

Though most of the instance creating behavior is implemented in the meta-types, some behavior is implemented directly in the element types. The element types may implement three instance creating behaviors: `SetChild`, `SetSpecificAttribute`, and `ResolveRef`.

The `SetChild` behavior sets pointers in the parent of a newly created object. Each structured object maintains a list of all its child objects as well as lists for each differing child type. The `SetAttribute` method in `ElementType` loops through all the attributes and invokes `SetSpecificAttribute` methods specific to each element type. The `ResolveRef` method implements the behavior of resolving an element's references. If the attribute's declared value type is `IDREF`, the resolution of the attribute is deferred until the end of the document instantiation process. This reference resolving approach is similar to a pass-and-a-half compiler.

6. RELATED WORK

Among the numerous multimedia DBMS described in the literature, there are three object-oriented systems supporting SGML/HyTime documents which warrant comparison with our own.

6.1 HyperStorM

HyperStorM (Hypermedia Document Storage and Modeling) project currently undertaken at GMD-IPSI (Germany) has similar goals to ours but is built upon the object-oriented DBMS VODAK[4]. The Structured Document Database component [BA96] of HyperStorM, which investigates various object-oriented technologies for structured documents, is the research effort that most closely resembles our system.

The current version [Böh95] provides a hybrid approach for physical document representation that is configurable by the DTD designer. In this version, the DTD designer can use special attributes to specify which elements should be *flat*. In each flat element, the textual content of all nested elements is stored as one continuous string, including markup. Indexing mechanisms are similarly configurable.

Whereas the Structured Document Database uses VODAK's metaclass feature to dynamically add classes to the database schema, our system dynamically generates C++ code that gets linked into ObjectStore applications. Moreover, we do not rely on external data files, such as configuration files, outside the control of the database for document insertion or querying.

6.2 VERSO

VERSO [CACS94], developed at INRIA in France, is an object-oriented database system for SGML documents. It is built on top of O_2 to exploit its sophisticated type system and extensible query language O_2SQL.

While VERSO models SGML constraints in the data model, our system enforces those constraints through the SGML parser. We also have HyTime support explicitly built into the system. Our textual storage model is also inherently more efficient for document retrieval than the fragmented model in VERSO.

6.3 HyOctane

A distributed multimedia information system being developed at the University of Massachusetts [KRRK93] consists of an SGML parser, an ObjectStore database for storing HyTime documents, and the HyTime engine HyOctane for accessing and presenting the document instances. The database schema is based on one specific DTD, the HDTD (a HyTime conforming DTD for interactive multimedia presentations).

In contrast to our system, the HyOctane system does not support the automatic addition of new DTDs to the system. However, the authors claim their system can be easily extended to support other DTDs.

7. CONCLUSIONS

In this paper we have described an object-oriented multimedia DBMS that is compliant with SGML/HyTime. The system, as described here, is operational and has been integrated with the other components (continuous media file system, QoS negotiation, synchronization and client modules) of the Broadband Services Project to provide a demonstration prototype. The initial implementation platform was IBM RS6000 workstations running AIX 4.x. The implementation language (for the multimedia

[4] VODAK is itself built upon ObjectStore. It utilizes ObjectStore's persistent storage management, but implements its own modeling language (VML) and query language (VQL). The application programmer codes in these languages, instead of using C++. The VML code is converted by the VODAK compiler into C++ code that is linked to the VODAK and ObjectStore libraries.

DBMS component) was C Set++, which is the IBM's implementation of ANSI C++ for the AIX platform.

The primary contributions of this research, as reflected in the capabilities of the system, are the following:

1. It is an object database that can store not only meta-data about multimedia objects, but also multimedia objects themselves. This facilitates querying over multimedia data.
2. It is implemented as an open and extensible system with facilities to incorporate outside repositories. The current version supports linkages to a continuous media server; in the future, we intend to incorporate more general external repositories.
3. The document model is compliant with SGML/HyTime standards, enabling the use of tools developed for these standards.
4. The system can support multiple DTDs by storing objects from different document types in one logical database.
5. It provides facilities for automatic parsing of multimedia documents and their insertion into the database.

We are currently involved in extending the system in a number of directions. First, we are improving the modeling capabilities of the system. In this regard, we are working on the design of a document versioning mechanism[5] and the automatic generation of indexes and queries.

The second direction is the expansion of system capabilities by introducing more of the features that are common in traditional DBMSs. In particular, we are building multimedia-specific query models and languages that are easier to use and more efficient than general query models and languages. We have defined, and are in the process of implementing, a text-based query language, called MOQL [LÖSO96], based on the standard OQL query language defined by ODMG [Cat95]. We are also building a visual interface on top of this language that will support dynamic and incremental querying. In order to optimize the execution of multimedia queries, we are developing content-based indexing techniques to facilitate access to images based on the spatial relationships among objects. For example, the user should be able to ask for "all images that show Bill Clinton *in front of* the White House and *next to* Jean Chretien". Our current approach involves extensions to 2D strings [CSY87] in order to improve the search time [NÖL96].

[5] ObjectStore's versioning facility was architecturally flawed and consequently removed from its latest release, 5.0

Finally, we are studying architectural improvements to our system. Along these lines, we are investigating appropriate distribution architectures whereby the multimedia data that is stored on multiple servers can be transparently accessed by users. In this regard, system interoperability becomes an important issue since multimedia data is likely to be stored in varying repositories in different formats. The results of this work will be reported in future papers.

REFERENCES

[Böh95] K. Böhm. "Building a Configurable Databse Application for Structured Documents," Arbeitspapiere der GMD 942, GMD-IPSI Darmstadt, 1995.

[BA96] K. Böhm and K. Aberer. "HyperStorM—Administring Structured Documents Using Object-Oriented Database Technology," In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 547, June 1996.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. "From Structured Documents to Novel Query Facilities," In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 313-324, 1994.

[Cat95] R.G.G. Cattell (ed.). *The Object Database Standard: ODMG-93, Release 1.2*, Morgan Kaufmann, 1995.

[CSY87] S.K. Chang, Q.Y. Shi and C.W. Yan. "Iconic Indexing by 2D Strings," *IEEE Trans. Pattern Analysis and Machine Intelligence*, 9(3): 413–428, 1987.

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

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

[EM96] S. El-Medani. Support for Document Entry in the Multimedia Database, M.Sc. Thesis, University of Alberta, Department of Computing Science, 1996. Also available as Technical Report 96-23 (http://ftp.cs.ualberta/pub/TechReports/1996/TR96-23).

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

[ISO92] International Standards Organization. *Hypermedia/Time-based Structuring*

Language: HyTime (ISO 10744), 1992.

[KRRK93] J.F. Koegel, L.W. Rutledge, J.L. Rutledge, and C. Keskin. "HyOctane: A HyTime Engine for an MMIS," In *Proceedings of the ACM Multimedia Conference,* pages 129-135, 1993.

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

[LÖS96a] J.Z. Li, M.T. Özsu, D. Szafron. "Spatial Reasoning Rules in Multimedia Management System," In *Proc. International Conference on Multimedia Modeling*, pages 119-133, November 1996.

[LÖS96b] J.Z. Li, M.T. Özsu, D. Szafron. "Modeling of Video Spatial Relationships in an Objectbase Management System," In *Proc. International Workshop on Multimedia DBMS*, pages 124-133, 1996.

[LÖS96b] J.Z. Li, M.T. Özsu, D. Szafron. "Modeling Video Temporal Relationships in an Object Database Management System," In *Proc. SPIE Multimedia Computing and Networking* (MMCN97), pages 80-91, February 1997.

[LÖSO96] J.Z. Li, M.T. Özsu, D. Szafron and V. Oria. "MOQL: A Multimedia Object Query Language", in preparation.

[NMH96] G. Neufeld, D. Makaroff, and N. Hutchinson. "Design of a Variable Bit Rate Continuous Media Server for ATM Network," In *Proc. SPIE Multimedia Computing and Networking* (MMCN96), January 1996.

[NÖL96] Y. Niu, M.T. Özsu and X. Li. "2DS Tree: An Index Structure for Content-Based Retrieval of Images", submitted to *ACM Multimedia Conference*, 1997.

[ÖSEV95] M.T. Özsu, D. Szafron, G. El-Medani, and C. Vittal, "An Object-Oriented Multimedia Database System for News-on-Demand Application", *Multimedia Systems*, 3: 182-203, 1995.

[PS97] P. Pazandak and J. Srivastava. "Evaluating Object Database Management System Functionality to Support Multimedia,"*IEEE Multimedia*, Fall 1997.

[Sch96] M. Schöne. A Generic Type System for an Object-Oriented Multimedia Database System, M.Sc. Thesis, University of Alberta, Dept. of Computing Science, 1996. (http://ftp.cs.ualberta/pub/TechReports/1996/TR96-14).

[WLE+97] J.W. Wong, K.A. Lyons, D. Evans, R.J. Velthuys, G.v. Bochmann, E. Dubois, N.D. Georganas, G. Neufeld, M.T. Özsu, J. Brinskelle, A. Hafid, N. Hutchinson, P. Iglinski, B. Kerhervé, L. Lamont, D. Makaroff, and D. Szafron. "Enabling Technology for Distributed Multimedia Applications", *IBM Systems Journal*, 36(4), 1997 (in press).