University of Alberta

Support for Document Entry in a Multimedia Database

by

Sherine El-Medani
Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta
Canada T6G 2H1

Technical Report TR 96–23
September 1996

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

Abstract

This Technical Report describes a document entry system that supports the automatic insertion of documents with arbitrary types into a multimedia database. The documents conform to the international standards of SGML and HyTime. This is achieved by instantiating objects in the database to represent the document components. These are persistent objects that can be queried and retrieved using a query interface.

To achieve this goal, the database has to be queried dynamically to retrieve meta-information about the document structure. A generic meta type system was designed to model such meta-information. This type system consists of a number of built-in types that model the characteristics common to SGML document components. Whenever support for a new class of documents is added to the system, meta types modeling the characteristics specific to this class are dynamically generated and added to the meta type system. To insert documents conforming to this class into the database, two system components were developed: the SGML Parser and the Instance Generator. The SGML Parser validates the SGML documents to ensure database consistency. The Instance Generator instantiates the appropriate database objects to represent the document.

Contents

List of Figures

Chapter 1

Introduction

In recent years, there has been a rapid increase in the use of multimedia applications in a diversity of fields including education, entertainment, libraries, news-on-demand applications, advertising, medicine, teleconferencing and many others. One of the major reasons for this popularity is the ability of multimedia systems to incorporate and integrate information from diverse media sources such as graphics, audio, video, images and text. This presents users with various channels for communication and information delivery, thus allowing for a broader and richer means of interaction.

However, as the popularity of multimedia applications has increased, so have the amount and the complexity of avaliable data. Despite the diverse nature and the vast amount of data embedded within multimedia applications, there is currently no uniform way of managing and accessing such data efficiently. Different multimedia file systems provide different solutions for solving these problems. The problem becomes even more complicated for multi-user and distributed multimedia systems where concurrency control, access control, application independence and data distribution become important issues. Since these issues have been successfully addressed in database systems for years, it is only reasonable for multimedia applications to make use of database management systems (DBMS) instead of attempting to solve the same problems independently. Multimedia applications can benefit from standard DBMS services such as data abstraction, application neutrality, fault tolerance, concurrency control and data distribution (in distributed DBMS technology) [OSEMV95]. A DBMS will also provide these applications with the ability to go beyond the simple retrieval of information to high-level accesses through queries.

Despite all these benefits, the use of database technology in multimedia applications is currently quite restricted. This is what motivated our work in developing a multimedia DBMS at the Laboratory for Database System Research at the University of Alberta. The goal of our research is to provide a "complete" multimedia database system. It is part of a major project funded by the Canadian Institute of Telecommunications Research (CITR) which started in 1993 with a six year duration (see Chapter 3 for more details). This Master's thesis is a step towards this goal. It aims at supporting document entry into the multimedia database. Supporting document entry involves instantiating objects in the database to represent the documents' components. These are persistent objects that can be queried and retrieved using a query interface. Since multimedia applications require different document structures, our database design allows multiple document structures to be represented and stored in the database. Complying to this design, the document entry system supports the insertion of different types of documents in the database, thus providing the flexibility required by multimedia applications.

1.1 Motivation and Design Criteria

One of the shortcomings of many of the existing DBMS is the unavailability of tools for the insertion of multimedia documents in the database. This is generally considered outside the scope of the DBMS. It is our belief that document entry is an important DBMS tool.

If a multimedia DBMS is to be used for multimedia applications, it has to provide a method for inserting documents in the database. Manually inserting these documents is not a realistic option given the large volume of data.

Another reason for supporting document entry is to ensure database consistency. Database consistency is considered an important database concept that has been extensively researched over the years. Concurrency control techniques have been used in DBMS to ensure data integrity and consistency. However to maintain a consistent database, one has to start with a consistent database. Supporting document entry can ensure the consistency of the database by ensuring that the inserted documents are error-free.

This thesis provides an automatic way of inserting documents in the multimedia database conforming to multiple document structures. The system meets the following design criteria:

- **Flexibility:**

  As mentioned earlier, multimedia applications are diverse in nature and purpose. It is almost impossible to have one document structure to represent the different types of multimedia documents. Moreover, if one exists it will be too general and inefficient to serve the application requirements. Therefore, supporting multiple document structures is almost inevitable. With multiple document structures, the automatic insertion of documents conforming to the different structures has to be supported. This provides the flexibility and generality needed for multimedia applications.

- **Data Consistency:**

  The system ensures database consistency by validating multimedia documents before insertion. If the document does not conform to a document type definition (DTD) in the database or if it already exists in the database, it is rejected.

- **Adherence to International Standards:**

  Since multimedia sources are diverse, there is a need for a standard document representation. This is a way to support heterogeneity of tools. The Standard Generalized Markup language (SGML) standard [Org86] has been chosen for this representation. Being an ISO standard, there are a number of authoring tools and conversion tools that support SGML. Also, SGML is the basis of the Hypermedia/Time-Based Structural Language (HyTime) standard [Org92], an ISO standard that supports hypermedia documents (e.g., links, video, audio, etc).

1.2 Thesis Scope and Organization

This thesis describes an automatic way of inserting documents with different document types into a multimedia database. The major contributions of this thesis are:

- The design of the meta type system which is used to store meta information about the SGML document elements. This meta information is necessary to instantiate objects in the database to represent the document.
- The dynamic extension of the meta type system to support new document types. This is achieved by generating C++ code whenever a new document type definition (see section 2.1.1) is added to the system. The generated code defines new meta types to represent the new elements defined in the DTD.
- The dynamic extension of the element type system by generating member functions necessary for modeling the document instances.
- The development of the Document Entry System which is composed of the SGML Parser and the Instance Generator components. The SGML Parser is used to validate the SGML documents before inserting them in the database. It forms a parse tree for the document structure. The Instance Generator uses the parse tree to automatically instantiate objects in the database.

The thesis is organized as follows. Chapter 2 gives an overview of the SGML/HyTime standard. Chapter 3 introduces the monomedia project components and their general architecture. The design of the generic multimedia type system is discussed in Chapter 4. The type system consists of three main components: the Atomic Type System which models the basic multimedia objects, the Element Type System which provides the abstract classes from which the document components are subtyped, and the Meta Type System which is used to store some meta information about the DTD elements. Chapter 5 describes the system for handling multiple DTDs and its interface with the document entry system. Chapter 6 describes the document entry system, focussing on the SGML Parser component. The Instance Generator component is discussed in Chapter 7. A review of some related work is introduced in Chapter 8 together with a comparison between their approaches and the ones adopted by our group. Conclusions and future directions are discussed in Chapter 9.

Chapter 2

SGML/HyTime Overview

The Standard Generalized Markup language (SGML) and the Hypermedia/Time-Based Structural Language standards ([Org86]; [Org92]) have been adopted as the document representation in the *CITR Project*. Both are ISO standards that are widely used and accepted in the publishing field. SGML supports textual documents whereas HyTime adds support for hypermedia documents.

SGML was chosen because of its suitability for multimedia applications. The strengths of SGML can be summarized as follows:

- SGML has been widely used in both conventional and multimedia publishing. That's why there is a variety of authoring tools and conversion tools that support SGML. Multimedia authors can make use of these tools to facilitate the process of document creation. Also, the availability of different conversion tools can provide a "smooth" transition for the existing multimedia documents from their formats to the SGML format.

- SGML separates structure from layout. Its generic markup schema enables the addition of information to the text to indicate the logical structure of the document. This makes it easily mapped to the database schema and thus facilitates the automation of data modeling in the database.

- SGML's unambiguous format makes it more portable across multiple platforms. Multimedia documents can be easily interchanged between different systems giving the multimedia authors the flexibility to work in their favorite environments.

This chapter provides an overview of the SGML/HyTime standards. Not all the features are discussed here, the interested reader is referred to [vH94], and [NKN91] for more details.

2.1 SGML Overview

SGML was introduced to allow document authors to communicate in a standard way. This is achieved by incorporating information in the document indicating its notation as well as its structure. A document author is allowed to reference other documents (hyperlink) regardless of the heterogeneity of their notations. It then becomes the responsibility of the processing application to interpret and integrate the different documents with the different notations.

SGML also provides a method to express documents in terms of their logical structure. It is considered a meta language that provides a means to express the document's syntax rather than the syntax of a specific document type. For every document type, there should exist a *Document Type Definition (DTD)* that specifies the syntax of the valid documents of that type.

A typical SGML system consists of SGML documents and SGML software. SGML software includes:

- **An SGML Editor or Conversion Tools**: An SGML editor can be used to help document authors in composing SGML documents. The SGML documents must conform to a DTD in the system. Alternatively, conversion tools can be used to convert documents from other notations such as LaTeX or MS Word format to the SGML notation. The success rate of conversion is dependent on how strictly the structure is enforced in the original document.

- **A DTD Parser**: A DTD parser is used to parse DTDs to ensure that they are syntactically correct. The DTD parser is not an essential part of an SGML installation since the DTD will be parsed by an SGML parser during the parsing of document instances. However, it might be useful for document designers to test their DTDs before composing documents. Also, in some processing applications such as our multimedia database application, parsing the DTD before the document might be of special importance.

- **An SGML Parser**: An SGML parser is used to validate SGML instances to ensure that they conform to their DTDs. If the document does not conform, a list of errors should be produced. SGML parsers help document authors find errors and prevent markup misuse.

- **A Processing System**: Once an SGML document is verified by a parser, it can be processed by the processing application. The processing of SGML documents is not under the control of the SGML standard.

An SGML document consists of three major components: *the document instance*, *the document type definition (DTD)* and the *SGML declaration*. A detailed discussion of the different components is provided in the rest of this section.

2.1.1 Document Type Definition

As mentioned earlier, a DTD is used to define the valid syntax for a specific document type. It defines the logical components that can appear in the document and their hierarchical/sequential relationships. The logical components of the document are expressed using "generic markup" where tags are used to identify the names of the different components.

A typical DTD defines 3 types of markup commands: *elements*, *attributes* and *entities*. Consider the following example:

```
<!ELEMENT article   - - (docs_alts?, frontmatter, async, sync?)>
```

```
<!ATTLIST article
          id         ID        #REQUIRED
          language   CDATA     #IMPLIED>
<!ENTITY  SGML      "Standard Generalized Markup language">
```

As shown in the example, every DTD command starts with "<!" followed by a keyword (*ELEMENT*, *ATTLIST*, *ENTITY*) to denote the type of the command. The content of the command is then given followed by a ">" to close the command.

An element command is a production rule which is defined for every element in the DTD. The name of the element is specified on the left hand side of the production rule. The *omitted tag minimization* information is then represented by two characters; one for the start-tag and one for the end-tag. A minus sign '-' is used if the tag should not be omitted and a letter 'O' is used if the tag may be omitted. The right hand side of the production rule is called a *content model*. The content model specifies which elements can occur inside the element's content. Element names in the content model are followed by *occurrence indicators* such as ("*", "+" and "?") and are linked together by *connectors* such as ("&", "|" and ","). Occurrence indicators specifies the number of times an element can occur inside the content:

"*"	the element can occur zero or more times.
"+"	the element can occur one or more times.
"?"	the element is optional (can occur zero or one time).

Connectors indicate the relationship between the elements in the content model:

“,” elements separated by commas must occur in the same order.

“&” elements separated by “&” must appear in the element’s content but in any order.

“|” Only one of the elements separated by “|” should appear in the element’s content.

Elements that do not have any sub-elements are called *leaf elements*. Leaf elements can have one of the following content models:

#PCDATA stands for parsed character data. These elements contain the textual contents of the document. Markup inside the elements’ text should be interpreted by the parser.

CDATA stands for character data. Markup inside the elements’ text should be ignored by the parser.

RCDATA stands for replacable character data. Same as CDATA except that entity references are recognized by the parser. Entities are discussed at the end of this section.

EMPTY These elements do not have any content.

Optionally, attributes can be defined on elements in the DTD using an *attribute command*. Attributes do not belong to the content of an element but they provide more information about it. An attribute statement consists of the name of an element in the DTD and a list of all attributes that belong to this element. Each item in the list consists of the attribute name, the declared value and the default value. Possible declared values include: CDATA, ENTITY, ID, IDREF and IDREFS. An attribute with an ID declared value is used to uniquely identify an element. This element can be cross-referenced from other elements using an IDREF or IDREFS attribute. Attribute defaults include:

- #FIXED — Should be followed by the attribute's value. If the attribute's value is specified, it should be always equal to the fixed value.
- #REQUIRED — The attribute's value should always be specified in the document.
- #CURRENT — If the attribute's value is not specified, the most recent value should be used.
- #IMPLIED — If the attribute's value is not specified, the processing application should choose a default.

An entity is a unit of information that may be referred to by a symbol in a DTD or in a document instance. Entities are defined by an entity command. There are two types of entities: *parameter entities* and *general entities*. Parameter entities are only used within markup declaration, mostly within DTDs. General entities can be used anywhere within the DTD or the document instance. They can be further classified into: *internal* and *external* entities. Internal entities are used as a shorthand notation for text strings. In the above example, the string "Standard Generalized Markup Language" can be defined as the replacement text for an entity "SGML". External entities are used to include external files. They can be divided into **local** or **public** entities. Local entities are usually specific to the system on which they are installed. Public entities are public domain entities that are registered in a standard way and available for public use.

2.1.2 Document Instance

The document instance consists of the contents of the document together with markup information that identifies the different logical components of the document. Since every SGML instance should comply to a specific DTD, a reference to the DTD name must also be included. The boundaries of the document components are delimited by *tags* that specify their beginning and end positions. For example, consider the following SGML document:

```
<!DOCTYPE thesis SYSTEM "thesis.dtd">
<thesis>
   <frontmatter>
      <title> Support for Document Entry in the MM Database </title>
      <author> Sherine El-Medani </author>
      <degree> Masters of Science </degree>
   </frontmatter>
   <tableofcontents>
   <body>
      <chapter>
         <section> ... </section>
         <section> ... </section>
      </chapter>
      ...
   </body>
</thesis>
```

The DTD filename and the root element name is specified in the DOCTYPE statement. Since any valid element in the DTD can act as a root, the root element should be specified for every document instance.

2.1.3 SGML Declaration

The SGML declaration is used to define the details of how SGML should be applied to the document. For example, it specifies which character set (e.g. ASCII or other) and delimiters (e.g. $<$, $>$, $<$ /) can be used in the document. It is usually common to all documents in an SGML declaration. However, if no specific SGML declaration is associated with a document, a default declaration can be used.

2.2 HyTime Overview

HyTime is an ISO standard built on top of SGML to support hypermedia documents. It provides an expressive mechanism to represent the structure of multimedia, hypertext, hypermedia, time-based and space-based documents. It enables multimedia documents to communicate regardless of their heterogeneous notations. It also provides a standard technique to express the temporal and spatial relationships among the logical components of a multimedia document.

HyTime defines a set of *architectural forms (AF)* that can be used when designing a DTD. Whenever a DTD element needs to implement a HyTime functionality, it should conform to a HyTime AF. This can be achieved by defining a `HyTime` attribute with a fixed value equal to the AF name. For example,

```
<!ELEMENT  link      - - (#PCDATA)>
<!ATTLIST  link
           Hytime      NAME     #FIXED  ilink
           linkends    IDREFS   #IMPLIED >
```

defines an element `link` which conforms to the independent link (ilink) AF. An independent link is used to define a hyperlink with any number of link ends. Note that, for `link` to conform to the ilink AF, it has to define the `linkend` attribute which is defined in the HyTime standard.

HyTime architectural forms are grouped together into a number of modules each of which describes a concept or a group of related concepts. These modules are: the base module, the location address module, the hyperlinks module, the finite coordinate module, the scheduling module and the rendition module. The modules are designed in a hierarchical fashion where each module can use features defined in modules lower in the hierarchy.

Only those modules of HyTime which are appropriate to a certain document need to be declared in the document. For example for a DTD to support hyperlinks, the following declaration should be included[1]:

```
<?Hytime    support   base>
<?Hytime    support   hyperlinks>
```

The rest of this section provides a brief discussion of some of the HyTime modules namely the base module, the location address module and the hyperlinks module. The reader is referred to the HyTime standard for more details [Org92].

2.2.1 The Base Module

This module must always be supported by a HyTime compliant system. It includes facilities that are needed by almost all the other modules. These facilities include:

- **Hyperdocument Management Facility**

  It defines the basic HyTime concepts (such as architectural forms) and the representational and document management conveniences. For example, it defines the `HyDoc` architectural form which is used to represent the root element of a HyTime document. It also defines the `ID` attribute which is required by the definition of some other architectural forms including the `HyDoc` architectural form.

- **HyTime Identification Facility**

  It is used to solve name collision problems by allowing the replacement of HyTime-specific identifiers with user-defined identifiers [NKN91].

- **Coordinate Addressing Facility**

[1]Note that, the base module should be supported by any system that supports HyTime.

This facility is required by the address location module and finite coordinate module. It allows the dimensions and positions of events to be scheduled and allows document locations to be addressed by position.

- **Activity-Tracking Facility**

  It provides a standard way by which the author of a document containing a hyperlink can inform the owner of the linked-to document of the existence of the link and to request notification if the document is modified.

2.2.2 The Location Address Module

The location address module provides methods for addressing and referencing data. SGML attribute of type ID(s) and IDREF(s) can be used to identify/reference an element. However, this technique has a number of limitations. First, only the contents of an entire element can be referenced; for example, a substring of a text element cannot be referenced using this method. Second, only elements within the same local document that have an `ID` attribute can be referenced. This is because the uniqueness of IDs is guaranteed only within the document's name space and not across documents. The address location module extends SGML's reference capability to accommodate more general cases. HyTime supports three kinds of addressing:

- **Addressing by Name**

  HyTime defines the *named location address* architectural form that provides a method to address named SGML entities and uniquely identified SGML elements across documents. This is achieved by modeling the information needed to locate the external file and to retrieve the named element or entity within the file.

- **Addressing by Position**

  HyTime allows addressing of objects by their position in some universe by specifying the position and the units of measurement (quanta) in the universe. For example, a substring can by located by specifying the quanta to be characters or words, the starting location of the parent string and the start position of the substring. Similarly, a tree node in the document hierarchy or a specific object in a list can be located.

- **Addressing by Semantic Construct**

  HyTime provides two architectural forms which allow semantic addressing, namely the *attribute location address* AF and the *notation-specific location address* AF. Using the first AF, an element can be located based on an attribute value; for example, the first object with a `location` attribute of value Edmonton. The second AF allows a subset of a data object to be located even if the data is not in SGML. For example, locating the second polygon from the right in a figure. Note that, a system capable of interpreting the data has to be used for the data location.

2.2.3 The Hyperlinks Module

HyTime provides five kinds of hyperlinks classified according to the purpose of the link and the number of link ends allowed:

- **Independent Links (ilinks)**

  The most general purpose hyperlink architectural form as it can have any number of link ends.

- **Property Links (plinks)**

  plinks can only have two link ends. It is usually used to associate a property with an element.

- **Conceptual Links (clinks)**

  Like plinks, clinks always have two link ends. However, one of the link ends must be the clink's own location in the document. For example, clink can be used to represent a text's footnote.

- **Aggregate Location Links (agglinks)**

  It links multiple locations together in such a way that they are treated as a single aggregate location.

- **Span Links (spanlinks)**

  It allows the content of continuous SGML elements (possibly the whole SGML document) to appear as a continuous string regardless of the intervening SGML markup.

Chapter 3

Research Framework

3.1 The CITR Project

The work reported in this thesis is part of the Broadband Services Project funded by the Canadian Institute for Telecommunications Research (CITR) - a Network of Centres of Excellence funded by the Government of Canada. The project is a multi-university project spanning a six years duration starting in 1993. It aims at researching enabling technologies for distributed multimedia applications exploring broadband communications technologies. The research consists of 4 major components: Continuous Media File System (CMFS) (University of British Columbia), Quality of Service (QoS) Negotiation (University of Montreal), Synchronization (University of Ottawa) and Database (University of Alberta). The system architecture design and integration work is done at the University of Waterloo.

The database component developed at the University of Alberta focusses on data management issues such as application independent storage and management of data, versioning and meta data management.

3.2 The News-On-Demand Application

In the first phase of the project, a prototype was developed using *news-on-demand* as the target application. Two research areas were investigated: modeling the news-on-demand data in the database [Vit95] and the design of a visual query interface to the database [EM95]. Modeling the data involved the design of a suitable DTD for news articles and the implementation of the type system corresponding to the DTD. The system was built on top of the object-oriented DBMS *ObjectStore*.

The news-on-demand application was chosen for the prototype because it is a good representative of a distributed multimedia system. News documents, which include television news and newspapers, incorporate non-continuous media (such as text and images) and continuous media (such as video, audio and synchronized text). These documents are inserted by news providers (or information sources) into the distributed multimedia database. The documents can then be retrieved from the database by subscribers or end users using the query facility.

In designing the type system, four important issues were taken into consideration: (1) modeling of the individual document media components (such as text, image, audio, video); (2) modeling of the document structure (i.e. the hierarchical and the sequential relationship between the logical elements of the document); (3) modeling of the presentation information for the document using style sheets that are stored in the database; (4) representing meta information about the multimedia objects such as the temporal and spatial relationships between the different continuous media objects.

In the resulting news-on-demand application, the types that are necessary to model the news articles are "hard-coded" in a static type system that supports only news-on-demand documents. The articles, which comply to the SGML/HyTime standards, are inserted into the database by instantiating appropriate objects of theses types to represent them. Each article is treated as a base object with layers of annotations. This approach allows the article's content to be stored as a continuous text block which improves the search and display efficiency. Once articles are inserted in the database, they cannot be deleted or updated; thus the database implements a read-only model. In a future release, the news providers may insert newer versions of the news articles. The database management will handle the version management issues.

The reader is referred to [Vit95] and [EM95] for more details about the news-on-demand application.

3.3 Support for Multimedia Applications

The research, being conducted this year, aims at generalizing the multimedia type system, developed for the news-on-demand application, to support a wide range of applications. To achieve this goal, two major projects have been implemented. The first is the design of a generic multimedia type system that is capable of supporting multiple DTDs [Sch96]. This is achieved by defining a generic type system (i.e. a number of built-in types) to model the characteristics common to multimedia documents. Whenever a new DTD is added to the system, new types are dynamically added to the type system to support the new class of documents. This method provides a dynamic extensible type system in contrast to the static news-on-demand type system. A description of the generic type system and the dynamic creation of types is provided in Chapter 4.

The second project, which is the contribution of this thesis, provides an automatic method of inserting SGML documents of arbitrary types into the database. In the news-on-demand application, the news articles were assumed to conform to the news DTD so documents were not validated to detect errors in their structure. This places the responsibility on document authors to ensure that their documents are error-free before attempting to insert them into the database. With multiple DTDs, the need for a full-fledged parser to validate the documents gains more importance. It releases the document authors from the burden mentioned earlier and ensures the consistency of the database. Also, in the news-on-demand application, the instantiation routines were hard-coded to instantiate objects that represented the news-on-demand types. With the dynamic support of new classes of documents, the instantiation routines cannot be hard-coded. The types of the objects to be instantiated has to be dynamically detected after parsing the document instances. My research addresses these two issues. To validate document instances, the multimedia system is coupled with an SGML parser. The output of the parser is retrofitted to create a parse tree to represent the document instance. Automating the instantiation phase is done in two stages. First, when adding support to a new document class in the database, *meta types* are dynamically added to the database to store meta information about the DTD types. Second, whenever a new document is inserted in the database, the Instance Generator[1] queries the database for the DTD's meta types. These meta types together with the parse tree are used to instantiate the appropriate objects in the database to represent the document instance. A more detailed description of this research is presented in Chapters 5, 6 and 7.

3.4 System Architecture

The multimedia system follows a distributed multiple server/multiple client architecture. The clients are connected to a set of servers over a broadband ATM network. The servers are responsible for the storage and management of multimedia documents. There are two types of servers: *the continuous media servers (CMS)* and *the non-continuous media servers (NCMS)*. The CMS are file servers that store continuous media such as audio and video. The NCMS store non-continuous media such as text and still images. The current architecture integrates only the NCMS with the database (i.e. only non-continuous

[1] The Instance Generator is one of the components of the document entry system. It is discussed in Chapter 7.

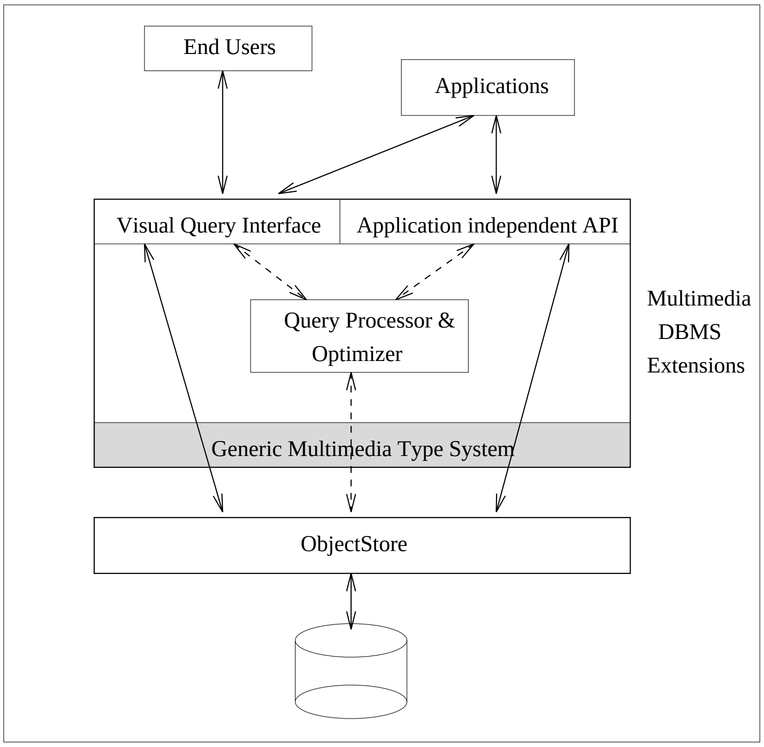

Figure 3.1: Conceptual Multimedia DBMS Architecture

media is stored in the database). As for the continuous media, the database only stores meta information, namely quality-of-service parameters, synchronization requirements, and information for locating the media files on the CMS.

The multimedia DBMS architecture is depicted in Figure 3.1. We use ObjectStore as the underlying DBMS for the multimedia system. Since ObjectStore does not provide native support for multimedia types other than text, a multimedia layer is built on top of ObjectStore. This multimedia layer defines types to represent the media objects (*atomic types*), the DTD logical components (*element types*) and meta information for the DTD components (*meta types*).

Users and applications can access the multimedia database via a visual query interface or an application independent API. Currently, the visual query interface is implemented based on the news-on-demand application only and has yet to be generalized. It interacts with the ObjectStore query processor using the multimedia type system. Future work includes the development of a query processor and optimizer that support content-based queries of images and videos. Once the query processor and optimizer are integrated to the system, the query interface will interact with it rather than with ObjectStore. The new interaction is shown with a dashed line.

In the long run, as discussed in Chapter 9, TIGUKAT [OPS+95], an extensible object-oriented database, currently under development at the Laboratory for Database System Research at the University of Alberta, will replace ObjectStore. TIGUKAT, as opposed to ObjectStore, has inherent support for multimedia objects.

The testbed environment for the multimedia system consists of IBM RS/6000 workstations acting as client and server machines and interconnected via a broadband ATM network.

Chapter 4

The Multimedia Type System

As mentioned earlier, the focus of this research is the automatic insertion of documents in the multimedia database. This is achieved by instantiating the appropriate objects in the type system that corresponds to the document's structure. That is why the design of a type system that enables such automatic insertion is a crucial step towards achieving this goal.

To automatically insert documents of arbitrary types in the database, the database should be queried dynamically to retrieve information about the different types. This is achieved by modeling DTDs as objects. Every DTD object stores meta information about its components such as the element names, their attributes and their content models. At run-time, the DTD object can be queried to determine which types should be instantiated to represent the different document components. Such meta information is modeled in the *meta type system*.

The design of the meta type system is one of the major contributions of this thesis. It is implemented as a kernel type system that can be extended to support new DTDs as need arises. The kernel type system consists of built-in types that define a number of meta types to model the characteristics common to multimedia document elements. Whenever a new DTD is added to the system, new types are dynamically added to the meta type system to support the new class of documents[1]. These new types are derived from the appropriate built-in meta types; thus forming a hierarchy of types in a structured type system. Section 4.1 discusses the reasons for adapting these techniques in designing the meta type system and the alternative techniques that were considered.

4.1 The Design of the Generic Meta Type System

In designing a type system, two fundamental questions need to be addressed:

(1) What data model should be used?

(2) Should a flat or a structured type system be supported?

The rest of this section discusses these two issues.

4.1.1 A Universal Data Model vs. a Kernel Data Model

To design a database system capable of supporting multiple DTDs, two possible data models can be used. The advantages and disadvantages of both models are listed below:

- The use of a universal data model that is capable of supporting all document types (DTDs). This means that one type system should be used to represent different meta types conforming to different DTDs.

  Advantages: Using a universal data model, the database schema will be static since there is no need for extending the meta type system

[1]The process of extending the meta type system is described in more detail in Section 4.5.10.

to support new DTDs. This simplifies the meta type system implementation and relieves the system from costly schema evolution techniques.

Disadvantages: To design a universal data model to support all DTDs is very difficult, if not impossible. Even if such a data model exists, it will be too general and inefficient to serve multimedia application purposes. Also, it will be hard to model the document structure in a way to allow type checking and querying built on document components. This will be a major drawback in the type system.

- The use of a kernel data model that can be extended to support new DTDs as need arises.

  Advantages: The structure of the different DTDs will be directly mapped to the database schema which facilitates type checking and querying.

  Disadvantages: Dynamically adding types to the type system involves dynamic schema evolution and instance conversion which can be very costly to applications linking to the database schema. Depending on which schema evolution strategy is employed, certain overheads will be imposed on the applications[2]. Overheads range from the necessity to relink all applications using the database (even if they are not using the new schema) to slower access time for object instances (if versioning techniques are employed).

The second approach was adopted in designing the meta type system. To minimize the dynamic schema evolution overhead, only leaf-types are dynamically added to the database schema; thus avoiding the need for instance conversion. This technique is adopted because ObjectStore allows the addition of leaf types to the database schema without the need to call the schema evolution

[2] A discussion of the different schema evolution strategies is out of the scope of this thesis. The reader is referred to [Sch96] and [PO95] for such discussion.

facility. These types are added automatically to the database schema whenever an application using these types opens the database. Other applications will not be affected by the database schema unless they need to use the new types; in this case such applications have to be relinked to the new schema. This approach increases the accessibility of the database and isolates applications using the database from the schema modifications as much as possible.

4.1.2 A Flat Type System vs. a Structured Type System

The second issue that needs to be addressed is the use of a flat versus a structured type system. Using a flat type system involves adding the new generated types as root types in the database (no supertypes). Even though this technique is easy and straightforward to implement, a structured type system was adopted in our system. This is because the flat type system does not make use of inheritance to reduce code redundancy and express common behavior. For example, if two elements in two different DTDs, say image1 and image2, represent a media object (image). Using a flat type system, the data members and code needed to instantiate the media object will be duplicated in the two generated meta types. If both types were subtyped from a common element, say ImageType, that abstracts the common code and data members, redundancy will be highly reduced. Also, the ability to express commonality between types allows for a more powerful query model. For example, the database can be queried for the meta information of all types that represent an image media object. To maintain a structured type system, information is gathered about the DTD elements to enable the system to automatically detect the supertypes from which each element should be subtyped.

Abstraction and reusability of types are not fully exploited in the current type system. As mentioned earlier, to minimize the dynamic schema evolution overhead, the new types added to the system are always created as leaf nodes in the hierarchy of types. This excludes the possibility of extracting features that are common to different types within one DTD or across DTDs and abstracting them into supertypes. Thus our system restricts reusability of types to the built-in types that are modeled in the kernel type system. A main disadvantage of this approach is that the resulting system suffers from some code redundancy. Future research aims at addressing the reusability and abstraction issues for types that are dynamically added to the database. Addressing this problem involves not only addressing dynamic schema evolution issues but also semantic heterogeneity issues (see Chapter 9).

4.2 The Design of the Multimedia Type System

Designing a type system for an object-oriented database is not easy since the type system will eventually be mapped to the database schema. In our system, it is even a more challenging task because our data model should be sufficiently general to support the different multimedia applications; i.e. to handle multiple DTDs.

The modeling of the meta information discussed earlier is only one component of the multimedia type system. Modeling of other information is crucial to the modeling of the multimedia documents:

- **Modeling of the different media objects:** A multimedia document consists not only of text but of other media components such images, audios and videos. Since ObjectStore does not provide native support for multimedia objects, a multimedia layer is built on top of ObjectStore. This layer models the multimedia objects together with their quality of service parameters in the so-called *atomic type system*. The atomic type system is described in Section 4.3.

- **Modeling of the document structure:** Since SGML documents are structured, the type system should reflect the structure of these documents and the hierarchical/sequential relationship between their components. In our system, the document structure is modeled in the *element type system* (see Section 4.4) where every DTD element is represented using a type in the type system.

The resulting multimedia type system consists of three components: the atomic type system, the element type system and the meta type system. The design of the multimedia type system conforms to the design of the meta type system. It is implemented as a kernel type system that can be extended to support new DTDs. The kernel type system consists of a number of built-in types that constitute the core of the multimedia database schema. These built-in types define the atomic types and a number of element and meta types that model the characteristics common to multimedia documents.

Even though the design and implementation of the atomic and the element type systems are part of a separate thesis [Sch96], they are discussed in the next two sections. The discussion is included for completeness and due to their impact on the meta type system.

4.3 The Atomic Type System

The atomic type system is used to model media objects (such as text, image, video and audio) that are the basic components of multimedia documents. In our system, these media objects are referred to as *primitive objects*. The primitive objects refer to the raw media representation together with synchronization and quality-of-service information.

The quality-of-service parameters are needed for presentational purposes and can be negotiated at the document's retrieval time according to the hardware avaliable on the client machine and the desired quality and cost required by the client. For example, the quality-of- service parameters for a video can include the duration of the video (number of minutes), the video color (black-and-white or colored), the width and height of the video, the video format (MPEG1, MPEG2 or JPEG), the frame rate, etc.

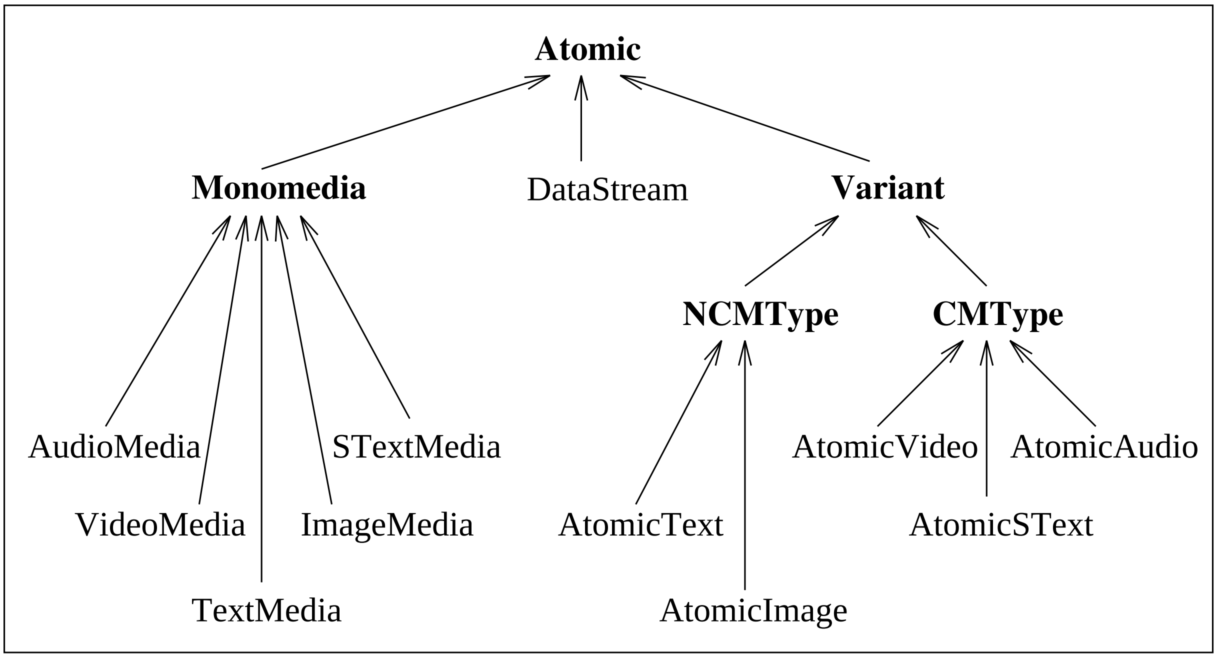

Figure 4.1: Atomic Type System

Primitive objects are allowed to have multiple variants that differ only in their quality-of-service parameters. For example, a video can have two variants: one being a colored version of the video and the second being a black-and-white version, or one variant being in a different format than the other. Also, for continuous media types such as audio and video, variants can differ in the number of data streams that are associated with each variant. Normally, the higher the number of data streams, the higher the quality-of-service and the higher the presentation quality.

Figure 4.1 shows the type hierarchy for the atomic type system[3]. At the root of the atomic type system, the type Atomic is defined as an abstract supertype of all atomic types. The three derived types of Atomic represent the three types of information that are modeled in the atomic type system.

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

4.3.1 DataStream

DataStream is an atomic type for streams. Streams are only used to represent files for the continuous media objects (audio, video and synchronized text)[4]. These files, in contrast to the non-continuous media (text and images) files, are not stored in the database but on a continuous media file server (CMFS). The database stores meta information about these files that is necessary for their retrieval from the CMFS. The type DataStream is used to capture some of that meta information such as the file `size` on the CMFS and the universal object identifier (`uoi`) which is a unique identifier used to locate the file on the server.

4.3.2 Variant Types

Variant is defined in the atomic type system to represent the variants that can be associated with a media object. It holds the raw media representation, the QoS information and some other information needed by the synchronization component. There are two types of variants according to the type of media they model: non-continuous media is represented by the type NCMType and continuous media is represented by the type CMType. The distinction between the continuous and non-continuous media is made since they are handled differently in the system. Non-continuous media is stored in the database as native objects whereas continuous media objects are stored on a CMFS with meta information stored in the database[5]. Since instances of NCMType store the raw media in their objects, it defines a data member called `content` which is an array of bytes. Instances of CMType store meta information such as the server on which the continuous media is stored (`site`) and references to the data streams that composes the media object (`DataStream *`). Note that, storing the server

[4]Note that, one particular audio or video can consist of a number of streams. The combination of several streams generates a quality-of-service level.

[5]This is a hybrid approach between non-integrated and integrated database systems. Non-integrated databases stores only meta information about multimedia objects in the database and the actual objects on files whereas integrated database systems store the multimedia objects together with their meta information in the database.

name in the variant object assumes that all the data streams associated with a variant object will be stored on the same server. If such assumption is invalid, the server name should be stored with each stream (in the DataStream type).

The five variant types are modeled using the five atomic types shown in figure 4.1. Each of these types are used to store the QoS information specific to that type of media. For example, the type AtomicImage stores a pointer to an image QoS instance (`imageQoS *`). Instead of modeling the QoS information directly in the atomic type system, a different type hierarchy is maintained (TQoSVariant type system). These types are introduced to achieve a clear distinction between information stored about the contents of an atomic object and the presentation information (QoS information). The discussion of the TQoSVariant type system is not presented in this thesis. It can be found in [Sch96].

4.3.3 Monomedia Types

The concept of Monomedia types are introduced to allow a primitive object to have multiple variants. Since each Monomedia instance can be associated with QoS information about the object irrespective of its variants, the attribute `monomediaQoS` is introduced to model this information in the TQoSVariant type system.

Monomedia is subtyped into five derived types that correspond to the five variant types. Instances of these monomedia types store references to all of their variants. For example, an instance of type ImageMedia will store references to all of its variants, which are of type AtomicImage.

The link between the Monomedia types, the Variant types and the DataStream is maintained through pointers. For example, an instance of type VideoMedia will have pointers to all its associated variants which are instances of type AtomicVideo. Each of the referenced instances will, in turn, have pointers to all its associated data streams, which are instances of type DataStream. In composing an SGML multimedia document instance, these references are represented by using SGML ID/IDREF referencing capability (discussed in Chapter 2). It then becomes the responsibility of the Instance Generator to resolve these references after instantiating the appropriate objects in the database (Chapter 7 describes this process).

4.4 The Element Type System

The element type system is used to model the DTD structure. It is a uniform representation of elements in the DTD and their hierarchical relationships.

As mentioned earlier, in designing our kernel type system, a number of built-in element types were defined to model the characteristics common to all (or some) DTD elements. These built-in types constitute the core of the element type system. Whenever support for a new DTD is required, new types are dynamically created to represent the logical elements in the DTD. According to the characteristics of the DTD elements, the corresponding types that represent them are subtyped from the predefined element types.

In general, any DTD element that needs to be supported by our type system will fall under one of three categories.

4.4.1 Structure Elements

Structure elements are elements in the DTD that are defined to represent the structure of the document instances, such as an element title in a thesis or an article DTD. To represent these types in the element type system, a number of built-in types have been introduced:

Element

This is the supertype of all element types. Element stores some basic information that should be modeled for every element in the DTD. A DTD element type will be directly subtyped from Element only if the element is defined to have an EMPTY content model. Figure 4.2 shows the types derived from Element.

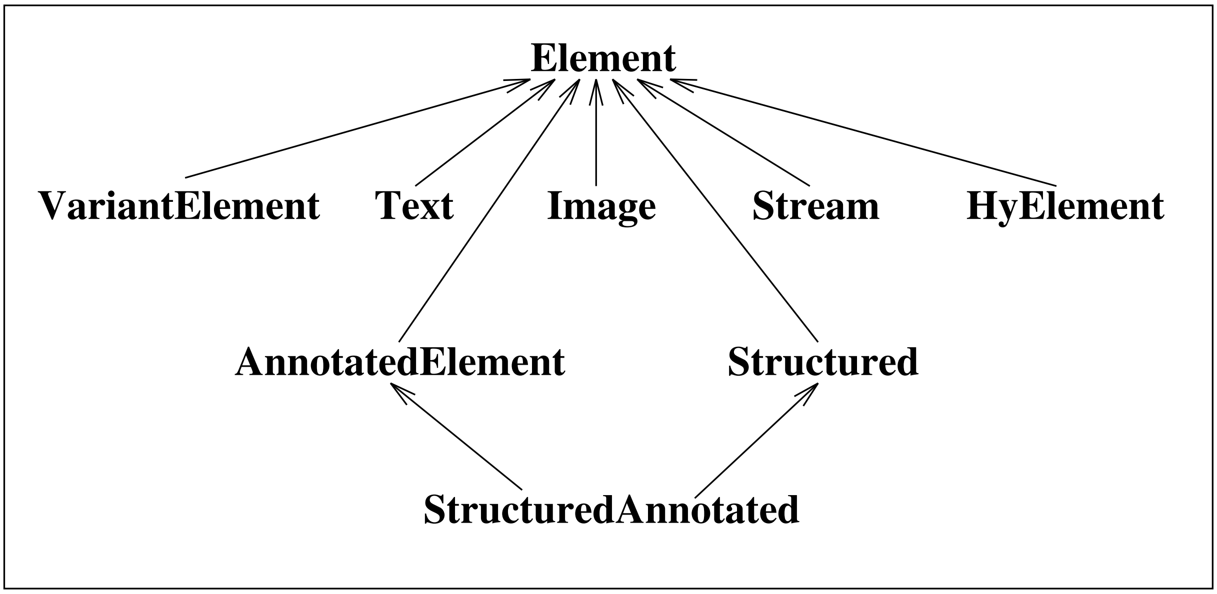

Figure 4.2: Top Level Hierarchy of the Element Type System

Structured

Structured is the supertype of all DTD elements that have a complex content model; i.e. non-leaf elements. Since non-leaf elements always have child elements, the type Structured maintains a `childList` which stores references to all the element's children in the order in which they appeared in the instance.

AnnotatedElement

For efficiency, the textual content of a document instance is stored as a one continuous string with layers of annotations. Every textual element is associated with an *annotation* which stores the start and end position of the element with respect to the document's text string. Annotations are needed for all elements that need to be located within the text string. For example, annotations can be used to store the position of a certain image within the document for display purposes. The type AnnotatedElement is the supertype of all types with annotations. An element is said to be annotated if the element or any of its subtypes have a #PCDATA in their content model or if the element needs to be located within the text block (e.g. images and links).

StructuredAnnotated

StructuredAnnotated is the supertype of all elements which are structured and have annotations. The type StructuredAnnotated is derived from both supertypes: Structured and AnnotatedElement. It inherits the `childList` attribute from Structured and the `absoluteAnnotation` attribute from AnnotatedElement.

4.4.2 HyTime Elements

HyElement is the supertype of all elements in the type system that conform to a HyTime architectural form; i.e. the elements that have a `HyTime` attribute. Its immediate subtypes represent the architectural forms that are currently supported by our system. These are: HyDoc_AF, Dimspec_AF, Axis_AF, Event_AF, Ilink_AF, Fcs_AF, Extlist_AF, and Evsched_AF. Our type system can be extended to support other HyTime architectural forms by defining built-in types to represent these AFs and deriving them from the HyTime type.

Following the HyTime standard, all the HyTime types representing HyTime AFs define an `id` attribute (inherited from HyTime) to uniquely identify elements for referencing. Furthermore, they contain data members for all attributes that are required by the HyTime standard to support the architectural form. The list of attributes defined to support the different architectural forms is provided in [Sch96].

Note that, types Video, Audio and SText are derived from the HyTime type Event_AF since they use HyTime events to represent their synchronization information.

4.4.3 MM Elements

The **MM** types have been introduced in our type system to provide a consistent way of handling objects that refer to monomedia objects. Since monomedia types are modeled in the atomic type system, it follows that MM types refer to atomic types.

In our system, every concrete type in the atomic type system is modeled in the element type system using an MM type. This maintains a one-to-one correspondence between the concrete atomic types and the MM types. Whenever an element in the DTD refers to a multimedia object, it is subtyped from one of the MM types. This indirection is maintained so that no new types will be directly derived from an atomic type.

There are MM types defined in our type system to correspond to the three types of information represented in the atomic type system:

- The MM type for DataStream is Stream which contains an attribute `datastream` that represents a pointer to type DataStream.
- The MM types for media atomic types (TextMedia, ImageMedia, AudioMedia, VideoMedia and STextMedia) are Text, Image, Audio, Video and SText. Each one of these MM types contains a reference of the corresponding media type. For example, Image contains an attribute `media` which points to an ImageMedia object.
- MM types for variant atomic types (AtomicText, AtomicImage, AtomicAudio, AtomicVideo and AtomicSText) are TextVariant, ImageVariant, AudioVariant, VideoVariant, and STextVariant. These MM types are derived from the type VariantElement.

Figure 4.3 shows the relationship between the atomic types and the MM element types for Video types. In the figure, the following convention is used: user-defined types appear with a 'My' prefix followed by the element type name from which they are derived. Instances of types are appended by 'a' or 'an' followed by their type name.

4.4.4 Helper Types

There are a number of types in the generic type system that are not subtyped from Element but logically belong to the element type system.

Annotation Type

The Annotation Type is used to store the start and end position of an element in the textual string of the document. Any annotated element (refer to Section 4.4.1) has an instance of type Annotation associated with it.

DocumentRoot Type

The DocumentRoot type is used to store the document's text string and a list of all the Monomedia objects (TextMedia, ImageMedia, AudioMedia, etc). This list is stored in the DocumentRoot for efficiency purposes since atomic objects are required quite frequently by other components of the system.

The DocumentRoot also stores the document's lists of annotations with its text string. This list can be used for, among other purposes, displaying a document since presentation information might depend on the document's structure (for example, all titles are displayed in boldface).

Since annotations are specific to each DTD's elements, the annotation lists can not be stored in the built-in DocumentRoot type. Instead for every DTD, a new type is created as a subtype of DocumentRoot. The new type will contain data members to represent the document's annotations. For example, an article DTD will be represented by a new type ArticleRoot.

DTD Type

To facilitate the process of document entry, the DTD type has been introduced. For every DTD in the type system, a DTD object is created which stores the

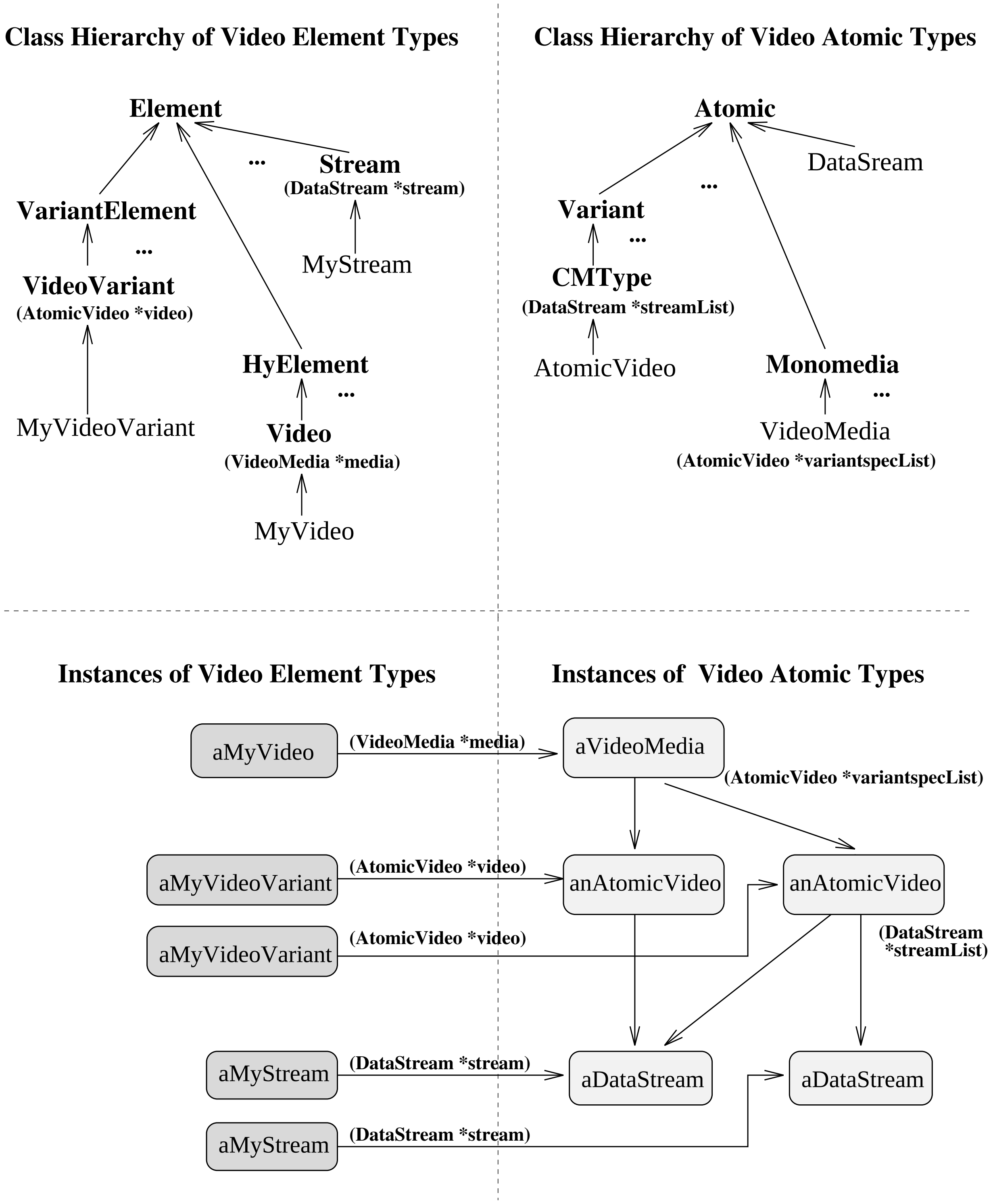

Figure 4.3: Example for `MM` Video Types

DTD name, the DTD string and a types list. The types list contains a meta type object for each element defined in that particular DTD. At document instantiation time, these types are used to automatically instantiate objects to represent the document instance.

4.5 The Meta Type System

The meta type system is used to model meta information about DTD elements that is necessary for the automatic document instantiation. For every logical element defined in the DTD, there is a corresponding type in the meta type system as well as in the element type system. This provides a one-to-one correspondence between the concrete element types and the concrete meta types in the type system. The design of the meta type system is one of the major contribution of this research.

In our type system, meta types are used to perform two important tasks:

- They store meta information about the logical elements in the DTD such as the elements names, their attributes, and the supertypes from which these elements are derived in the element type system. Such information is necessary for instantiating the appropriate database objects and setting their attributes.

- They define functions to instantiate database objects to represent the different components of the document instance. These functions are referred to as *instantiation routines*[6].

A number of built-in meta types have been defined in our kernel type system to represent the characteristics common to all (or some) element meta types.

[6] The reader should make the distinction between the instantiation routines and the Instance Generator. The term *instantiation routines* is used to refer to the dynamically generated code for object instantiation. This code is generated by the Type Generator once for each DTD. The Instance Generator is the system component that is invoked after the SGML Parser for every document instance. It is described in detail in Chapter 7.

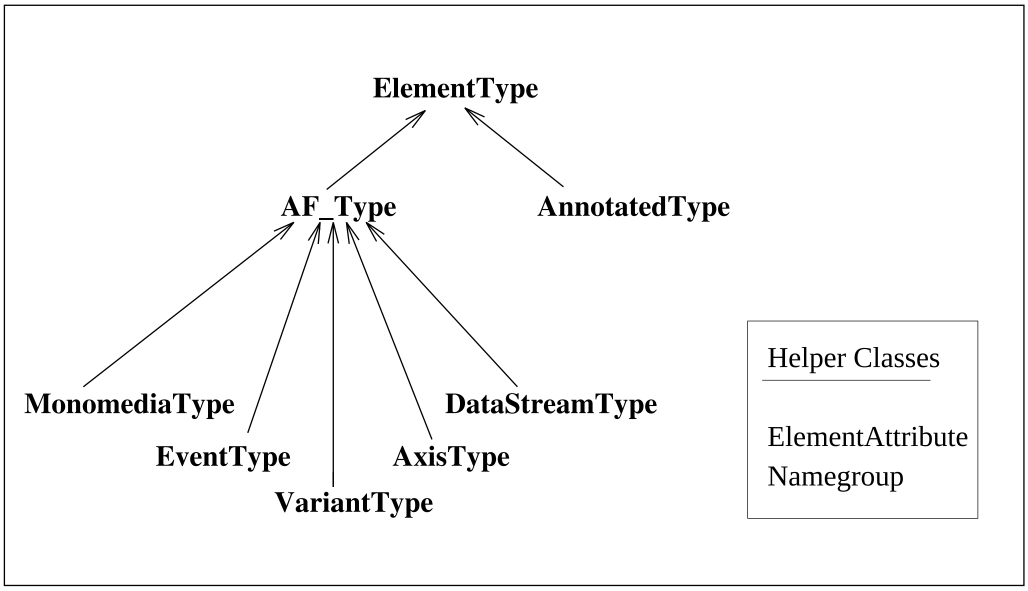

Figure 4.4: Built-in Meta Types

Whenever a new DTD is added to the system, the generated meta types to represent the DTD will be derived from one of these predefined types.

Figure 4.4 shows the top level type hierarchy for the meta type system. The rest of this section discusses these built-in types, their member functions and data members.

4.5.1 The ElementType

The ElementType type is the supertype for all meta types. It is used to store the meta information necessary to represent the elements in the DTD. It contains the data members: `name`, `attrList`, `supertype` and `isStructured`. The `name` data member stores the element name as it appears in the DTD[7]. A list of all the attributes defined in the DTD for the specific element is stored in the `attrList` data member. `Supertype` is an enumeration that represents the built-in element type from which the element is derived. The `isStructured`

[7] Since SGML element names are case insensitive, the element names are stored in uppercase letters.

data member is used to denote whether the element type has a complex data model or not (i.e: is derived from Structured or not). This information is important for the Instance Generator to decide whether to call `CreateObject` or `CreateStrObject` to create the element type. The member functions can be divided into the following categories:

- **Functions used to instantiate database objects**

  These functions, namely `CreateObject` and `CreateStrObject`, are virtual functions that are redefined in the generated meta types for the DTD elements[8]. `CreateObject` is redefined for the leaf elements (i.e: elements with simple content model) whereas `CreateStrObject` is redefined for elements with a complex content model.

- **Functions for setting attributes**

  The member function `SetAttributes` is a general function that loops over the attribute list and calls `SetSpecificAttribute` for all attributes that do not need special handling. The `SetSpecificAttribute` function is a virtual function defined in type Element and redefined in the generated element types.

  The member functions `CreateAString`, `FillInList` and `Get_TLanguage_Value` are also used to set attributes. `CreateAString` is used to create a persistent string in the database and set its contents. `GetStrValue` searches the attribute list to find the value of specific attribute given its name. `FillIntList` is used to fill an ObjectStore list of type integer given a string representation of the values. `Get_TLanguage_Value` takes a string and returns its corresponding TLanguage enumeration value if any[9].

[8]Note that, the document entry system relies on virtual functions for instantiating the appropriate database objects. This concept will be addressed in Section 4.5.10.

[9]The function `Get_TLanguage_Value` should be placed in LanguageType. It was moved to ElementType because the root element should update the language of the document's text block.

- **Functions for determining special attributes**

  Special attributes should be set during object creation and skipped during `SetAttributes`[10]. These functions include: `IsInHandledList`, `IsAttributeHandled` and `HandleID`.

Type ElementType has two abstract derived types: AnnotatedType and AF_Type.

4.5.2 AnnotatedType

The Type AnnotatedType is used to represent the meta type for elements that have annotations. These elements are derived from the Type AnnotatedElement. It defines the function `Create_Annotation` which is used to create a persistent Annotation object. The meta types for all annotated DTD elements will be derived from this type.

4.5.3 AF_Type

The AF_Type type is used to represent the meta types for elements that conform to a **HyTime** or an **MM** architectural form. Some of the attributes defined on these elements need special handling (special attributes).

As mentioned earlier, special attributes should be set during object creation and skipped during SetAttributes. These attributes are special either because they are represented on the atomic rather than the element side (e.g., attribute `price` for an Image MM type) or they need special handling such as type conversion. The latter are usually HyTime attributes.

AF_Type defines two member functions:`GetStrAttrValue` and `GetIntAttrValue`. `GetStrAttrValue` searches the attribute list to find the value of a specific attribute given its name and creates a persistent string that corresponds to that value. `GetIntAttrValue` gets the value of an attribute given its name and returns an integer that corresponds to that value.

[10] A discussion of special attributes is provided in Section 4.5.3

AF_Type has five abstract derived types: MonomediaType, VaraintType, EventType, AxisType and DataStreamType.

4.5.4 MonomediaType

The type MonomediaType is used to model the MM elements that represent media atomic types (elements derived from Text, Image, SText, Audio and Video).

MonomediaType defines the following member functions: `GetDoubleAttrValue`, `Get_id` and `Get_price`. `GetDoubleAttrValue` returns a double number that corresponds to the value of a specific attribute given its name. `Get_id` and `Get_price` create a persistent string to represent the id and the price respectively for a monomedia object.

As shown in Figure 4.5, MonomediaType has five abstract derived types: TextType, ImageType, AudioType, VideoType and STextType. Each of these types defines a function that creates a Monomedia object that corresponds to its media type. For instance, TextType defines the function `Create_TextMedia` which creates a persistent TextMedia object and sets its attributes; ImageType defines the function `Create_ImageMedia` which creates a persistent ImageMedia and so on. Also, the function `IsAttributeHandled` is redefined in each of these types to reflect the handled attributes.

Since Text and Image have annotations in our type system, the types TextType and ImageType are derived from the type AnnotatedType. Similarly, AudioType, VideoType and STextType are derived from EventType since Audio, Video and SText are derived from Event_AF.

4.5.5 VariantType

VariantType models the meta types of MM elements that represent variant atomic types. These elements are derived from TextVariant, ImageVariant, STextVariant, AudioVariant or VideoVariant.

Since all atomic objects have the data members: id, format and size, the

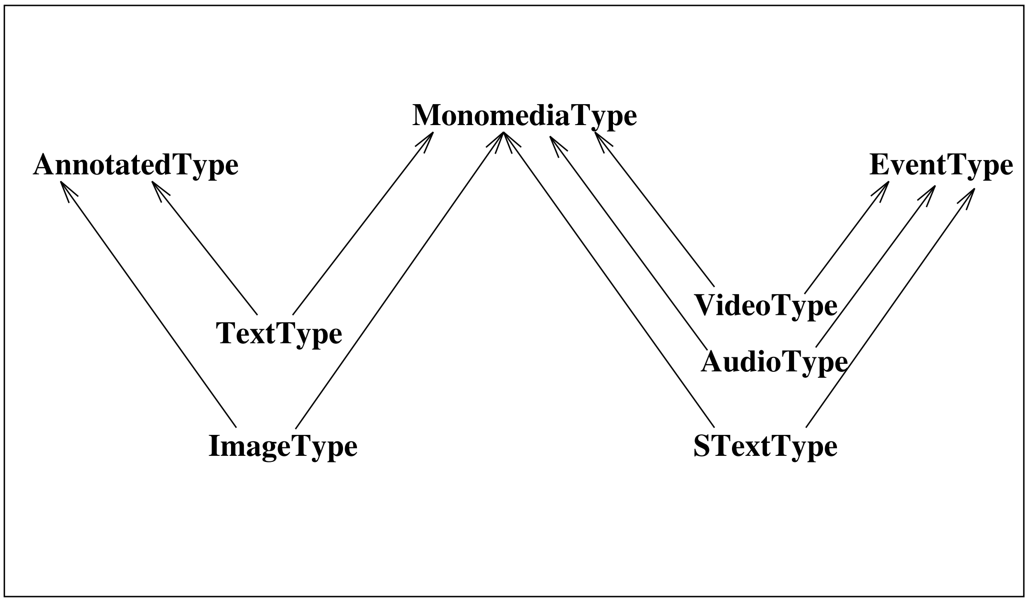

Figure 4.5: Type MonomediaType and its Derived Types

functions `Get_id`, `Get_format` and `Get_size` are defined in this type. It also defines: `GetUnsignedAttrValue` and `GetFormatAttrValue` which scan the attribute list for a certain attribute and return an `unsigned integer` or a `TFormat` enumeration value to correspond to the attribute's value.

VariantType is subtyped into ColorType and LanguageType. The two types represent the elements that have a `color` and `language` attribute respectively. ColorType defines the member functions: `Get_width`, `Get_height` and `Get_color`. On the other LanguageType defines the member functions: `Get_language` and `GetLanguageAttrValue`.

ColorType is further subtyped into ImageVariantType and VideoVariantType whereas LanguageType is subtyped into TextVariantType, AudioVariantType and STextVariantType. Each of the five types define a function that creates a persistent Variant object corresponding to its atomic type and redefines the function `IsAttributeHandled`. For instance, ImageVariantType defines the function Create_AtomicImage which creates an AtomicImage instance, TextVariantType

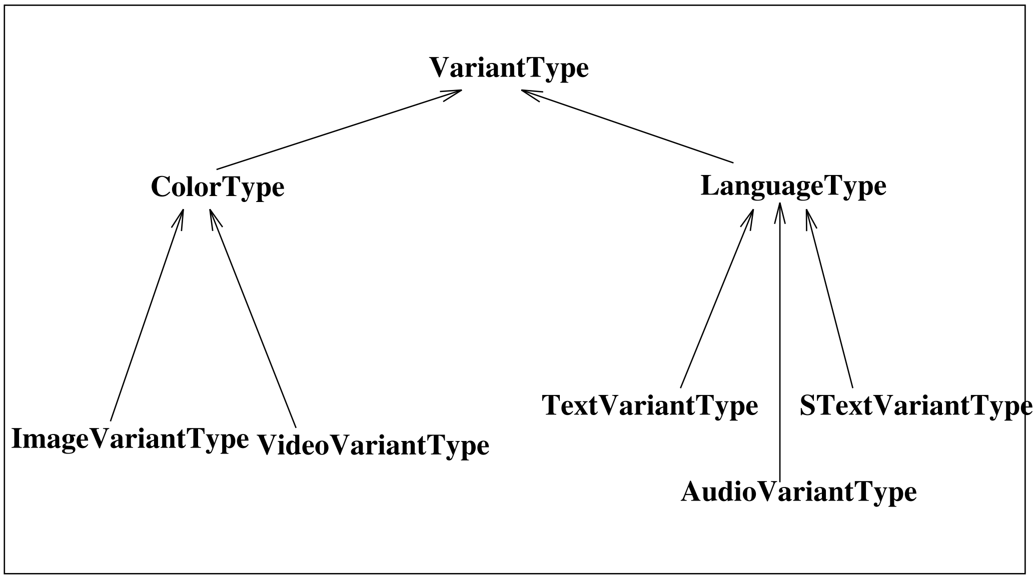

Figure 4.6: Type VariantType and its Derived Types

defines the function Create_AtomicText and so on.

4.5.6 EventType

EventType is used to represent elements in the DTD that are derived from Event_AF or Evsched_AF. These elements are characterized by having a `pls2gran` list[11].

EventType defines a function `Fill_pls2gran` which fills the `pls2gran` list with integer items that correspond to its string value.

As shown in figure 4.7, EventType is subtyped into: AudioType, VideoType, StextType and EvschedType. The first three types were discussed earlier.

EvschedType is used represent the meta type for elements that are derived from Evsched_AF. It contains the function `Fill_gran2hmu` which fills the `gran2hmu` list[12] with integer items that correspond to its string value.

[11]`pls2gran` is a HyTime required attribute for Event and Evsched AF. It stands for pulse to granule ratio.

[12]`gran2hmu` is a HyTime required attribute for Evsched AF. It stands for base granule to HyTime measurement granule (HMU) ratio.

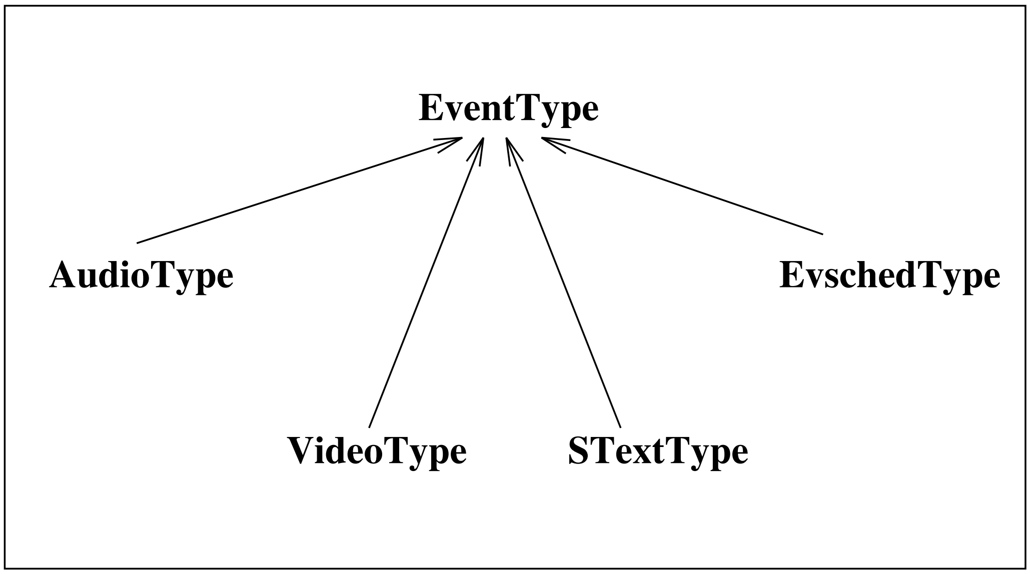

Figure 4.7: Type EventType and its Derived Types

4.5.7 AxisType

AxisType is used to represent elements in the DTD that are derived from Axis_AF. These elements are characterized by having an `axismdu` list and an `axisdim`. It defines functions `Fill_axismdu` and `Get_axisdim` which set the values of `axismdu` and `axisdim` respectively[13].

4.5.8 DataStreamType

The type DataStreamType is used to represent DTD elements that are derived from the MM element type Stream. These elements are characterized by having a reference to an atomic DataStream type.

DataStreamType defines a function Create_DataStream to create an atomic object of type DataStream. It, also, defines functions `Fill_axismdu` and `Get_axisdim` which set the values of axismdu and axisdim respectively.

[13] `axismdu` and `axisdim` are HyTime required attributes for Axis AF. `axisdim` stands for the size of the axis in MDU (measurement domain units) and `axismdu` stands for axis measurement domain unit.

4.5.9 ElementAttribute Type

Even though ElementAttribute is not subtyped from ElementType, it logically belongs to the meta type system. It is used to store meta information about element attributes. It defines the following data members:

- `name` is the attribute name as specified in the DTD.
- `valtype` is the attribute type (e.g. `CDATA`, `ID`, `IDREF(s)`, `NAME(s)`, `NUMBER(s)`, etc).
- `deftype` is the attribute's default type (e.g. `FIXED`, `REQUIRED`, `CURRENT`, `IMPLIED`, etc).
- `defvalue` is the attribute's default value defined in the DTD if any.
- `curvalue` is the last value assigned to the attribute during parsing the current document instance. This data member is used only when `deftype` is set to `CURRENT`.
- `namegrp`: name group if exists

4.5.10 Extending the ElementType Type System

All the types discussed above are built-in types that constitute the generic type system. As mentioned earlier, these types represent the multimedia objects (atomic types) and the behaviors and characteristics common to DTD elements (element and meta element types).

Whenever support for a new DTD is needed, new types are generated in the type system to support the new class of documents. These types extend the type system to model the features specific to the new DTD.

To model a new DTD in the system, a number of types need to be generated.

- For each DTD, a subtype of DocumentRoot is generated to represent the specific DTD's root. The new type will define data members to represent the list of annotations for the specific DTD.

- An element type will be generated to represent the root element of the DTD. The new type will be a subtype of type Document and type StructuredAnnotated.

- For every element in the DTD, a new type will be generated and subtyped from one of the predefined element types[14]. If the DTD element has attributes, the new type will redefine the virtual function `SetSpecificAttributes` to set the attributes defined in the attribute list according to their types (whether they represent an integer, a string, a list of integers, etc). If the DTD element contains some attribute(s) of type IDREF(s), the virtual function `ResolveRef` is redefined to resolve the references. Furthermore, if the new type is derived from `Structured`, the virtual function `SetChild` is redefined to set the children data members. This link is important for traversing the document.

- For every element in the DTD, a new meta type will be generated and subtyped from one of the predefined meta element types. One of the virtual functions `CreateObject` or `CreateStrObject` will be redefined in the new meta type depending on whether the element is structured or not.

[14] Determining the supertypes for the generated types is out of the scope of this thesis (see [Sch96]).

Chapter 5

Handling Multiple DTDs

Since multimedia documents are diverse in nature and structure, it is almost impossible to design a single catch-all DTD that can represent all multimedia documents. Even if such a DTD was designed, it would be too general to serve the different application requirements. Therefore, multiple DTDs should be designed to represent the different document types.

A system that supports different multimedia applications has to be sufficiently general and extensible to handle the dynamic insertion of multiple DTDs in the database. This is achieved by implementing a system that is capable of analyzing DTDs and creating types to represent these DTDs in the database.

Figure 5.1 shows the system components that were developed for handling multiple DTDs. These components are: the *DTD Parser*, the *Type Generator* and the *DTD Manager*. Even though the implementation of this system is part of another thesis [Sch96], some of its components directly affect document instantiation and are contributions of this thesis. In this chapter, the system is presented as a whole. The reader is referred to Section 5.2.1 for a description of the contributions of this thesis.

The rest of this chapter provides a short discussion about these components. Only the issues that affect the automatic document entry system are discussed

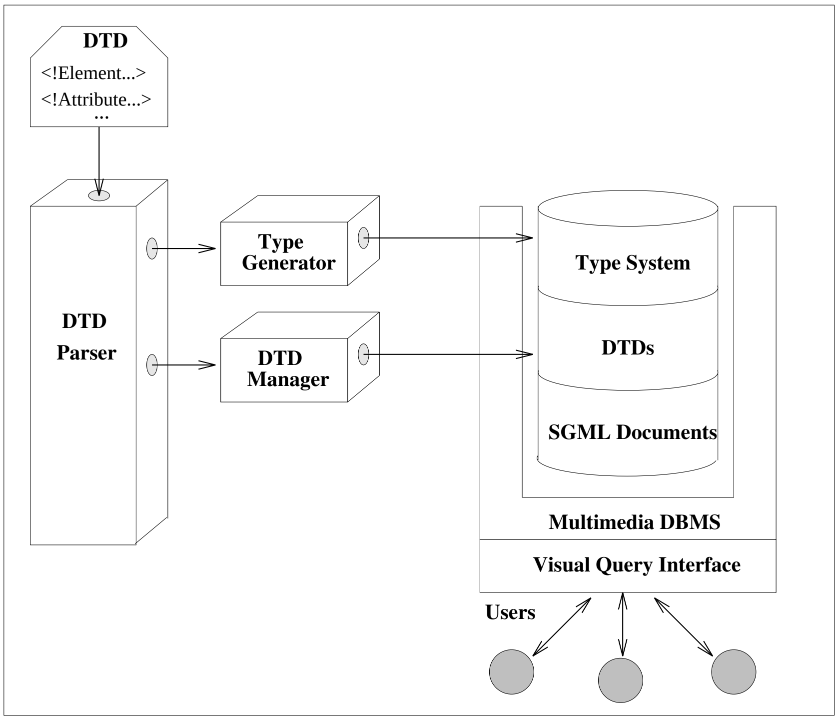

Figure 5.1: Architecture for Handling Multiple DTDs

in detail. A complete description of the system can be found in [Sch96].

5.1 The DTD-Parser

The DTD Parser component is used to check a new DTD for correctness. During parsing, the parser collects information about the SGML elements defined in the DTD to be used for the automatic generation of types to represent the DTD. After a successful validation of the DTD, the parser calls the Type Generator

A publicly avaliable DTD parser called *dpp*, developed at the University of Illinois, was used for validating DTDs. The parser is written in C and uses flex and bison (lex/yacc replacement). *dpp* checks the correctness of the DTD by checking its conformance to a meta DTD. This meta DTD describes a grammar for defining DTDs.

Some modifications were made to the original version of *dpp* to meet our system requirements. While validating the DTD, *dpp* does not build data structures to represent the DTD elements. These data structures are not necessary to validate the DTD, which is its main function. However, in our system such data structures are necessary to invoke the Type Generator and they have to be maintained.

dpp has been modified to maintain a list of the elements defined in the DTD. While parsing the DTD, whenever an element definition is encountered, an entry in the element list is created. Each entry has the form:

```
struct Element {
       char *name;
       struct ContentDecl *content;
       struct Attribute *prt_attr_list;
       struct Element *next_elem;
       struct Group *incl;
       int reachable;
       int visited;
       struct Group *mult_elem;
       int isList;
       int isStructured;
       enum SType supertype;
};
```

where `name` is the element name, `content` is a pointer to the content model of the element, `prt_attr_list` is a pointer to the element's attribute definitions, and `next_elem` is a pointer to the next element in the element list. This data member is important for traversing the elements list. `incl` is a pointer to the list of elements that can occur inside the current element through inclusion[1].

Whenever an attribute definition is found in the DTD, an entry is created in the attribute list of the element that contains the attribute. Each entry of the attribute list has the following structure:

```
struct Attribute {
       char *name;
```

[1] Inclusion is an SGML feature which is used as a shorthand for defining a content model. Elements named in an inclusion can occur anywhere in the element being defined or any of the subelements.

```
        char *valtype;
        enum DefType deftype;
        char *defvalue;
        struct NameGrp *namegrp;
        struct Attribute *next_attr;
        int predef;
};
```

where `name` is the attribute name, `valtype` is the attribute type (e.g. `CDATA`, `ID`, ...), `deftype` is the default type (e.g. `FIXED`, `REQUIRED`, ...), `defvalue` is the default value, `namegrp` is a pointer to the attribute name group, if defined, and `next_attr` is a pointer to the next attribute in the attribute list.

5.2 The Type Generator

After the successful validation of the DTD, the Type Generator is invoked with the element list data structure. For every element entry in the element list, the Type Generator generates C++ code to dynamically create the necessary C++/ObjectStore types to represent the element. After the successful completion of the code generation, the generated code is compiled and executed to update the database schema. Applications using the new DTD can link to the new schema and make use of the new types.

The Type Generator is the system component responsible for generating the types which model the specific characteristics of the DTD elements. This task is a critical step that affects the whole system. During document instantiation, instances of the generated types will be instantiated to represent the document instance. Hence, the type design affects the way the documents are stored in the database. It follows that the type design greatly affects the performance and capabilities of the system. A "good" design facilitates querying and document retrieval and increases the system's functionality.

To successfully model a new DTD in a type system, a very important question needs to be addressed: What information needs to be modeled in the new types and how can this information be represented in the most efficient way?

Modeling DTDs in a type system should involve modeling the following information:

- **Modeling of the Element Structure:** SGML documents are structured with the element root at the top of the tree and DTD elements as nodes in the hierarchy. Elements appearing in the content model of an element represent the children of this element's node in the composition hierarchy. Leaf nodes represent elements with simple content model (`PCDATA` or `EMPTY`). The document structure should be reflected in the type system. This enables document traversal starting from a specific tree node as well as querying built on document structures.

- **Modeling of the Element Attributes:** In SGML, elements can define attributes. Even though attributes are not part of the document hierarchy, they can be useful in identifying certain information about the DTD elements and they should be modeled in the type system. This enables queries built on attribute values to be supported by the system; for example, queries like: "give me all books whose `language` attribute is set to `English` or whose `subject` attribute is set to `databases`".

- **Modeling of the Element Behavior:** DTD elements have behaviors. For example, an instance of element myImage defined in a DTD should be able to retrieve the image it is storing or a structured element should know its children. Such behaviors should be modeled in the type system.

- **Modeling of the Element Type:** This involves storing meta information (such as element names, attributes) about DTD elements in the type system. Such information can be necessary to automatically instantiate documents in the database. It can also be used in querying (for example, "give me all elements in the system that have a `keyword` attribute" or "give me all the attribute names/types defined on the element book").

To model this information in our type system, the Type Generator generates two types for every logical element defined in the DTD. The first type represents the DTD element in the element type system. It is used to model the structure, behavior and attributes of the DTD elements. The second type, which represents the element in the meta type system, models the element type. The rest of this section discusses how the system models the different types of information.

5.2.1 Interface with the Document Entry System

The Type Generator is the system component reponsible for extending the type system to support new document types. As mentioned earlier, the extensions include generating new types to represent the DTD elements as well as the meta types that correspond to the elements. Even though the implementation of the Type Generator is part of [Sch96], some of these extensions directly affect document instantiation and are contributions of this thesis.

Since the Type Generator is one coherent system, it is hard and unnatural to separate the discussion based on the contributions of this thesis vs. those of [Sch96]. The system is, therefore, presented here as an integral entity.

The rest of this section highlights the contributions of this thesis to the Type Generator. The details of these contributions are presented in the discussion of the Type Generator provided in the rest of the chapter. They can be summarized as follows:

- The extension of the element type system to model some behaviors necessary for document instantiation. These behaviors are modeled as member functions in the newly generated element types. The member functions are `SetSpecificAttribute`, `SetChild` and `ResolveRef`. They are discussed in Section 5.2.4.

- The extension of the meta type system to model the new element meta types. To generate the C++ meta types, after the element types are generated, the Type Generator calls the modules for meta type generation. The element list described in Section 5.1 is passed to these modules. The discussion of the meta type generation is presented in Section 5.2.5.

- The generation of an application that initializes the types list of the `DTD` object to store the DTD meta objects. This is achieved by parsing the element list and generating code to instantiate a meta object for each entry in the list. These meta objects are persistent objects that are stored in the `DTD` object in the database. The code for fetching the `DTD` object from the database is also generated. The Type Generator compiles the generated application and invoke it after invoking the DTD Manager 5.3.

5.2.2 Modeling of the Element Structure

As mentioned earlier, elements appearing in the content model of an element are considered its child nodes in the composition hierarchy. These child elements (subelements) are modeled as data members in the element type.

A DTD can specify constraints in an element's content model (such as the type, the number and the order of the subelements). Modeling some of these constraints is difficult, since ObjectStore and C++ provide no support for expressing constraints. However, these constraints should be modeled in the type system. The rest of the section discusses how we model these constraints in the type system.

Modeling the Number of Subelements

An element's content model specifies the number of times an element can appear in it using *occurrence indicators* (refer to Chapter 2) and the possibility of inclusions.

If an element can appear more than once in the content model, the type system should provide enough storage space to keep information about all child elements; i.e. the parent type should store a list of elements instead of a pointer to a specific element. Similarly, if the element can appear more than once in the DTD, it should be maintained as a list in its annotation data member in the specific subclass of DocumentRoot.

A simple solution to this problem is to implement all subelement's data members as lists. This way the Type Generator does not have to keep track of the constraints. However, due to the high overhead of using lists in ObjectStore, this solution has not been adopted. Currently, the Type Generator automatically detects whether or not the element should be a list and generates the data members accordingly. It, also, sets the data member `isList` to indicate whether or not the element is implemented as a list in the DocumentRoot type annotations. This information is important for the instantiation routines since the method for setting a pointer to an object is different than that of setting an element in a list.

Modeling the Order of Subelements

In the DTD, the order of the elements is specified using connectors. Subelements can be ordered (if they are separated by ",") or unordered (if they are separated by "&"). To model this constraint, the abstract supertype Structured, from which all the structured elements are derived, contains a `childList` and the method `GetNth` which returns the *nth* element in the childList. Elements in the `childList` are assumed to be in order. It is the responsibility of the Instance Generator to instantiate the subelements in order[2].

Note that, the solutions described above allow our type system to model the constraints; however, they are not automatically enforced. This is not a major drawback of the system since the constraints are enforced by the SGML parser while parsing the document instances. The Instance Generator is not invoked unless the document is error-free.

[2]Using this technique, the type system does not provide a mechanism to express whether the elements are ordered or unordered.

5.2.3 Modeling of the Element Attributes

Attributes are implemented as data members in the element type unless they are inherited from one of the abstract supertypes or represented in the atomic type system.

The type of the attribute data member depends on the value type defined in the attribute. If the value type is `NUMBER/NUMBERS`, it is represented by an `integer` or a list of `integers` respectively. If it is `IDREF` or `IDREFS`, it is represented by a pointer to type Element or a list of pointers to Element respectively. All other attribute values are stored as character strings.

Note that, the instantiation routines responsible for setting the elements' attributes use the same technique to determine the type of the data member being set. This information is accessed from the `valtype` data member in the `Attribute` data structure.

5.2.4 Modeling of the Element Behavior

There are two possible ways of modeling element behaviors in our system. The first is through inheritance from the built-in classes that the element is derived from. For example, type Structured defines the member function `GetNth` described in the previous section. Since any structured element in the DTD will have Structured as one of its supertypes, it will automatically inherit the `GetNth` behavior.

In designing our generic type system, an object-oriented approach was adopted; as much functionality as possible was promoted to the abstract built-in supertypes. This allows for more abstraction and reusability in our system and decreases redundancy in the generated code. For every DTD element, the Type Generator should automatically detect from which abstract types it should be derived to inherit the appropriate behaviors. The automatic detection of supertypes is discussed in [Sch96].

Besides the inherited behaviors, each concrete type should model the behaviors that are specific to its particular DTD element. Such behaviors include functions for creating, initializing and deleting objects, access functions, etc. For simplicity, all data members for the concrete types are made public. This eliminates the necessity to define access functions for them; thus reducing the amount of generated code.

For every concrete type, a constructor and a destructor is generated. Constructors usually involve memory allocation and initialization for a new instance whereas destructors involve cleanup and deallocation of memory for that object.

Other functions need to be defined for the concrete types to facilitate object instantiation during document entry. The rest of this section discusses these functions in details.

SetChild

As discussed in Section 5.2.2, to model document structures in our type system, every structured element in the DTD stores references to all its children as well as a `childList` to keep track of their order. During document instantiation, these references must be updated to point to the child instances.

To achieve this goal, a function `SetChild` is generated for every concrete type that is derived from `Structured`. `SetChild` takes a string representation of the child name and a pointer to the child's instance. It updates the data member that stores the child reference as well as the `childList` with the instance pointer.

To generate the code for `SetChild`, the element's content model is traversed. For every element in the content model list, code is generated to set the appropriate data members.

At document instantiation time, whenever an object is created, the `SetChild` function for its parent element is called to set the parent's child elements. Since the initial traversal of the tree structure for valid SGML documents is always in a depth-first order (i.e. the parent element is always defined before its subelements), the object's parent element will always be instantiated before its subelements. Note that, using this technique, the `childList` will always have the children in the order in which they appeared in the document instance[3]; thus maintaining the elements' order constraint (discussed in Section 5.2.2).

SetSpecificAttribute

As discussed in Section 5.2.3, element types contain data members to model the values of their attributes. During document instantiation, these data members should be updated to store the actual attribute values.

To achieve this goal, a function `SetSpecificAttribute` is generated for every concrete type that has attributes. `SetSpecificAttribute` takes the attribute name and a string representation of the attribute value and updates the element's appropriate data member.

To generate the code for `SetSpecificAttribute`, the element's attribute list (stored in `prt_attr_list`) is traversed. For every attribute in the attribute list excluding special attributes (see Section 4.5.3) and IDREF attributes (see Section 5.2.4), code is generated to set the appropriate data members. Note that, since `SetSpecificAttribute` takes the attribute value as a string, data conversion might be required for attributes of other types (eg. integer attributes).

During document instantiation, after an object is created, the general-purpose function `SetAttributes` is called to set all the attribute values (see Section 4.5.1) which in turn calls `SetSpecificAttribute` to set a particular attribute value.

ResolveRef

Attributes of `valtype IDREF`, as discussed in Section 5.2.3, are modeled as pointers of type Element since they can reference any DTD element that has an `ID` attribute.

Even though IDREF attributes are modeled in the same fashion as all other attribute types, resolving IDREFS (updating the data members) during docu-

[3] This is because the Instance Generator traverses the instance's `parse tree` in sequential order.

ment instantiation time should be handled differently. The difference lies in the time the data members should be updated. In SGML documents, attributes are defined within the element's markup; therefore, the function `SetAttributes` is called immediately after instantiating the object. IDREF attributes are different. In SGML documents, it is valid for elements to reference other elements before the referenced element is defined. In this case, the instance with the IDREF attribute will be instantiated before the referenced element.

To solve this problem, resolving IDREFS is delayed until after all the document objects are instantiated. During object instantiation, a dictionary of all objects with ID/IDREF attributes is maintained (see Chapter 7 for a detailed discussion). This dictionary is then used to call `ResolveRef` for all the elements in the dictionary that need IDREF resolution. `ResolveRef` takes the IDREF attribute name and a pointer to the referenced instance.

To generate the code for `ResolveRef`, the element's attribute list is traversed. For every IDREF attribute in the list, code is generated to set the appropriate data members.

5.2.5 Modeling of the Element Type

To model meta information in the type system, the Type Generator generates types in the meta type system to represent the DTD elements.

For every entry in the element list, a meta type is generated as a subtype of one of the built-in meta types (described in Section 4.5). The choice of the supertype from which the meta type should be derived is determined by the `supertype` data member of the element list entry.

Besides the inherited behaviors from built-in types, the following functions are defined for each concrete type: a constructor for initializing new instances, a destructor for cleanup and a function `CreateObject` or `CreateStrObject`. `CreateStrObject` is generated for all the structured elements in the DTD; CreateObject is generated otherwise[4]. The parameters for `CreateObject` are: `node`

[4]For simplification, the two functions are referred to as `CreateObject` hereafter.

which represents an entry in the parse tree (see Section 6.2.2), `parent` which is pointer to the element's parent object in the document hierarchy and `table` which represents the dictionary used for resolving IDREFS in the document. `CreateObject` performs the following tasks:

- It creates an instance of the corresponding element type. For example, a meta type ArticleSectionType defines the function `CreateStrObject` that creates an instance of type ArticleSection. If the corresponding element type is annotated, it creates an `Annotation` object and updates the document root annotation list. Note that, if the corresponding element type is an **MM** type, the corresponding atomic object is created. To reduce code redundancy, functions for creating atomic objects are abstracted in the generic meta type system and inherited by the concrete types.

- Each element in the element type system should know its type (its corresponding meta type). After creating the element's object, `CreateObject` updates its type data member to point to the meta type.

- It updates the data members for the element's special attributes (if any).

- If the element has a parent (it is not the root element), it calls the function `SetChild` defined for the parent object to update its child data members.

- It calls the function `SetAttributes` to set the attribute values.

5.3 The DTD Manager

The DTD Manager is invoked by the Type Generator after the type generation is successfully completed.

The DTD Manager stores the DTD in the database as an object of the built-in element type DTD. The DTD object stores the DTD name, the DTD string and a types list. The types list contains a meta object for each element defined in that particular DTD. At document instantiation time, the meta objects are used by the Instance Generator to automatically instantiate SGML documents. Each meta type object can be asked to create an object of its corresponding element type by calling the member function `CreateObject`.

Note that, even though the DTD elements are represented by meta types, the DTD is still stored as a text string in the DTD object. This is because the DTD is needed by the SGML parser to validate the document instances. Regenerating the DTD string from the list of meta types objects is currently not possible since the information stored is not sufficient. For example, entity information is not stored in the database since the DTD Parser resolves entity references automatically. On the other hand, the SGML Parser will need such information since entities can be defined in the DTD and referenced in the document instance. Even if all the necessary information is encoded, the regeneration process will be quite costly as it will involve extensive querying to the database and string generation for the DTD. Since DTDs are usually not too long, storing the DTD string does not involve a significant overhead on the system.

Chapter 6

Automatic Document Entry System

The main focus of this thesis is to develop a system capable of supporting the automatic instantiation of SGML document instances in the multimedia database. To achieve this goal, the system components shown in Figure 6.1 have been developed.

The system consists of three major components:

- **Authoring Tools:**

  SGML documents are structured with markup information encoded together with the document contents. Authoring such documents manually can be a very tedious and time-consuming job especially for those documents conforming to complex DTDs. Therefore, there is a great need for SGML authoring tools that can support document authors in composing their documents. A number of authoring tools are commercially and publicly available. Examples of these include: *Near & Far* developed by Microstar[1] and the psgml extension of the Emacs editor. Psgml was

[1]For information about Microstar's products, see http://www.microstar.com.

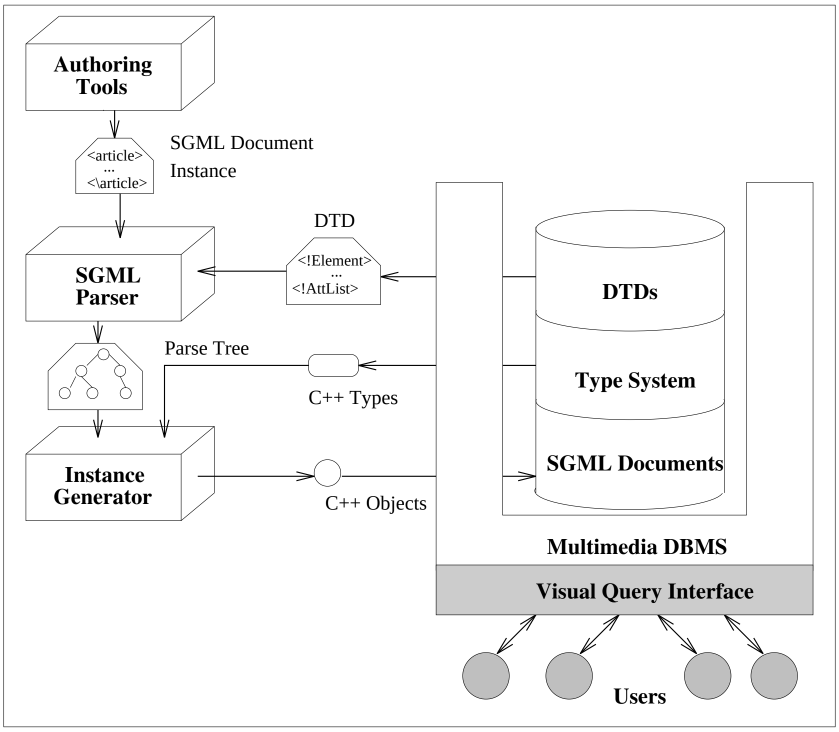

Figure 6.1: Architecture for Automatic Document Entry

examined in our project to explore the ability of coupling our system with authoring tools.

Another alternative for composing SGML documents is using conversion tools. Document authors can use their favorite tools and notations to compose documents as long as conversion tools are available to convert the notations to SGML.

- **The SGML Parser:**

  The SGML Parser parses an SGML document, created by an authoring tool to validate its conformance to its DTD. While parsing the document, the parser gathers information about the document structure and builds a parse tree. After the successful validation of the document instance, the Instance Generator is invoked.

- **The Instance Generator:**

  The Instance Generator traverses the parse tree and instantiates the objects in the database to represent the elements in the document. To instantiate these objects, the Instance Generator uses the meta type objects stored in the DTD object. Chapter 7 discusses the Instance Generator.

The rest of this chapter discusses the SGML Parser component of the system.

6.1 The SGML Parser

To instantiate SGML documents in the database, a parser is needed. The SGML parser should perform the following tasks:

- For every SGML document, its DTD should be fetched from the database and used in parsing. If the DTD is not found in the database, the document should be rejected.

- Validate the SGML documents to ensure they are error-free before inserting them into the database. If the document does not conform to its DTD, it should be rejected.

- Gather information about the document's parse tree. Every element in the parse tree should store a reference to its parent node, as well as other information. Such information is needed by the Instance Generator to automatically instantiate objects in the database that correspond to the document.

- Compose a continuous text string for the document contents without the markup information and the list of annotations to represent the document's elements.

- Invoke the Instance Generator if the document validation is sucessfully completed.

6.1.1 Developing the SGML Parser

Two alternatives were considered to develop the SGML Parser component of our system:

- Develop a new SGML Parser from scratch.
- Use an existing SGML Parser and retrofit it to fit our system requirements.

Since SGML is very complex, it would be difficult to develop a full-fledged SGML Parser within the time frame of this thesis. Also, the development of such a parser is not a research contribution. For these reasons, the first possibility was eliminated and the possibility of using an existing SGML Parser was explored.

Finding an SGML parser is not difficult, since there are a number of freeware and commercially avaliable SGML parsers. Among these are: *Arc SGML*, *Sgmls*, *Nsgmls*, *TEI Parser* and many others. After examining the different alternatives, *Nsgmls* version 1.0.1[2] was chosen.

sgmlnorm is a freeware parser developed by James Clark[3]. It is developed in C++ based on the *Sgmls* parser which was developed in C.

This SGML parser was chosen over the other parsers for a number of reasons:

- It was developed in C++ which makes it easier to integrate with ObjectStore, our object-oriented DBMS.
- Its object-oriented design made it easier to modify and reuse the code.
- It is a full-fledged SGML parser that provides support for a number of SGML advanced features.
- It is widely accepted and used in the SGML community (according to the discussions raised in the comp.text.sgml newsgroup).

[2]The parser actually used is called *sgmlnorm*. It is a variation of Nsgmls which uses the same C++ libraries (sp libraries).

[3]It can be obtained from ftp.jclark.com.

6.2 Modifications to the Original Parser

Modifications to the original version of *sgmlnorm* were made to satisfy our system requirements. Due to the complexity of *sgmlnorm* code (approximately 45K lines of C++ code) and the lack of documentation[4], I attempted to make my modifications as independent as possible of the *sgmlnorm* code. These modifications are discussed in this section.

6.2.1 Fetching the DTD from the Database

As mentioned in Section 2.1.2, every valid SGML document should have a `DOCTYPE` statement to specify the document root and the DTD's file name. The original version of *sgmlnorm* used this file name to load the DTD to be used in validating the document. In our system, the DTD is stored in the DTD object and it should be fetched from the database.

To achieve this goal, the SGML Parser queries the database for the DTD objects using the document root name defined in the `DOCTYPE` statement. If the DTD is not found, the system displays an appropriate error message and exits. Otherwise, the `dtdstring` for the DTD is fetched and written to a temporary file that can be read in the same way that the original *sgmlnorm* expected it.

6.2.2 Maintaining Data Structures

Even though *sgmlnorm* maintains data structures to represent the document instance, they are not easily mappable to the structures needed to invoke the Instance Generator. Therefore, I decided to maintain my own data structures independent of the original *sgmlnorm* data structures. A description of the data structures maintained is provided in Section 6.3. This also provides an easy upgrade path for our system as new *sgmlnorm* versions are released.

[4]When I started implementing the system in spring 1995, there was almost no documentation, currently some external documentation is avaliable at http://www.jclark.com/sp/index.htm.

6.2.3 Porting sgmlnorm to AIX

sgmlnorm was originally developed for Sun SunOS using *g++* as the compiler. It was successfully ported by Nelson Beebe to a number of different operating systems including Solaris.

Since our system uses ObjectStore on IBM RS/6000s, *sgmlnorm* had to be ported to *AIX*. To link with ObjectStore modules, the porting had to be done using AIX's native compiler (*xlC*). Unfortunately, the reported attempts made by Nelson Beebe to port the parser to AIX failed so I had to do the porting.

In porting *sgmlnorm*, I faced a number of problems. *sgmlnorm* makes extensive use of templates to support its functionality. Template handling in g++ and xlC are very different. xlC handles its templates by creating a `tempinc` directory that is used to store the code generated for handling templates. During compilation of programs, the `tempinc` directory is automatically created. At linking time, the generated code is compiled and linked to the executables. If a library is to be created, all the template objects should be explicity compiled and linked to the library.

After a number of attempts, the porting of *sgmlnorm* to xlC under AIX was successful.

6.2.4 Invoking the Instance Generator

The output for the original version of *sgmlnorm* is a list reporting the errors encountered during validation and a textual string representation of the document instance that is being parsed.

sgmlnorm has been modified to suppress the textual output. After the successful validation of the document, it invokes the Instance Generator with the information gathered from parsing. If the validation fails, errors are reported and the system is exited without any instance generation.

6.2.5 Changes to the Command Line Parameters

For an SGML document instance to be instantiated, it has to specify the database to be used. The specified database is used to fetch the DTD string as well as to model the document.

An option `-D` was added to the command line parameters of *sgmlnorm* to allow the document to specify the database name.

6.3 Data Structures

sgmlnorm was modified to create data structures while parsing the SGML documents. These data structures store information that is needed by the Instance Generator to instantiate the appropriate types in the database.

The information gathered by the SGML Parser is stored in an instance of the ParsingResults class. The ParsingResults class has the form:[5]

```
class ParsingResults {
   char *textBlock;
   TreeNode *root;
   int textSize;
   int bufSize;
}
```

The ParsingResults object stores the following information:

- the contents of the document as a continuous string after removing the markup information. During instantiation, this text string will be stored as an AtomicText that is referenced from the document root.
- the root of the document's tree structure which represents the document root specified in the `DOCTYPE` statement. Note that, the document's tree structure is represented as a linked list with each tree node maintaining a pointer to the next node in the tree in depth-first sequence, as well as a pointer to its parent node.

[5]For simplification, member functions declaration are removed from the class definitions presented.

- `textSize` stores the size of the document's contents string.
- `buffSize` stores the size of the allocated buffer for the `textBlock`.

For every element defined in the document, a node is created in the parse tree by instantiating a TreeNode object. The TreeNode object has the following structure:

```
class TreeNode {
   char *elementName;
   int startAnnot;
   int endAnnot;
   NodeAttribute *attrList;
   TreeNode *parentNode;
   TreeNode *nextNode;
}
```

The TreeNode models the following information:

- the element name as defined in the DTD. This will be used to retrieve the appropriate meta type from the DTD object.
- `startAnnot` and `endAnnot` are used to store the annotation information of the element; i.e. its start and end location in the text block. Note that, if the element is not annotated, the end location will be equal to the start location minus one.
- `attrList` stores points to the first attribute of the attributes' linked list.
- `parentNode` is a pointer to the element's parent in the document's hierarchy tree.
- `nextNode` is a pointer to the next node in the parse tree following a depth-first sequence. It is used to traverse the document nodes sequentially.

Each attribute defined in the element's attribute list is modeled using a NodeAttribute object which is in the form:

```
class NodeAttribute {
     char *attrName;
```

```
        char *attrValue;
        NodeAttribute *nextAttr;
    };
```

where `name` is the attribute name, `attrValue` is a string representation of the attribute value and `nextAttr` is a reference to the next node in the attribute list.

6.4 Limitations of the Current System

In the current system, there are a number of limitations. Most of them arise from the fact that only the text of the document instance and annotation lists are stored in the database. In some cases, there is no way of regenerating the original marked-up documents from the information in the database. This means that we are loosing some information in the process of modeling the document. To understand this point, let's look at a number of SGML features:

6.4.1 External and Parameter Entities

As mentioned in Section 2.1.1, external entities in SGML are a way of dividing the document into different files which can then be included in the main document instance. To define an external entity, a definition is added in the DTD:

```
<!ENTITY     chapter1  SYSTEM "chapter1.sgm">
```

This external entity can be included in any place in the document by referring to its name. For example:

```
<!DocType Book SYSTEM "book.dtd">
...
&chapter1
...
```

While parsing the document, the SGML Parser includes the external file in the document output. All the text is stored in the `textBlock` in the database as one string; there will be no indication of the presence of the external entity. If the same document is rewritten without the reference to the external entity (the text of the entity included in the same file), the modeling of the two documents will be exactly the same in our system.

The same argument applies for the parameter entities as well.

6.4.2 Marked Sections

The problem of loosing information in modeling the document is more obvious in the *marked section* feature of SGML. The concept of marked sections in SGML is the same as #ifdef in C.

Document author can group parts destined for different group of readers into the same document and decide which parts should be INCLUDE(d) and which parts should not. The following example clarifies the use of marked sections:

```
  <!doctype Book [
  <!ELEMENT Book          - -     (title, body)>
  <!ELEMENT title         - 0     (#PCDATA) >
  <!ELEMENT body          - 0     (#PCDATA) >

  <!ENTITY % ENGLISH      "INCLUDE">
  <!ENTITY % FRENCH       "IGNORE">
  <!ENTITY % DUTCH        "IGNORE">
]>

  <Book>
  <![ %ENGLISH [ <title> Practical SGML]]>
  <![ %FRENCH [ <title> SGML en pratique]]>
  <![ %DUTCH [ <title> SGML in de praktijk]]>
  <body>
  This example illustrates the use of marked sections.
  </Book>
```

While parsing this document, only the included text will be stored in the `textBlock`. If the modeled document is regenerated from the database, we will get the following output[6]:

[6]Document regeneration is not provided by the current system; however, it can be easily incorporated if the feature is needed.

```
<Book>
  <Title> Practical SGML  </Title>
  <Body>  This example is illustrates the use of marked sections. </Body>
</Book>
```

Note that, there is no indication of the presence of the marked section in our system. Also, since the rest of the document (the excluded parts) is not stored in the database, it cannot be regenerated.

This problem gains more importance when discussing update issues. Currently, our database is read-only; once SGML documents are inserted, their contents cannot be modified. For example, the author of the previous document cannot change the parameter entity values to use the French or the Dutch version instead of the English version. If updates are allowed in the system and the author can change these entity values, it will not be possible to model the new document from the information stored in the database.

There are two ways to solve this problem. The first and the simplest solution is to store the original document string in the database. This way, if re-modeling of the document is required, the missing information can be restored from the original string representation. Even though this approach is adopted in our system for DTDs (see Section 5.3), it is unacceptable for document instances. Multimedia documents are likely to be very large in volume (on the order of Mbytes or even Gbytes); thus replicating the document by storing its original format will impose a huge overhead on the system. An even more serious problem arises when the document contents are updated (if updates are allowed). How should we reflect the changes to the original text string?

The second solution is to model entities and marked sections as objects in the database in a way that enables the restoration of the original document. Using this technique, the problem is reduced to how to model entities to facilitate such restoration. The problem is not addressed in this thesis due to direct connection to update issues, Which will be considered next year (see Chapter 9).

Chapter 7

The Instance Generator

The Instance Generator is the system component responsible for modeling the document instance in the multimedia database. It is invoked by the SGML Parser after the successful validation of the document instance. The `ParsingResults` data structure (see Section 6.3), generated by the parser, is used by the Instance Generator to automatically instantiate objects in the database. These objects represent the SGML document's components. To achieve this goal, the following questions were addressed:

1. How can the system ensure the consistency of the database by ensuring the uniqueness of the inserted documents?

2. How can the Instance Generator automatically detect which object types to be instantiated?

3. How can the system keep track of parent nodes and update their child list every time a child object is created?

4. How can the Instance Generator resolve references within the document?

The next four sections discuss the four questions, respectively.

7.1 Ensuring Database Consistency

As discussed in Section 1.1, database consistency is an important concept. Maintaining database consistency requires that the system rejects the input of a document if it already exists in the database. This means that whenever a new SGML document is to be added to the database, the system should ensure that it is not equal to other documents. To achieve this goal, an important question needs to be answered: How to determine that two documents are equal?

Theoretically, two documents are equal if they have the same DTD, the same structure and the same contents. Discovering whether documents have the same DTD is simple. Since type extents are maintained for all user-defined types in our database, root objects of documents conforming to the same DTD will be maintained in the same type extent. Only those documents should be considered for comparison. However, there is no easy way to discover that documents have the same content or structure. Documents in our system are stored as a continuous text string with layers of annotations. Two documents are equal in structure and content if their text strings and their annotations are equal. Therefore, to ensure that the inserted document does not exist in the database, its text string and annotations have to be checked against all the documents in the specific type extent. This technique is inefficient and will propose a huge overhead on the system especially with a large database.

Another approach to solving this problem is to devise a method to uniquely identify the documents in the database. This can be achieved by defining an `ID` attribute in the document root attribute list. This attribute should be unique for documents conforming to the same DTD. Whenever a new document is added to the database, the root type extent is queried to check if there is a document with the same id in the database. If there is, the document is rejected; otherwise it is accepted. This approach has been adopted in the system. It is more efficient as it relieves the system from the time-consuming string matching required for the other approach. However, it relies on the definition of the `ID` attribute. If the

DTD does not define an `ID` attribute, there is no way for the system to ensure the database consistency. It is the responsibility of the DTD author to define such an attribute and to make it required for all the conforming documents to maintain a consistent database.

7.2 Automatic Instantiation of Objects

To model the document in the database, objects must be instantiated for every node in the parse tree. The Instance Generator must automatically detect the types of objects to be instantiated.

In our system, this is achieved by maintaining the types list in the DTD object. For every node in the parse tree, the types list is queried using the element name to get the meta type object corresponding to the node. This meta object is used to create the element object by calling the `CreateObject` member function. Since `CreateObject` is a virtual function, it is dynamically resolved to call the appropriate function. After creating the object, `CreateObject` calls `SetAttributes` to set the attribute values for the created object. `SetAttributes`, defined in ElementType, loops over the defined attributes in the document and calls `SetSpecificAttribute` (see Section 5.2.4).

7.3 Updating Child List

Another task the Instance Generator should perform is to update the parent's child list after instantiating its child objects. To achieve this, each child object should keep a pointer to its parent's persistent object. For every node in the parse tree, its parent's pointer is passed to `CreateObject`. After the object is created, the parent's child list is updated by calling the parent's `SetChild` (see Section 5.2.4). One problem with this approach is how can the child object know its parent's persistent pointer?

Every node in the parse tree has a `parentNode` data member. During parsing, the SGML Parser updates this data member to store a transient pointer to the parent. This reduces the problem to one of mapping the transient pointer to the persistent pointer?

To accomplish this mapping, the class `ParentsTable` has been defined. Whenever a persistent object is instantiated for a structured element, `ParentsTable` is updated to store the transient pointer and the corresponding persistent pointer. Each entry in the ParentsTable has the form:

```
class TableEntry {
     TreeNode *node;
     Structured *element;
     TableEntry *nextEntry;
}
```

where `node` is the transient pointer, `element` is the persistent object pointer and `nextEntry` is used to point to the next entry in the `ParentsTable` linked list.

For every node in the parse tree, before calling `CreateObject`, the `ParentsTable` is searched to find the parent's persistent pointer. This persistent pointer is passed to `CreateObject` as discussed earlier.

7.4 ID Reference Resolution

To solve the forward referencing problem discussed in Section 5.2.4, the Instance Generator delays reference resolution until all the document objects are instantiated.

There are a number of approaches to resolve references:

- After instantiation, the parse tree can be traversed to get the elements that have an `IDREF` attribute. For each element, the database can be queried to find the object with the referenced `ID`; then the ParentsTable[1]

[1]The ParentsTable would have to be extended to store the information for all elements in the document instead of only the structured elements.

can be searched to get the original object's persistent pointer. The pointer can be used to fetch the object from the database and update it.

- After instantiation, the database can be queried to get all the elements in the document that have an `IDREF` attribute. For each element in the list, the transient pointer can be fetched by searching the `ParentsTable`. The id referenced in the transient object can be used to query the database for the referenced object which will be used to update the element.

- During the instantiation of the objects, a dictionary can be maintained to keep track of the ids defined in the document, the elements that define them and those that reference them. This dictionary can then be used to resolve the references.

The first two approaches are inefficient since they involve time-consuming database queries and list searches. The only advantage is that no extra data structures must be maintained. Since maintaining the dictionary simplifies the id resolution problem and does not involve a huge overhead, the third approach has been adopted.

Whenever an element with an `ID/IDREF` is instantiated, the dictionary is checked. If there is an entry with the same id, the entry is updated to store the new reference; otherwise, a new entry is added to the dictionary. Each entry in the dictionary has the form:

```
class DictEntry {
      char *id;
      void *defElement;
      RefElement *refElement;
      DictEntry *nextEntry;
}
```

where `id` is the id value, `defElement` is a pointer to the element that defines the id, `refElement` is the list of elements referencing the id and `nextEntry` is a pointer to the next DictEntry in the `dictionary`.

The RefElement class has the following definition:

```
class RefElement {
      Element *referent;
      char *attrName;
      RefElement *nextReference;
}
```

where `referent` is a pointer to the element referencing the id, `attrName` is attribute name defining the IDREF and `nextReference` is a pointer to the next RefElement in the linked list.

After instantiation, the dictionary is used to resolve references. For each entry in the dictionary, the reference list is traversed and the elements are updated by calling `ResolveRef` for each element (see Section 5.2.4). `ResolveRef` takes the attribute name (`attrName`) and a reference to the element defining the id (`defElement`).

Chapter 8

Related Work

The use of DBMS for multimedia applications has gained the attention of many researchers in the last number of years. This is because multimedia applications can benefit from basic DBMS services such as transaction management, concurrency control and data distribution.

This chapter discusses a number of multimedia database systems that are reported in literature and compares their approaches to the approaches adopted in our system. Special attention is paid towards object-oriented database systems that support SGML and HyTime multimedia documents.

8.1 D-STREAT

At GMD - Integrated Publication and Information Systems Institute (IPSI), an application for handling structured documents, called *D-STREAT*, is currently being developed based on the object-oriented DBMS *VODAK*. *D-STREAT* is part of the *HyperStorM (Hypermedia Document Storage and Modeling)* which researches the use of object-oriented technologies to handle structured documents. The structured documents conform to the SGML and HyTime standards.

To meet the objective of their project, the following requirements are identified [BA96]:

- The database application should handle documents conforming to arbitrary DTDs.
- The system should have a powerful query model that supports queries built on document structure.
- The system should allow updates for documents that already exist in the database without repeating the process of parsing the entire documents.
- Document components' semantics should be available within the database to allow for querying based on meta information.
- Documents' conversion to HTML should be available for the documents to be accessed via the WWW. Since the conversion between different DTDs and the HTML DTD is not always straightforward, it can be specified by the DTD designer.

To administer documents of arbitrary types in *D-STREAT*, documents are fragmented in the database according to their logical structure. Every element in the DTD corresponds to a class in the database and every element in the document corresponds to a database object. In the first version of *D-STREAT*, described in [ABH94], the textual content of the document was not stored as a single continuous text string. It was fragmented and stored in the database objects that correspond to the different document components. Even though this technique is efficient in handling declarative queries and document component updates, it poses a performance overhead in certain access operations (such as retrieving entire documents), document insertion into the database and text-based searches. On the other hand, if the document is stored in an unstructured way (as a one flat object with one text string), declarative querying and updates will be costly.

In [B9̈5], an improvement to the first version is provided. Since the choice of the most appropriate data structure is highly dependent on element access pattern, the new version provides a hybrid approach for physical document representation. Using this approach, the DTD designer can specify which DTD element types should be represented as database objects and which ones should be *flat*. Since each flat element type is stored in the database as a one object, the text content of all elements types occurring within the type is stored as a one continuous string. This system is said to be *configurable* as it depends on the DTD's semantic information, provided by the DTD designer, to handle document management in an efficient way. Configuration is not only restricted to the internal representation of the DTD element types, but is extended to include several indexing techniques and mechanisms for handling a document's secondary structures.

To handle multiple DTDs, *D-STREAT* follows the following steps [B9̈5]:

- The DTD is parsed by a parser generator to generate the DTD as an SGML document instance of the *super-DTD*. The *super-DTD* describes the definition of the DTDs as defined in the SGML standard. For example, the definition of an `ELEMENT` statement is specified in the super-DTD as follows:

  ```
  <!ELEMENT ELEM  (ATTRIBUTE*)>
  <!ATTLIST ELEM
            elemName       NAME     #REQUIRED
            contentModel   CDATA    #IMPLIED
            FLAT           (YES|NO|...) NO
   ...  >
  ```

  The super-DTD instance differs from the original DTD only on the syntactic level. For example, an element defined in the DTD as:

  ```
  <!ELEMENT   article    (frontmatter, body)>
  ```

  will be generated in the super-DTD instance as:

  ```
  <ELEM  elemName="article"  contentModel="(frontmatter, body)"
          FLAT=NO ...>
  ```

  The `FLAT` attribute defined on `ELEM` specifies whether the element should be flat or non-flat. By default, the `FLAT` attribute value is set to `NO` for all element types. This default can be changed, after the super-DTD instance is generated, by overwriting the `FLAT` attribute value with value `YES`. If an element is specified as flat, all element types contained in it will be flat. For simplification, the DTD designer has to specify only the top-level flat element types (i.e.: not contained in a flat element type). The system automatically detects the nested types and treats them as flat types. To achieve this, a tool for analyzing DTDs called *the DREAM-Parser* is extended and integrated in the system. This feature is advantageous especially for big DTDs with a lot of element types.

Besides the super-DTD instance, the parser generator generates a document parser for parsing instances of that DTD.

- The super-DTD instance, generated by the parser generator, is parsed by a parser for super-DTDs[1]. The parser validates the document and issues the database commands that creates database objects to represent the DTD.

- For each element in the DTD, new classes are created [2]. For every non-flat element type in the DTD, an *element-type* class is created; whereas for every flat element type, an instance of a virtual element type, FLAT_TYPE, is created. To encapsulate the internal representation of element types, a class called ELEMENT_TYPE is introduced. ELEMENT_TYPE contains an instance for each element type in the DTD. All methods invoked on element types are invoked on the ELEMENT_TYPE instance which forwards the invocation either to the element-type class or the instance of FLAT_TYPE.

To support HyTime AF, a metaclass is defined for each architectural form.

[1]The parser is an extension of the publicly avaliable *Amsterdam Parser (ASP)*.

[2]Class creation is achieved by using the *metaclass* feature of VODAK. In VODAK, instances of metaclasses are themselves classes. They have instance creation methods whose invocation lead to the creation of new classes. Therefore, system shutdown is not necessary to extend the type system.

These metaclasses are called *HyTime metaclasses.* Each Hytime element type in the DTD corresponds to an element-type class and an instance of the relevant HyTime metaclass. Each HyTime element in the document instance is represented by two objects in the database: an instance of the element-type class and an instance of the HyTime element-type class storing the Hytime information. This is necessary, since VODAK does not support multiple inheritance.

- As mentioned earlier, the configuration information provided in the super-DTD instance is used to either create an element-type class or an instance of FLAT_TYPE. To speed the document instantiation process, this information is stored in a *configuration file* that is made avaliable to the document parser. This is opposed to storing the information in the database, since accessing the database will impose a performance overhead on the system.

Once the new classes for the DTD elements are created, SGML documents conforming to that DTD can be inserted in the database. These documents are parsed using the specific document parser generated in Step 1. This parser is an extension of the Amsterdam Parser (ASP). It checks the document's conformance to the DTD and instantiates database objects to represent the document.

8.1.1 Comparison Between D-STREAT and Our System

Compared to the multimedia applications described in literature, *D-STREAT* is the one that is most similar to our system. Both systems use an object-oriented DBMS to support arbitrary document types and achieve the dynamic addition of DTD types without system shutdown. However, their are a number of differences:

- Since both systems are different in their underlying DBMS (VODAK for *D-STREAT* and ObjectStore for our project), they differ in the way they perform dynamic addition of types. In contrast to our structured type system, *D-STREAT* does not explore reusability and data abstraction to reduce code redundancy. Its database schema has a flat class hierarchy. This simplifies the generation of types, since there is no need for the automatic detection of supertypes, but increases code redundancy.

- Since *D-STREAT* uses the Amsterdam Parser for parsing DTDs as well as document instances, DTDs are transformed into document instances before being parsed. Our system uses a DTD-parser that understands DTD syntax so there is no need for such transformation.

- *D-STREAT* and our system handle the storage of documents' textual content differently. In our system, a document is stored as one continuous string with layers of annotations. In *D-STREAT*, each element type instance, with the exception of flat elements, stores a portion of the document's text content. Even though our approach is efficient in retrieving an entire document, it poses a considerable overhead if updates are considered.

- In our system, the meta information about the DTD elements is stored in the meta type system. Whenever a document instance is added to the system, the database is queried to get the necessary information. Even though this technique increases the document instantiation time (since it involves costly database access), it encapsulates all the necessary information in the database which is a desirable feature. On the other hand, *D-STREAT* uses a configuration file to make the meta information available to the document parser.

- In contrast to *D-STREAT*, the current version of our system does not allow updates for documents that already exist in the database. It, also, does not provide a number of facilities that are supported by *D-STREAT*, such as system configurability, indexing mechanisms and document conversion to HTML.

8.2 VERSO

VERSO, developed at INRIA, France, is a database system that aims at modeling SGML documents in an object-oriented database. O_2 was chosen as the underlying object-oriented DBMS for its sophisticated type system and its extensible query language O_2SQL. Two major goals are identified for the system [CACS94]:

- To extend the O_2 data model to facilitate the mapping from SGML documents to O_2 instances. Two extensions of the data model are proposed to model the DTD constraints, namely the use of occurrence identifiers and connectors (see Section 2.1.1). These extensions are: ordered tuples and union types. Ordered tuples are used to represent the ordering of elements in the DTD. They model the "," connector. Union types are used to describe alternative content models of elements (separated by the "|" connector). For example, the following DTD element definition:

  ```
  <!ELEMENT section  ((title, body+) | (title, body*, subsec+))>
  <!ATTLIST section
            status    (final|draft)  draft   #IMPLIED>
  ```

  is mapped to the following O_2 declaration:

  ```
  class Section public type union
                (a1: tuple(title:Title, bodies:list(Body)),
                 a2: tuple(title:Title, bodies:list(Body),
                           subsecs:list(Subsec)))
                constraints:((a1.title!= nil, a1.bodies!= list())
                           | (a2.title!= nil, a2.subsecs!= list())),
                               status in set ("final", "draft")
  ```

  where `a1` and `a2` are system supplied names for the unnamed SGML `name groups`.

- To extend the O_2 query language to deal with the requirements of SGML document retrieval. These extensions include: (1) the *contains* predicate to handle querying on strings, (2) implicit selectors to select the correct path while handling queries over union types, (3) two new sorts: `PATH` and `ATTR` to query data without exact knowledge of its structure.

To represent SGML documents in the database, DTDs are mapped into O_2 schema and a document instance into corresponding objects. This is achieved by extending the Euroclid parser with semantic actions. These actions are used to annotate the DTD (or some intermediate BNF grammer). As shown in the example, each SGML element definition in the DTD is mapped into an O_2 class with a type, some constraints and a number of default behaviors (such as displaying, setting and getting attribute values, etc).

VERSO provides some predefined classes, such as Text and Bitmap, from which basic SGML types should be derived. This means that the textual document elements store fragments of the document's text content.

8.2.1 Comparison between VERSO and Our System

Similar to our system, *VERSO* handles documents with arbitrary types. The differences between the two systems can be summarized as follows:

- In contrast to *VERSO*, DTD constraints are not modeled in our database schema; therefore, our type system does not enforce these constraints. As discussed in Section 5.2.2, this is not a significant drawback, since the constraints are enforced by the SGML parser during document instantiation.

- Our query model is dependent on ObjectStore's query model; no extensions have been added to handle SGML element retrieval. Since ObjectStore does not allow for queries with incomplete structure knowledge, our data model does not support this feature.

- Similar to *D-STREAT*, the textual content of the document in *VERSO* is stored as fragments in the database objects representing the textual document elements. This imposes a performance overhead while fetching the entire document.

- *VERSO*'s type system is rather flat with no inheritance relationship between classes. This increases code redundancy and decreases data abstraction.
- *VERSO* does not provide any native support for HyTime documents. However, the authors claim that their query language extensions are particularly suited for the current extensions of SGML to multimedia and hypermedia documents (i.e. HyTime documents).

Chapter 9

Conclusion and Future Work

This thesis provides an automatic way of inserting SGML documents in an object-oriented multimedia database. This is achieved by instantiating objects in the database to correspond to the document's components. To automatically instantiate these objects, the database is dynamically queried to retrieve meta information about the objects' types. This meta information is modeled in the meta type system.

The major contributions of this thesis can be summarized as follows:

- A meta type system was developed to store meta information about DTD elements. The type system is sufficiently general and extensible to support different multimedia classes. It is implemented as a kernel structured type system that defines a number of built-in types to model the characteristics common to multimedia meta types.

- A facility to extend the meta type system to support new document types was developed. This is achieved by generating C++ code to define new meta types to represent the DTD element types. The new meta types correspond to the new elements defined in the DTD and are dynamically added to the database schema to model the DTD types.

- A facility to extend the element type system was developed. For every new type created to represent a new DTD element, a number of member functions are generated. These functions perform tasks that are necessary for the correct instantiation of documents' objects such as updating attribute values, updating parent lists and resolving id references.

- The document entry system was developed which is composed of the SGML Parser and the Instance Generator.

- The coupling of the system with the psgml extension of the Emacs editor has been studied to explore the ability of coupling our system with authoring tools.

In the future, there are a number of modifications and enhancements that can be introduced to the system:

- **Unifying the two Parsers**

  Currently, the system uses two different parsers to parse the DTDs (*dpp* parser) and the document instances (*sgmlnorm* parser). This imposes an overhead of having to maintain the code for two parsers. Since *sgmlnorm* version 1.0.1 gathers information about the DTD while parsing the documents, it can be used to replace the DTD Parser currently used.

- **Reusability**

  To avoid schema evolution overhead and semantic heterogeneity problems in our current system, reusability of user-defined types was not implemented. In the future, reusability of types especially across multiple DTDs, should be exploited. Elements having the same definitions across different DTDs should be represented by common types in the type system. This will eliminate code redundancy in the system and enrich the query model of the database. To achieve reusability a number of problems such as the semantic heterogeneity and name conversion problems should be addressed.

- **Updates**

  The database system supports a read only environment where documents in the database cannot be updated. Since a document is modeled as a single continuous text string with lists of annotations indexed from the start of the string, updates are very costly especially for large documents. This is because updating one element will involve updating the annotations of all annotated elements positioned in the document after the updated element.

  To support updates in our system, relative annotations should be explored where the original text string should be stored as substrings and annotations are stored relative to the beginning of the substrings. This will decrease the update overheads. A question that needs to be addressed is: How can we divide the text string into substrings with as little impact as possible?

  Also, as discussed in Section 6.4.2, if updates are to be supported, entities and marked sections should be modeled as types in the type system and not replaced in the text as in the current approach.

- **Indexing**

  The current system does not make use of indexes to improve access efficiency since there is no automatic way of discovering which elements will be accessed more frequently and consequently benefit by an index; maintaining indexes on all elements in the DTD will be very costly and inefficient. Future research will aim at how to detect the elements that need indexing? One solution to this problem is to give the DTD developer the ability to indicate in the DTD which elements should be indexed.

- **Generalizing the Query Interface**

  The visual query interface developed in [EM95] is specific to the news-on-demand application. This query interface should be generalized to support different document types.

- **The development of a Query Processor and Optimizer**

  Currently, any query interface that needs to query the database will interact directly with ObjectStore. Future work includes the development of a query processor and optimizer that support content-based queries of images and videos. Once the query processor and optimizer are integrated in the system, the query interface will interact with it rather than with ObjectStore.

- **Using TIGUKAT as the DBMS**

  Currently, our multimedia database system is built as a layer on top of ObjectStore. In the long run, TIGUKAT [OPS+95], an extensible object-oriented database, currently under development at the Laboratory for Database System Research at the University of Alberta, will replace ObjectStore. TIGUKAT, as opposed to ObjectStore, has inherent support for multimedia objects.

Bibliography

[ABH94] K. Aberer, K. Böhm, and C. Hüser. The prospects of publishing using advanced database concepts. In *Proceedings of the Conference on Electronic Publishing*, pages 469–480, April 1994.

[B9̈5] K. Böhm. Building a configurable database application for structured documents. Arbeitspapiere der GMD 942, GMD-IPSI Darmstadt, 1995.

[BA96] K. Böhm and K. Aberer. Hyperstorm - administering sturctued documents using object-oriented database technology. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 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.

[EM95] G. El-Medani. A visual query facility for multimedia databases. Master's thesis, University of Alberta, Department of Computing Science, 1995.

[NKN91] S. Newcomb, N. Kipp, and V. Newcomb. The HyTime hypermedia/time-based document structuring language. *Communications of the ACM*, 34(11):67–83, 1991.

[OPS+95] M. T. Özsu, R.J. Peters, D. Szafron, B. Irani, A. Lipka, and A. Munoz. Tigukat: A uniform behavioral objectbase management system. *The VLDB Journal*, 4(3):445–492, 1995.

[Org86] International Standards Organization. *Information Processing - Text and Office Systems - Standard Generalized Markup language (ISO 8879)*. International Organization for Standardization, first edition, 1986.

[Org92] International Standards Organization. *Hypermedia/Time-Based Structural Language: Hytime (ISO 10744)*. International Organization for Standardization, first edition, 1992.

[OSEMV95] M.T. Özsu, D. Szafron, G. El-Medani, and C. Vittal. An object-oriented multimedia database system for a news-on-demand application. *Multimedia Systems*, (3):182–203, 1995.

[PO95] R. J. Peters and M. T. Özsu. Axiomatization of dynamic schema evolution in objectbases. In *Proceedings of the 11th International Conference on Data Engineering*, pages 156–164, March 1995.

[Sch96] M. Schöne. A generic type system for an object-oriented multimedia databasesystem. Master's thesis, University of Alberta, Department of Computing Science, 1996.

[vH94] Eric van Herwijnen. *Practical SGML*. Kluwer Academic Publishers, second edition, 1994.

[Vit95] C. Vittal. An object-oriented multimedia database system for a news-on-demand application. Master's thesis, University of Alberta, Department of Computing Science, 1995.

Appendix A

A DTD for News-Articles

```
<!-- 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              - -  (doc-alts?, frontmatter, async, sync?)>
<!ELEMENT doc-alts             - -  (text, text-variant*)>
<!ELEMENT text                 - -  EMPTY>
<!ELEMENT text-variant         - -  EMPTY>
<!ELEMENT frontmatter          - -  (edinfo, hdline, subhdline?, abs-p)>
<!ELEMENT edinfo               - -  (loc & date & source & author+
                                         & keywords & subject)>
<!ELEMENT (loc|source|subject) - -  (#PCDATA)>
<!ELEMENT (hdline|subhdline)   - -  (#PCDATA)>
<!ELEMENT date                 - -  (#PCDATA)>
<!ELEMENT (author|keywords)    - -  (#PCDATA)>
<!ELEMENT abs-p                - -  (paragraph)>
<!ELEMENT async                - -  ((section|figure|text|link)*,
                                     (image-variant|text-variant)*)>
<!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              - -  (image, figcaption?) >
<!ELEMENT figcaption          - -  (#PCDATA) >
<!ELEMENT image               - -  EMPTY>
<!ELEMENT sync                - -  (audio-visual+)>
<!ELEMENT audio-visual        - -  (x, y, time, av-fcs, av-extlist+,
                                        (audio-variant|video-variant|
                                        stext-variant)*, stream*)>
<!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)
                              - -  (#PCDATA)>
<!ELEMENT (image-variant|audio-variant|video-variant|stext-variant)
                              - -  EMPTY>
<!ELEMENT stream              - -  EMPTY>
<!ENTITY  % variant-attbs
          "id          ID      #REQUIRED
          format       CDATA   #REQUIRED
          streamspec   IDREFS  #REQUIRED
          site         CDATA   #REQUIRED">
<!ATTLIST article
          HyTime      NAME    #FIXED HyDoc
          id          ID      #REQUIRED
          language    CDATA   #IMPLIED>
<!ATTLIST quote
          source      CDATA   #IMPLIED>
<!ATTLIST author
          bio CDATA   #IMPLIED
          designation CDATA   #IMPLIED
          affiliation CDATA   #IMPLIED>
<!ATTLIST text
          MM          NAME    #FIXED Text
          id          ID      #REQUIRED
          price       CDATA   #IMPLIED
          variantspec IDREFS  #REQUIRED>
<!ATTLIST text-variant
          MM          NAME    #FIXED TextVariant
          id          ID      #REQUIRED
          filename    CDATA   #REQUIRED
          format      CDATA   #REQUIRED
          language    CDATA   #REQUIRED
          size        NUMBER  #REQUIRED>
<!ATTLIST image
          MM          NAME    #FIXED image
          id          ID      #REQUIRED
          price       CDATA   #IMPLIED
```

```
          variantspec IDREFS  #REQUIRED>
<!ATTLIST image-variant
          MM          NAME    #FIXED ImageVariant
          id          ID      #REQUIRED
          filename    CDATA   #REQUIRED
          format      CDATA   #REQUIRED
          size        NUMBER  #REQUIRED
          width       NUMBER  #REQUIRED
          height      NUMBER  #REQUIRED
          color       CDATA   #REQUIRED>
<!ATTLIST audio-visual
          id          ID      #REQUIRED>
<!ATTLIST x
          HyTime      NAME    #FIXED axis
          id          ID      #REQUIRED
          axismeas    CDATA   #FIXED "virspace"
          axismdu     CDATA   #FIXED "1 1"
          axisdim     CDATA   #FIXED "1280">
<!ATTLIST y
          HyTime      NAME    #FIXED axis
          id          ID      #REQUIRED
          axismeas    CDATA   #FIXED "virspace"
          axismdu     CDATA   #FIXED "1 1"
          axisdim     CDATA   #FIXED "1024">
<!ATTLIST time
          HyTime      NAME    #FIXED axis
          id          ID      #REQUIRED
          axismeas    CDATA   #FIXED "SISECOND"
          axismdu     CDATA   #FIXED "1 1"
          axisdim     CDATA   #FIXED "3600">
<!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 SISECOND">
<!ATTLIST audio
          HyTime      NAME    #FIXED event
          MM          NAME    #FIXED audio
          id          ID      #REQUIRED
          price       CDATA   #IMPLIED
          variantspec IDREFS  #REQUIRED
          exspec      IDREFS  #REQUIRED
```

```
     --  1 exspec for each variant, or 1 for all -->
<!ATTLIST video
         HyTime      NAME    #FIXED event
         MM          NAME    #FIXED video
         id          ID      #REQUIRED
         price       CDATA   #IMPLIED
         variantspec IDREFS  #REQUIRED
         exspec      IDREFS  #REQUIRED
     --  1 exspec for each variant, or 1 for all -->
<!ATTLIST stext
         HyTime      NAME    #FIXED event
         MM          NAME    #FIXED stext
         id          ID      #REQUIRED
         price       CDATA   #IMPLIED
         variantspec IDREFS  #REQUIRED
         exspec      IDREFS  #REQUIRED
     --  1 exspec for each variant, or 1 for all -->
<!ATTLIST audio-variant
         MM          NAME    #FIXED AudioVariant
         %variant-attbs
         duration    NUMBER  #REQUIRED
         samplerate  NUMBER  #REQUIRED
         bps         NUMBER  #REQUIRED
         quality     CDATA   #REQUIRED
         language    CDATA   #REQUIRED>
<!ATTLIST video-variant
         MM          NAME    #FIXED VideoVariant
         %variant-attbs
         duration    NUMBER  #REQUIRED
         width       NUMBER  #REQUIRED
         height      NUMBER  #REQUIRED
         framerate   NUMBER  #REQUIRED
         bitrate     NUMBER  #REQUIRED
         color       CDATA   #REQUIRED>
<!ATTLIST stext-variant
         MM          NAME    #FIXED StextVariant
         %variant-attbs
         language    CDATA   #REQUIRED>
<!ATTLIST stream
         MM          NAME    #FIXED stream
         id          ID      #REQUIRED
         uoi         NUMBER  #REQUIRED
         size        NUMBER  #REQUIRED>
<!ATTLIST av-extlist
         HyTime      NAME    #FIXED extlist
         id          ID      #REQUIRED>
<!ATTLIST (xdimspec|ydimspec|tdimspec)
         HyTime      NAME    #FIXED dimspec
         id          ID      #REQUIRED>
```