University of Alberta

A Generic Type System for an Object-Oriented Multimedia Database System

by

Manuela Schöne ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfill ment of the requirements for the degree of Master of Science.

Department of Computing Science

Edmonton, Alberta
Fall 1996

ISBN 0-612-18322-X

To My Parents

Abstract

This thesis describes the design of a generic multimedia database system that supports a wide class of documents. The design is characterized by an object-oriented approach and a strict adherence to the international standards SGML and HyTime. In order to support different multimedia applications, a multimedia type system must be flexible and extensible. This is achieved by defining a number of built-in types that model primitive multimedia objects, the spatial and temporal relationships between them, and characteristics that are common to all SGML documents and their components. Whenever support for a new class of documents is added to the multimedia database system, types modeling the characteristics specific to this class of documents are dynamically created and added to the type system. These types model the components and structure of the new document class.

Acknowledgements

I would like to take this opportunity to thank my supervisors, Dr. Özsu and Dr. Szafron for their guidance and assistance throughout my thesis research. I would also like to acknowledge the help of the other examiners of my thesis, Dr. Montgomerie and Dr. Sorenson. Their constructive criticisms and suggestions have been incorporated into the final version of my thesis.

Thanks also go to Dr. Elio for chairing my thesis defence and to Paul Iglinski, Sherine El-Medani, and Andreas Junghanns for all sorts of interesting discussions and ideas, and for proofreading my thesis.

Finally, I would like to acknowledge the financial support from the Department of Computing Science at the University of Alberta, by way of Graduate Teaching Assistantship and the M.Sc. Research Award.

Contents

List of Figures

Chapter 1

Introduction

In the last few years, multimedia systems have gained much popularity. *Multimedia* refers to the integration of text, images, audio, and video in a variety of application environments. Application domains are, among others, education, medicine, teleconferencing, and computer games. Multimedia applications have enormous data management requirements, since multimedia data is usually very large and complex. The size and complexity of the data stresses the capabilities of current database systems and leads to new database research.

The subject of this thesis is the design of a multimedia database system that can manage a wide class of multimedia documents.

1.1 The CITR Broadband Project

This thesis is part of a major project funded by the Canadian Institute for Telecommunications Research (CITR). The goal of this project is to research and prototype enabling technologies for distributed multimedia applications. The project was started in 1993 with a six year duration. The multimedia system under development consists of five major components that are developed by research groups at various Canadian universities. At the University of Alberta, the focus is on the data management component of the project, which involves storage, management, access, and retrieval issues in multimedia databases. The other components are the distributed continuous media file system (CMFS) (University of British Columbia), the quality of service (QoS) management, negotiation, monitoring and control component (University of Montreal), the synchronization component (University of Ottawa), and the scalable video encoding component (INRS). The system architecture design and integration work is done at the University of Waterloo.

In the first phase of the project, a multimedia system prototype was developed using *news-on-demand* as the target application. The design of the multimedia database management system (multimedia DBMS) for this application was done at the University of Alberta. Since the relational data model, which is [illegible]ly the most popular data model for a variety of applications, has difficulties in representing the complex data present in multimedia applications, an object-oriented approach was chosen to model, manage, and store the multimedia data. When designing an object oriented multimedia DBMS, the design of the type system is of fundamental importance, since the type system design can be viewed as the database schema design. The database schema, as such, limits which applications are supported by the system. For the type system design, three important issues must be considered: first, the modeling of the basic media components of the document (e.g. text, image, audio, and video); second, the representation of the document structure; and third, the capture and storage of meta information about the multimedia objects, such as spatial and temporal relationships between the multimedia objects. These issues were incorporated into the type system design for the news-on-demand application. The basic media components are modeled as *atomic* types in the type system, the representation of the document structure is done by strictly following the SGML[1] standard for document representation, and the spatial and temporal relationships between the multimedia objects are modeled in accordance with the HyTime[2] standard.

[1]Standard Generalized Markup Language.
[2]Hypermedia/Time-based Document Structuring Language Standard.

1.2 Motivation

The initial type system design for the multimedia DBMS was based on the news-on-demand application. Types that were necessary to model components of multimedia news-on-demand documents were "hard-coded". The whole type system was developed statically to provide support only for news-on-demand documents.

My thesis is that the multimedia type system, developed for the news-on-demand application, can be generalized to support a wide range of multimedia applications. Specifically, any applications based on SGML/HyTime documents can be supported.

The news-on-demand type system was used as the basis for the new design. A detailed description of the news-on-demand type system can be found in [Vit95]. Each class of SGML documents needs specific types to model the document components and relationships between them, since documents belonging to different document classes may consist of different components and may have a totally different structure. Making the type system sufficiently general to support all conceivable document classes is a challenging task. My approach was to design a number of predefined types, called "built-in" types. These types model characteristics common to all types that model document components. Whenever support for a new class of documents is added to the system, new types must be dynamically added to the type system that model the characteristics specific to the new class of documents. Thus, the database schema must be dynamic, in contrast to the static news-on-demand type system. Besides the type system design, this thesis introduces a new mechanism to realize the necessary dynamic type creation.

Currently, a closely related project is under way at the University of Alberta. It deals with automating the insertion of SGML documents of arbitrary types into the database. This work is described in [EM96].

1.3 Thesis Overview

This thesis is organized as follows. Chapter 2 gives an overview of the SGML and HyTime standards. Chapter 3 describes the overall system architecture, including the components for extending the type system and the components for automatic document entry. Chapter 4 discusses possible approaches for dynamic type creation. It looks at schema modification facilities present in different object-oriented database systems and suggests an approach for dynamic type creation in our system. Chapter 5 describes the design of the generic multimedia type system. The type system has two parts: the Atomic Type System for the modeling of the basic multimedia objects, and the Element Type System for the modeling of the document components and their inter-relationships. Chapter 6, 7, and 8 discuss the three main system components that support new classes of multimedia applications: the DTD Parser, the Type Generator, and the DTD Manager. Furthermore, the interface to the system components that provide facilities for automatic document entry is described. These system components have been developed as part of the work described in [EM96]. Chapter 9 reviews related work and compares it with the approach taken in this thesis. Chapter 10 presents some conclusions and gives a preview of future work.

Chapter 2

SGML/HyTime Standard

In the CITR project, the *Standard Generalized Markup Language* (SGML) and the *Hypermedia/Time-based Document Structuring Language Standard* (HyTime) have been chosen for document representation. Both are ISO standards that have gained much popularity in the last couple of years. For example, the Hypertext Markup Language (HTML), an application of SGML, is the document standard used for the World Wide Web. Because of the widespread use of these standards, many tools have already been designed that support authors in creating their documents. One example is the psgml extension to the Emacs editor that supports the creation of SGML documents. This tool was used in this project. Designing a multimedia database system based on these standards makes it possible to use a multitude of these existing tools and applications together with the multimedia database.

This chapter will give a short overview of SGML/HyTime. For more details see [Vit95], [vH94], and [NKN91].

2.1 Document Structure and SGML

The logical components of a document (e.g. chapters, sections, and paragraphs of a book) are hierarchically ordered. SGML is a generalized markup meta-language which can be used to specify this structure for any class of documents. The markups mark the boundaries of the logical components and are of the following form:

```
<book>
   <title> Introduction to SGML <\title>
   <chapter>
       This is the introduction to the first chapter.
       <section> This is the first section of the first chapter.
           <subsection>
               This is the first subsection.
           <\subsection>
           <subsection>
               This is the second subsection.
           <\subsection>
       <\section>
       ...
   <\chapter>
   ...
<\book>
```

Note that *<component>* is a *start tag* indicating the beginning of a logical component in the text and < *\component>* is an *end tag* that specifies the end of this component. SGML documents have a tree structure that may be analyzed by computers and can easily be understood by humans. The nodes of the tree are the logical components of a document. In SGML terminology, they are called *elements*. The subelements of an element are its *content*. Only the tree nodes that contain **(#PCDATA)** in their content declaration hold elementary *data content*. In SGML syntax, text strings are called **(#PCDATA)**.

SGML is architecture independent. It has been designed to enable text interchange and is intended for use in the publishing field. However, it can also be applied in other areas. SGML has no constraints on the data formats used to store the document, and it separates the description of the document's content and structure from the presentation layout.

An SGML document has, in addition to its marked-up data, two formal parts associated with it: the *SGML declaration* and the *document type definition* (DTD). The SGML declaration specifies which character sets (ASCII or others) and delimiters should be used to define an SGML document. If no specific SGML declaration is given by the author, the default SGML declaration is used. This default declaration is predefined and cannot be modified by authors. The document type definition will be discussed in the following section.

2.2 Document Type Definition

SGML standardizes only the meta-syntax of an SGML document: the syntax is standardized by the *DTD*. A DTD defines the rules for marking up a class of documents. It specifies the logical elements of the documents, their attributes, and the relationships between the elements.

A DTD is written in SGML by the document designer for each category of documents being used. For example, all HTML documents are based on the HTML-DTD, books will follow a Book-DTD, and memos will conform to a Memo-DTD. In [Vit95], a News-DTD was developed for the news-on-demand application which specifies the document structure of all news-documents being stored in the multimedia-news database.

A DTD consists of a number of BNF-like production rules, one for each element to be defined. Each element definition consists of the *element name*, a number of optional or mandatory *attributes*, and a *content model*. Attributes do not belong to the content of an element. Rather, they provide further information about the element that defines them. Each attribute has a name, an attribute type (e.g. **CDATA** for character data, **NUMBER** for a number), and a default value (e.g. **#REQUIRED** for required attributes, **#IMPLIED** for optional attributes) associated with it. The content model specifies which elements are allowed in the element's content. Elements in the content model are followed by occurrence indicators and linked together by connectors. SGML defines the three occurrence indicators "?", "+", and "*", and the three connectors ",", "&", and "|". "?" indicates that the element is optional; "+" means required and repeatable; and "*" represents optional and repeatable. If two elements are connected using ",", they have to follow each other in the given order. If the connector "&" is used, both operands must appear in the element's content, but the order is irrelevant. "|" indicates that either operand must occur, but not both.

The example below illustrates a simple DTD that defines the four elements: **book**, **chapter**, **section**, and **subsection**.

```
<!ELEMENT book       - - (title, chapter+)>
<!ELEMENT chapter    - - (section|#PCDATA)*>
<!ELEMENT section    - - (subsection|#PCDATA)*>
<!ELEMENT subsection - - (#PCDATA)>
<!ELEMENT title      - - (#PCDATA)>
<!ATTLIST book
          language   CDATA   #IMPLIED>
```

The root element of the DTD-element-tree is **book**. Its content model consists of a **title** element followed by at least one **chapter** element. **book** has the optional attribute **language** to specify the language in which the book is written. If an element's content model is defined as **(#PCDATA)**, as in the case of **subsection** and **title**, this means that the element is a leaf node in the tree. It has no further structure and consists of textual data only.

2.3 HyTime Overview

HyTime is an ISO standard for structured representation of hypermedia and time based information. A document is seen as a set of concurrent time-dependent events, such as audio and video. HyTime uses SGML syntax for describing these events.

HyTime is defined as a set of rules, called architectural forms, that can be applied when designing a DTD. Using these architectural forms, multimedia, hypertext, hypermedia, time- and space-based documents can be modeled. A DTD contains only those semantics of HyTime that are needed for its specific class of documents. A DTD element can be defined as a HyTime element by giving it an attribute with the name **HyTime** and values that are specified by the HyTime standard.

```
<!ELEMENT book       - - (title, chapter+)>
  ...
```

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

Among others, there is an architectural form called HyDoc defined which represents the root element of a HyTime document. The example above shows the use of this architectural form. The architectural form is specified by the value the HyTime attribute is set to, in this case HyDoc. Each HyDoc element (in this case book) must have an attribute called id that is used as unique identifier when referencing the element.

HyTime Modules

HyTime consists of several modules, each of which describes a group of concepts and architectural forms. These modules are: the base module, the location address module, the hyperlinks module, the measurement module, the scheduling module, and the rendition module. Each module may use features defined in modules lower in the hierarchy. For example, the base module provides facilities for object identification. Object identification is needed, e.g., in the hyperlinks module to build links between objects. Each DTD that uses HyTime architectural forms must declare the modules in which they are defined. The example below shows how this is done.

```
<?HyTime support base>
<?HyTime support hyperlinks>
```

Here are brief descriptions of the modules that are defined as part of the HyTime standard:

Base module: This module must always be declared if HyTime is used. It defines the basic HyTime concepts (such as architectural forms) and terminology, and includes facilities that are available regardless of which of the other modules are supported. These include facilities for: hyperdocument management, HyTime identification, and coordinate addressing. For example, the base module defines the ID attribute which is the most important HyTime attribute. Any HyTime element may declare and use this ID attribute. It is required in many cases by the definitions of architectural forms (e.g. by the HyDoc architectural form shown in the example above).

Location address module: Using unique identifiers (IDs) and identifier references (IDREFs) is a simple way to refer to particular elements in the document. However, the usefulness of this addressing method is limited to the elements that have an ID. The location address module contains architectural forms to locate objects that do not have unique identifiers in the document's name space. For example, a coordinate location address describes an object by its position along a list of objects: "the second object in this list", or a semantic location address describes objects by some property they have: "the first object with an age attribute whose value is 27."

Hyperlinks module: This module consists of five architectural forms that model the different hyperlinks: independent links (ilink), property links (plink), contextual links (clink), aggregate location links (agglink), and span links (spanlink). Basically, these differ in the number of link ends used, the kind of elements linked, and the purpose of the links. Hyperlinks connect data items known as anchors (link ends). They record relations between data objects and form the basis of hypermedia navigation. Cross references which provide an optional path by indicating potentially useful information elsewhere are perhaps the most common kind of hyperlink.

Measurement module: This module uses the concept of a finite coordinate space (FCS) to support the definition and expression of units of measurement in time, space, and other domains. This is needed to describe the locations and dimensions of HyTime objects. In HyTime, a measurement is associated with an axis. The coordinates along an axis form its addressable range, from a minimum to a maximum positive integer value.

Scheduling module: This module provides facilities to specify and schedule events in both space and time. A component of data can be declared as an event and then placed into space and time (using a FCS) relative to other components. This can be used to represent the spatial and temporal relationships of data in multimedia presentations. With the help of the scheduling module, sequences and combinations of various media such as text, images, audio, and video can be coordinated.

Rendition module: This module allows one to describe certain presentational aspects of HyTime documents. It is, for example, possible to specify how extents within one event schedule can be mapped into another event schedule by defining an application-specific rule.

Chapter 3

System Architecture

3.1 Conceptual Multimedia DBMS Architecture

Our multimedia DBMS is an extension of the commercially available object-oriented DBMS ObjectStore. Since ObjectStore does not provide native support for multimedia applications and because of the unavailability of the ObjectStore code, a multimedia layer was built on top of ObjectStore to support multimedia applications. In the future, TIGUKAT [OPS$^+$95], an extensible object-oriented DBMS with inherent multimedia support, currently under development at the Laboratory for Database Systems Research of the University of Alberta, will replace ObjectStore. That is, the type system that resulted from this research will be incorporated into the TIGUKAT type system to directly support multimedia applications.

Figure 3.1 shows the conceptual multimedia DBMS architecture. Users and applications can access the multimedia database via a visual query interface[1] and an application independent API. The generic type system, which provides support for multimedia applications by defining the basic multimedia types, constitutes the main part of the multimedia extension. The design of this generic type system is the main contribution of this thesis. Future work is focussed on the development of a query pro-

[1]Currently, the visual query interface is implemented based only on the news-on-demand application. In the future, it will be generalized.

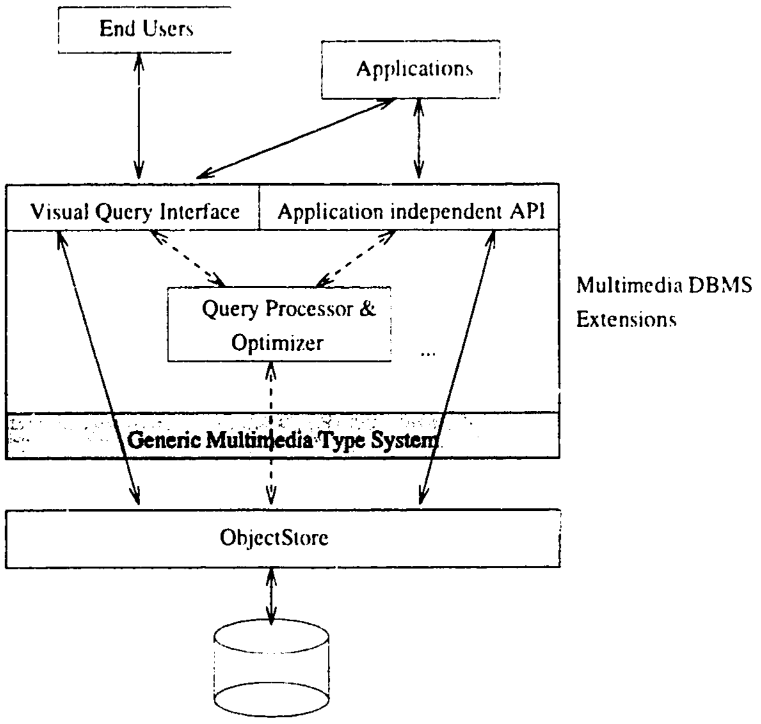

Figure 3.1: Conceptual Multimedia DBMS Architecture

cessor and optimizer that handle a distributed objectbase and support content-based queries of images and videos.[2]

The multimedia system is distributed over a broadband network using a multiple client/ multiple server architecture. In the prototype, IBM RS6000 machines, interconnected via an ATM network, are functioning as clients and servers.

3.2 Architecture for Handling Multiple DTDs

The main focus of this thesis is to develop a generic type system that is able to support different multimedia applications. This means, in the SGML context, that the database system must be able to reflect multiple DTDs. To add support for a new DTD to the system, the system analyses the DTD and automatically generates the types that correspond to the elements it defines.

[2]The dashed lines in Figure 3.1 indicate that these system components are not yet developed.

Figure 3.2 shows the system components that have been developed for handling multiple DTDs. These are the *DTD Parser*, the *Type Generator*, and the *DTD Manager*. A short description of the functionality of these system components is given here. For more details refer to Chapters 6, 7, and 8.

DTD Parser

The DTD Parser parses the DTD according to a meta-DTD that is a grammar for defining DTDs. Parsing the DTD creates an element list where each entry represents an SGML element defined in the DTD. Each entry contains information about the element, such as its name, its attributes, and its content model. After a successful validation of the DTD, the DTD Parser calls the Type Generator.

Type Generator

The Type Generator uses the element list created during the parsing process to automatically generate C++ code that defines a new ObjectStore type for each element in the DTD.[3] In addition, a meta-type is created for each new type.[4] These meta types contain information that is used by the Instance Generator (see Figure 3.3) to automatically insert documents into the database.

After the type generation is done, the generated code is compiled and the database schema is updated to reflect the new types. Furthermore, type extents are created and initialized for each new type.

DTD Manager

The DTD Manager takes a DTD file as input and stores the DTD as an object in the database. The DTD object is used for parsing documents and inserting them into the database. The DTD Manager is invoked after type creation. As soon as a

[3]The generated code is specific to ObjectStore. It has to be mapped to the TIGUKAT model when TIGUKAT is used as the underlying database system.

[4]This is part of the work done in [EM96]. See Section 7.5 for a detailed interface description.

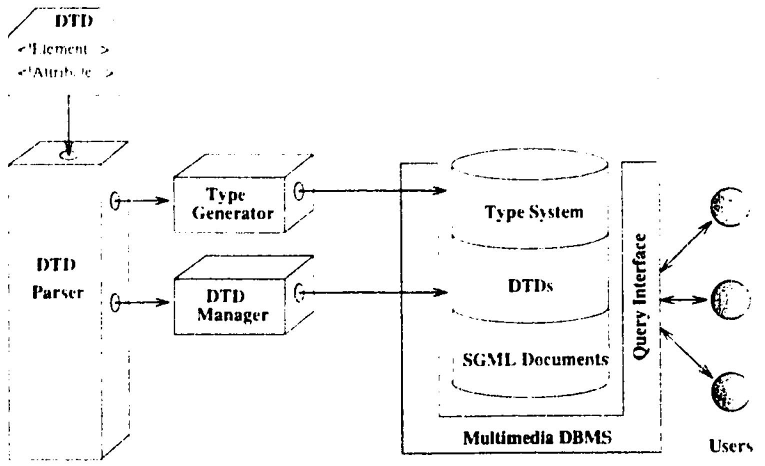

Figure 3.2: Architecture for Handling Multiple DTDs

DTD is stored in the database. SGML documents confirming to that DTD can be inserted into the database. A DTD object contains the name of the DTD, the DTD as a character string, and a type object for each meta-type defined for this particular DTD.

3.3 Architecture for Automatic Document Entry

Figure 3.3 shows the system components that were developed to support the automatic insertion of documents into the database. These system components are the *SGML Parser*, and the *Instance Generator*. These components depend heavily on the success of this thesis research in supporting a general multimedia type system. They have been developed as part of the project described in [EM96].

SGML Parser

The SGML Parser parses an SGML document, created by an authoring tool or by hand, according to the DTD to which it conforms. The parser retrieves the DTD from the database, where it was placed by the DTD Manager. Thus, if there is no DTD stored in the database, the document insertion process fails and a missing DTD is reported. During document validation, the parser builds a parse tree. There will

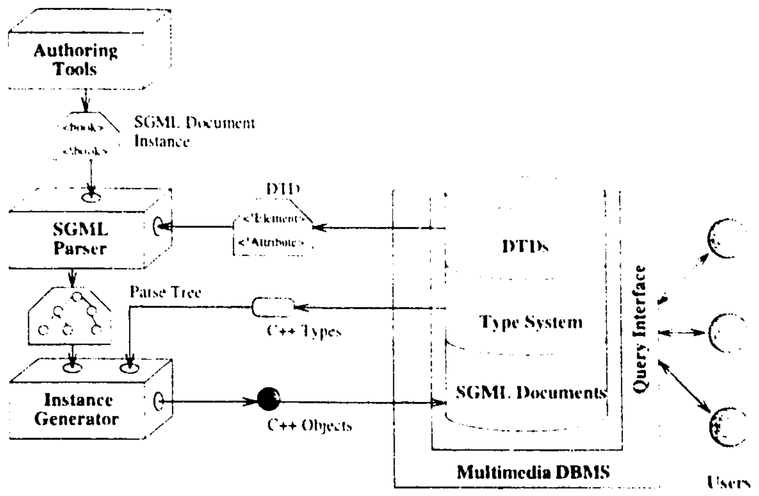

Figure 3.3: Architecture for Automatic Document Entry

be a tree node for each element defined in the document, containing information such as element name, defined attributes, parent element, and child elements. This information is necessary to instantiate the elements as objects in the database.

Instance Generator

The Instance Generator traverses the parse tree and instantiates the objects in the database that correspond to the elements in the document. Each node in the parse tree becomes a persistent object in the database. To create these objects, the Instance Generator uses the meta-type objects, stored in the DTD object, since they contain all the information necessary to create an object of a particular type.

Chapter 4

Dynamic Type Creation

As stated in Chapter 3, dynamic schema changes, more specifically dynamic addition of types to the database schema, are necessary to support multiple DTDs in the database. This issue has been studied extensively within the context of schema change/evolution in object-oriented database systems. In this chapter, various approaches to schema change/evolution are reviewed, before the approach followed in this thesis is described.

4.1 Varieties of Schema Evolution

The different kinds of schema modification can be divided into the following broad categories [PO95]:

- adding and dropping types (classes[1]),
- adding and dropping type properties (type redefinition), and
- adding and dropping sub/supertype relationships.

Schema evolution refers to changes in the database schema during the existence of the database. *Dynamic schema evolution* is the management of schema changes while

[1]The term "type" is used throughout this thesis for consistency. However, the term "class" is used in [PO95] and is also used in C++.

the database system is in operation [PO95].

In most cases, schema changes have a large impact on the database system, in particular, when *change propagation* is necessary. Change propagation refers to the need for propagating schema changes to the objects. This is necessary when the database schema is changed in a way that existing instances are somehow influenced. Instance transformation, that is the modification/migration of the affected instances, must then be performed to make the objects consistent with the new schema. Which schema changes require change propagation?

Type creation: Adding a leaf type to the database schema without changing the sub/supertype relationships of all existing types does not, by itself, require change propagation, since the new type cannot have any existing instances and existing instances of other types are not affected by the change.

Type deletion: When a type is deleted, all instances of that type must be deleted too. In most cases, change propagation is necessary, since all objects in the database that refer to these instances must be updated (references must be deleted).

Type redefinition: Type redefinition includes changing, adding, and deleting data members (properties) and member functions. In general, type redefinition requires instance transformation of all instances if the type representation in the database changes. This is almost always the case. The only exceptions are, when the implementation of a member function changes[2], or non-virtual member functions are added or deleted.

Changes in the sub/supertype relationships: Changes in the sub/supertype relationships include, for example, adding types as non-leaf nodes in the inheritance hierarchy, deleting types that are supertypes of other in the database schema's remaining types, or changing type inheritance from virtual to non-virtual and

[2]Note that, the implementation of a virtual member function can change but not the function header!

vice versa. In all these cases, the representation of affected instances must be changed. (For example, when a type is added as a supertype of an existing type, the definition of the existing type must be changed to specify the new inheritance relationships. The representation of the instances of that type must be changed to contain the data members and virtual member functions defined in the supertype.)

To summarize, the simplest form of schema modification is the creation of a type that does not change the inheritance relationships of existing types. Other schema changes are generally more complex and require change propagation.

4.2 Strategies for Schema Evolution

Different commercially available OODBMS or research prototypes can be distinguished in the way they handle schema changes and instance transformation. The four schema evolutions strategies used most are: *eager evolution*, *versioning*, *screening*, and *lazy evolution* [TK89].

Eager Evolution

Instance conversion is done immediately after schema evolution. An object of an existing type can be transformed into a *new* object of the modified type using a conversion function. Eager instance conversion has the advantage that it keeps the database in a consistent state. Instances can be accessed directly once they are converted. However, instance conversion may take a long time. During this time, these objects are not accessible. In addition, application programs that use these types must be recompiled after the schema change and all applications using the database must be relinked. Examples of OODBMSs using eager evolution are GemStone [PS87], and ObjectStore [Obj94].

Versioning

This strategy uses the concept of versioning for schema evolution. Each time a type is modified, a new version of this type is created. A type has a version chain and each instance of the type is associated with a specific version according to its creation time. Each type has a general interface (version set interface) which unites the definitions of all its versions. Application programs are written using this general interface. This strategy has the advantage that no time consuming instance conversion is necessary, and the disadvantage that the overhead caused by versioning and error handling creates performance problems when accessing objects. Versioning is used, for example, in the Encore system [SZ86].

Screening

Instance conversion is not done physically but logically. After a schema change, affected instances are not converted. Whenever they are used, they are interpreted in the new definition. This strategy avoids database reorganization or system shutdown, but on the other hand, the object access times are very long. Screening must be done each time, an object is accessed. One representative system that uses screening is ORION [BKKK87].

Lazy Evolution

In this schema evolution strategy, instance conversion is done physically but is delayed until the instances of the changed type are accessed the *first* time. Thus, there are performance problems only when accessing an instance the first time after a schema change. This is in contrast to screening, where the instance must be newly interpreted each time it is used. In comparison to eager evolution, the availability of instances is increased, since instances are manipulated only when needed. Instances which have not been used between consecutive schema changes are not converted at all. Lazy evolution is used in the object-oriented database system OBJM [TK89].

4.2.1 Evaluation of the Strategies

All strategies have advantages and disadvantages. If direct conversion is used, much time will be consumed at the time a type is modified. The database becomes essentially inaccessible during schema evolution and while all application programs are relinked. On the other hand, the database is always in a consistent state and the instances can be accessed directly once they are transformed, which is not the case if we delay instance conversion. If we delay instance conversion or use versions, we can always access the database, but we pay with slower access times for instances and management overhead (e.g. for keeping versions or change histories).

It is very difficult to say which of the strategies is the best one. This surely depends on the applications that use the database and the impact of the change. If we can live with the inaccessibility of the database from time to time, the direct conversion strategies are probably preferable, otherwise one of the other strategies would be better. In this context, hybrid strategies, combining two or more of the above methods, might be a good idea. For example, direct conversion could be used if the system is idle, switching to e.g. screening otherwise.

4.3 ObjectStore's Approach to Schema Evolution

ObjectStore provides the function `evolve()` to perform schema evolution. This function is part of the ObjectStore class library. It allows the modification of a database schema in many different ways, e.g. through type redefinition. Calling the schema evolution function `evolve()` triggers the schema modification, and changes the representation of all existing instances in the database to conform to the new type definition.

Without using `evolve()`, the database schema can only be changed by adding new types to it that are leaf-nodes in the inheritance hierarchy or independent of other types, or by changing the definition of a type already stored in the database in ways that do not affect the layout (representation) of the type instances [Obj94].

Adding a non-virtual member function, for example, does not change this layout, but adding a new supertype to an existing type does. The addition of new types is handled automatically when an application using the new type opens the database.

Schema evolution, performed using `evolve()`, depends on three parameters [Obj94]:

- the database(s) to evolve,
- the schema modifications, and
- the work database.

By default, the schema modifications are specified by the application calling `evolve()`.[3] The database and the work database are both specified as arguments to `evolve()`. The work database is used internally as a "scratch pad" to hold the intermediate results of the evolution process. In case of an error, this database can be used to restart the evolution or resume it from the point of interruption.

The modification of existing instances is performed in two phases, *instance initialization*, and *instance transformation*.

Instance initialization: Instance initialization modifies existing instances to make them conform to the new type definition. This includes, for example, adding or deleting data members or supertypes, and changing the type of a data member. If new data members are added or their types have changed, their storage regions are initialized.

Instance transformation: Most of the time, instance initialization is sufficient to perform the desired schema changes. Sometimes, other application dependent modifications are necessary. ObjectStore provides the *transformer functions* to perform this task. These functions are designed by the application programmer who initiates the schema evolution process. For each type to be modified,

[3]Thus, the schema source file for this application should contain a new type definition for each type to be modified.

exactly one transformer function can be defined. For example, changing the type of a data member from **short** to **long** and migrating the instances is done by the instance initialization phase, since the transformation rules are known to the system, but changing the type from **short** to a user defined type must be done by using a transformer function.

Note that, when modifying an instance, a *new* object is created as a copy of the old one. Changes are then performed on this copy. Thus, the address of an instance usually changes when the instance is modified.

4.4 An Approach to Dynamic Type Creation

Tests have shown that the schema evolution facility takes a very long time to run. Even for small examples, it took several seconds to modify all instances to make them conform to a new schema. Whenever **evolve()** is called on a database, *all* applications accessing this database, even the ones not using the newly created or redefined types, must be relinked. So, schema changes are not transparent to the user.

On the other hand, if the only schema change is to add leaf-node types, then the ObjectStore schema evolution facility (**evolve()**) is not needed. These types are added automatically to the database schema, whenever an application using these new types opens the database. Other applications will be unaffected by schema change, as long as they do not want to create or access instances of these new types. Applications that require the new types must be relinked.

For this project, this simple approach to dynamic type creation was chosen. This approach does not modify the inheritance relationships of the existing types, so **evolve()** is not used. The database is always accessible to applications and the applications are as independent as possible from schema modifications. In practice, database accessibility and application program independence of schema changes are very important issues for the acceptance of database systems. Users want to be able to access their data independent of changes by other users. The main disadvantage of the approach followed in this thesis is that we do not extract all the features that are common to different types and promote them to supertypes in order to achieve as much abstraction as possible. Thus, our system suffers from some code redundancy.

Chapter 5

Design of the Generic Multimedia Type System

As discussed in the previous chapter, the dynamic changes of the database schema are an interesting but difficult and complex problem. When we design a generic database system that supports any number of document types (DTDs), we must address exactly this problem. Basically, there are two different possibilities for a database system to support any conceivable document class:

- the use of a database schema based on a universal data model used for all DTDs, or
- the use of a kernel database schema that can be extended to add support for any DTD when needed.

Using a universal data model means developing the database schema in such a generic way that it can store documents of any document class we can imagine. To design such a database schema is very difficult, if not impossible, since the structure of documents conforming to different document types can be very different. In addition, if we could find a universal data model that allows us to store documents of any kind, it would probably be too inefficient. (For example, each child element of a DTD element would have to be modeled using an abstract type (such as type ELEMENT) rather than a more specific type. This results in problems during type checking.) Thus, this approach was not used for the design of the generic multimedia type system.

5.1 Design Requirements and Approaches

The goal of this thesis is to design an extensible database schema to allow for different document types. Within the SGML context, this means that the database schema is able to reflect different document types (DTDs). For object-oriented database systems, the database schema design is closely related to the design of the type system. When designing a database schema for a multimedia database system, some fundamental issues must be considered [Vit95, OSI+95].

Modeling of the primitive objects: A multimedia document consists not only of text but also of other media components such as images, videos, and audio tracks. A multimedia database system must be able to model and store these media.

Modeling of variants of the primitive objects: Primitive objects should be allowed to have multiple variants. Closely related to this is the modeling of quality-of-service parameters for these variants. Since the concept of variants is very important for the design of the multimedia system, it will be discussed in more detail in Section 5.1.1.

Representation of the document structure: SGML documents are structured. The database schema should represent the logical document structure for query and presentational purposes.

Representation of spatial and temporal relationships: Spatial and temporal relationships between multimedia objects must be captured to enable the display of multimedia documents.

All these requirements have to be incorporated in the design of the generic multimedia type system.

In the first phase of the CITR broadband project, the news-on-demand type system, a specific type system implementation for the news-on-demand application based on the News-DTD, was developed as a prototype. Some of the requirements mentioned above were already incorporated in this design [Vit95]. Therefore, it was used as the basis of the design described in this thesis.

5.1.1 Variants

Primitive objects are associated with specific quality-of-service parameters that are needed for presentation purposes. For example, the quality-of-service parameters of an image can be the image format (FIG, GIF or TIF), the image size, and the image colour (monochrome or 8bit/16bit/32bit colour). For one *logical* primitive object there can be a number of concrete primitive objects that can be distinguished only in their quality-of-service parameters. These concrete primitive objects are called *variants* of a logical primitive object. At the time of the quality-of-service negotiation, depending on the hardware available and the desired quality and cost, different variants of the same primitive object can be displayed, e.g. a monochrome or colour image.

Figure 5.1 shows a primitive object with different quality-of-service levels represented by different variants. Continuous media objects such as audio and video consist of a number of data streams. In Figure 5.1, `Variant 1` consists of `Data Stream 1`. `Variant 2` shares `Data Stream 1` with `Variant 1`, but also contains `Data Stream 2`. In general, the presentation quality is higher if more data streams are used. Therefore, the quality-of-service level of `Variant 2` is higher than of `Variant 1`. In the example, `Variant 3` and `Variant n` consist of the same streams but differentiate in other quality-of-service parameters such as audio/video format.

5.1.2 Generic Type System Design

In the multimedia database system, we store non-continuous media such as images and text as native objects and use a continuous media file server as the underlying storage system to store continuous media such as audio and video. This is a hybrid

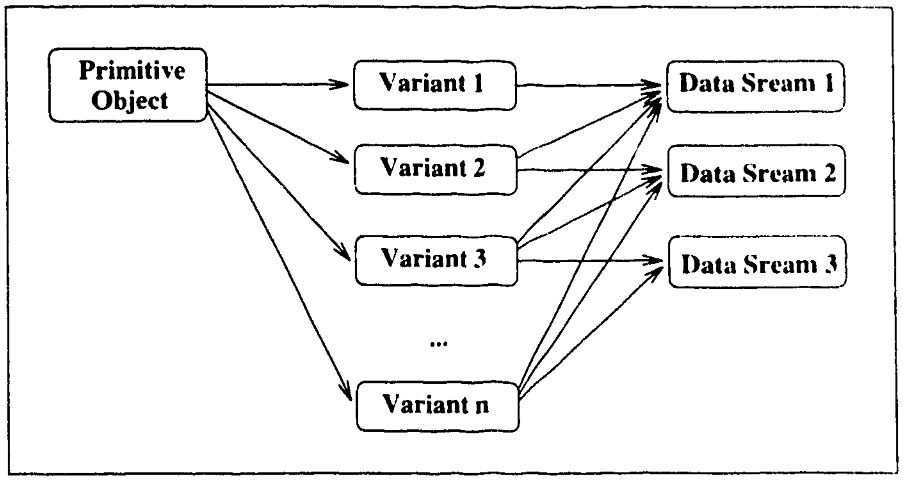

Figure 5.1: Variants of Primitive Objects

approach between non-integrated and integrated multimedia database systems. Non integrated systems store only meta information about the multimedia objects in the database but store the multimedia objects themselves in files. This keeps the database small and easy to manage but it does not allow the use of basic DBMS services such as transaction management, access control, and concurrency control for the management of the multimedia objects. On the other hand, integrated database systems store multimedia objects together with the meta information in the database. This approach has many advantages. Besides the advantage of using the DBMS services, it is possible to develop a uniform interface for accessing information about document content *and* document structure.

The modeling of the ***primitive objects*** (e.g. images) is done as a layer on top of ObjectStore, the object-oriented database system used for this project. Since ObjectStore does not provide native support for multimedia data other than text[1], the type system defines basic multimedia types and refers to them as *atomic types*. These types constitute the **atomic type system**. In our design, primitive objects consist of one or more *variants* that have *quality-of-service parameters* associated with them. Section 5.3 describes the design of the atomic type system in detail.

[1]Text could be stored as character string (`char*`).

The modeling of the *structure* of SGML documents and of the *spatial and temporal relationships* between multimedia objects is done in the **element type system**, another part of the generic multimedia type system. The reason for separating the atomic and the element type systems is discussed in Section 5.4.6. The element type system is a uniform representation of the elements in the DTD. Each element in the DTD corresponds to a type in the element type system. The spatial and temporal relationships between the elements in the DTD are modeled by types representing the architectural forms defined in the HyTime standard. Section 5.4 describes the design of the element type system in detail.

To achieve our objective, that is a generic and extensible type system, the following design approach is taken. The core of the multimedia type system consists of the atomic type system and some basic types of the element type system that model characteristics that are common to some or all elements of SGML documents. Whenever support for a DTD is added to the database system, types modeling the DTD elements are automatically generated and added to the element type system.

5.2 Flat Type System vs. Structured Type System

One major decision during the design phase of the generic type system is whether a flat or a structured type system should be used. Supporting any number of different DTDs in the multimedia database involves extending the database schema at runtime. This means that adding support for a DTD involves creating new types that represent elements in this DTD. The necessity of dynamic type creation adds a new dimension to the object-oriented design process and to the search for an "optimal" type system.

We now list the advantages and disadvantages of flat and structured type systems and give the reasons for choosing a structured type system.

Flat type system: Using a flat type system basically means creating a forest of types unrelated by inheritance. There might still be non-inheritance relationships between the types, such as containment or aggregation.

Advantages: The implementation of dynamic type generation is straightforward because all types are generated as root types. (They do not have supertypes.) The lack of a type hierarchy makes it easy to add and delete types from the type system. Costly schema evolution and instance conversion is not necessary.

Disadvantages: Inheritance is not used to reveal common behavior and reduce code redundancy. Therefore, we do not take advantage of the full power of object-oriented systems such as ObjectStore.

Structured type system: A structured type system is characterized by the existence of an inheritance type hierarchy.

Advantages: The type system is logically structured. This allows us to directly map the logical document structure to the type system in an effective way taking advantage of inheritance (e.g. a HyTime element in the DTD becomes a HyTime type in the type system). Using a type hierarchy allows us to reuse common data definitions and behavior for similar types.

Disadvantages: Dynamic type creation is a difficult task. Information (e.g. characteristics of elements) has to be obtained from the DTD to generate the new types as part of the type hierarchy.

Even though a structured type system makes dynamic type creation much more complex, it is essential when designing the type system for a multimedia database if the rich inheritance structure is to be exploited. Since the overhead for the dynamic type creation is still manageable, this approach is followed in this thesis. Overhead is, for example, the gathering of information about element characteristics to automatically detect the element's supertypes.

5.3 The Atomic Type System

The atomic type system models primitive objects that are the basic components of a multimedia document. The primitive objects contain the raw media representation (e.g. image representation) together with information for synchronization and quality-of-service negotiation.

The atomic type system described here is based on the design of the atomic type system developed in [Vit95]. Some new types have been added to the system and the existing types have been modified to meet the new demands of the synchronization and the QoS management component of the project. The type hi· archy is shown in Figure 5.2.[2]

All atomic types are defined as subtypes of the abstract supertype **Atomic**, which is the root type in the atomic type system. **Atomic** has one data member, the logical identifier `id`, to store the element identifier of an element that is part of an SGML document. In an SGML document, each element referring to an atomic type must have an SGML `ID` associated with it. SGML requires that this `ID` is unique throughout the document.[3]

The three types, **DataStream**, **Monomedia**, and **Variant**, are derived directly from **Atomic**. They will be di cussed in the following sections.

5.3.1 DataStream

DataStream is an atomic type for streams. A stream identifies a particular file on the continuous media file server. One particular audio or video can consist of a number of streams. The combination of several streams generates a quality-of-service level. In addition to the `id`, inherited from **Atomic**, **DataStream** has two other data members, `size` and `uoi`. `size` specifies the size of the file on the continuous media file server,

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

[3]In general, access functions (such as `GetId` and `SetID` for type **Atomic**) are defined for all atomic types to access their data members but we do not consider them here in the description of the type system.

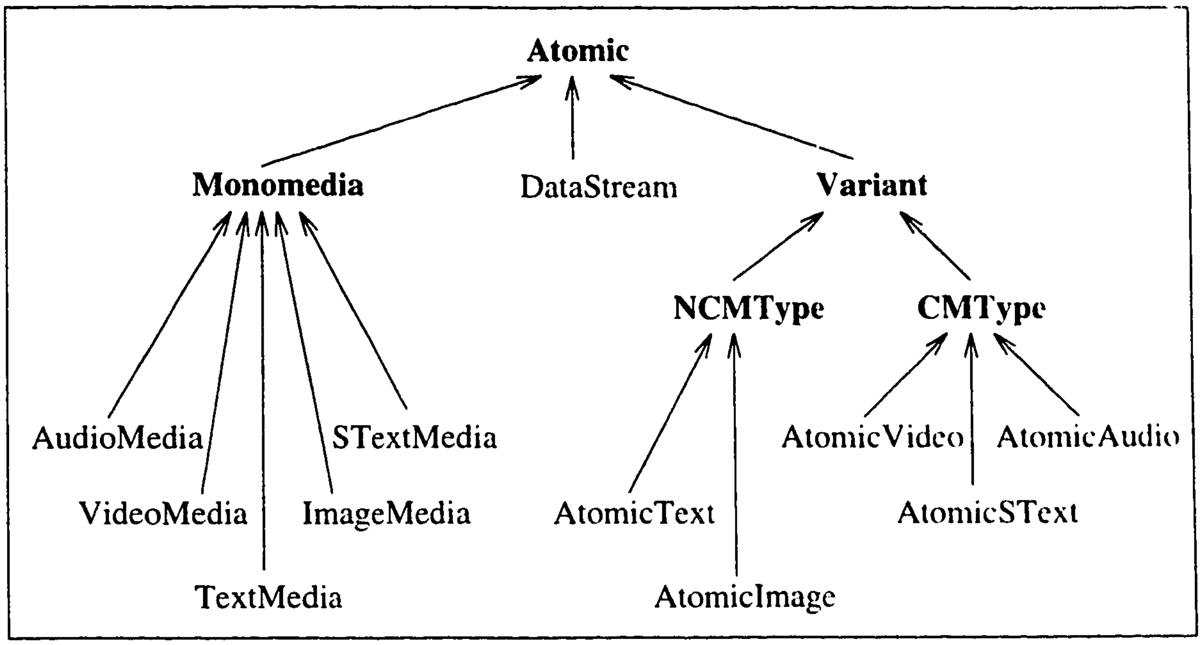

Figure 5.2: Atomic Type System

and the uoi (universal object identifier) is a unique identifier to identify the file on the server. Note the difference between the logical identifier id and the physical identifier uoi.

5.3.2 Variant Types

Instances of **Variant** subtypes hold the raw (mono) media representation, the quality-of-service information and information needed by the synchronization component. **Variant** subtypes and **DataStreams** define the smallest "units" in the atomic type system.

Type **Variant** contains functions to access quality-of-service parameters that are common to all variants. These are, for example, functions to access the size and the format of a variant. See Section 5.3.4 for a detailed discussion of QoS specifications.

There are two abstract subtypes of **Variant**, **NCMType** for non-continuous media such as text and images, and **CMType** for continuous media such as audio and video. This distinction has been made since non-continuous and continuous media are handled differently in the system. The difference between the two types is that instances of type NCMType store the raw media in their objects, whereas instances of type CMType have only meta-information stored about the continuous media. The actual data is stored on the continuous media file server.

To store the raw data, NCMType has a data member called `content`, which is an array of bytes. NCMType is subtyped into the concrete types AtomicText and AtomicImage. These two types also contain quality-of-service parameters (`textQoS` and `imageQoS`, respectively), as well as functions to access them (see Section 5.3.4). In addition to the access methods for the QoS information, AtomicText contains the two methods `Match` and `Substring` that are very important for the development of a query engine for the multimedia database. `Match` is a pattern matching algorithm that checks if the given search string is contained in the text content. `Substring` returns the part of the text content that is specified by a given start and end position.

CMType, the supertype for continuous media, is subtyped into the three concrete types AtomicVideo, AtomicAudio, and AtomicSText. As with the non-continuous media types, AtomicVideo, AtomicAudio, and AtomicSText each contain quality-of-service parameters and access functions that are associated with them.

Since continuous media are stored on a separate continuous media file server, CMType does *not* define a `content` data member. Instead, it defines attributes that specify the server on which the continuous media data is stored (`site`), and the data streams (`DataStream*`) that make up the continuous medium. Placing the location information in CMType assumes that all the streams for a particular variant will be stored on the same server. If this assumption is not valid, location information should reside with each data stream.

AtomicSText, the atomic type representing synchronized text, generates some interesting questions. Under HyTime, the synchronization information for any event is captured by an associated extent list relative to a finite coordinate space. This information is logically separate from the atomic objects, which do not know when and where they are displayed. The idea of representing all the synchronized text for a video as a single event with multiple variants having multiple streams is attractive in terms of aggregating. For example, all the English subtitles could be in one variant and all the French subtitles could be in another variant. However, this arrangement is contrary to the HyTime model since it would require data streams themselves to contain synchronization information. Even if we had only one event with a single variant, we could not have more than one stream associated with it, since we lack the information about how to synchronize them. Therefore, for the current implementation, an instance of AtomicSText consists of one data stream only and can represent only a single subtitle. To create and synchronize multiple subtitles, multiple events are necessary. This is not an optimal solution but it conforms to HyTime.

5.3.3 Monomedia Types

The concept of Monomedia types has been introduced to allow the primitive objects to have multiple variants. One logical object (e.g. the "Enterprise" video), referred to as a *monomedia object*, can consist of a number of variants that have different quality-of-service levels. For example, the video variants might have different frame rates or bit rates, or they might consist of a different number of streams, which may be located on different servers.

Type Monomedia contains QoS information (`monomediaQoS`) that is common to all of its subtypes, such as the `price` and the `type` of a monomedia object. Besides the access functions for the QoS information, type Monomedia defines functions to access the variants that make up a monomedia object.

The abstract supertype Monomedia has five concrete types derived from it. TextMedia, ImageMedia, AudioMedia, VideoMedia, and STextMedia. They correspond to the Variant types AtomicText, AtomicImage, AtomicAudio, AtomicVideo, and AtomicSText, respectively. Instances of these monomedia types store references to all of their variants. A monomedia object can have only variants of the same type associated with it, for example, a VideoMedia object can contain only AtomicVideo objects.

5.3.4 QoS Specification

As already mentioned, each type derived from **Atomic** has certain quality-of-service parameters associated with it that are used for the QoS negotiation. A QoS type hierarchy has been developed to model these parameters. Each atomic type has only one data member for the quality-of-service specification which is of the appropriate QoS type. For example, **AtomicAudio** has the data member `audioQoS` which is of type **TQosAudio**. Figure 5.3 shows the QoS types. This way, we achieve a clear separation between information stored about the content of an atomic object on one hand, and presentation information (synchronization and QoS information) on the other.

TQosVariant is an abstract supertype (root type) that contains the QoS information that is common to all variants. It has two data members, `format` and `size`. The `format` of a variant can be, for example, GIF or TIF for an image, or, MPEG or MJPEG for a video. `size` indicates the size of a variant; for non-continuous media it is the size of the object in bytes, for continuous media it is calculated as the sum of the sizes of the data streams belonging to the object.

TQosImage and **TQosText** are concrete QoS types directly derived from **TQosVariant**. **TQosImage**, the QoS type for **AtomicImage**, has two data members, `colour` and `spatialResolution`. `colour` specifies the colour of the image such as monochrome or super-colour, `spatialResolution` stores the width and the height of the image. **TQosVideo** is defined as a subtype of **TQosImage**. In addition to the inherited attributes `colour` and `spatialResolution`, **TQosVideo** contains the attributes `frameRate`, `bitRate`, and `duration`. These attributes are used for synchronization purposes. **TQosText**, the QoS type for **AtomicText**, has the data member `language` which specifies the language in which the text is stored (e.g. English, French, or German). **TQosSyncText** and **TQosAudio** are subtyped from **TQosText** to inherit `language`. Additionally, **TQosAudio** contains the four data members, `sampleRate`, `bitsPerSample`, `audioQuality`, and `duration`. The audio quality specifies, for example, CD quality or telephone quality. **TQosSyncText** does not contain additional attributes but has been defined to obtain a one-to-one correspondence of variant types

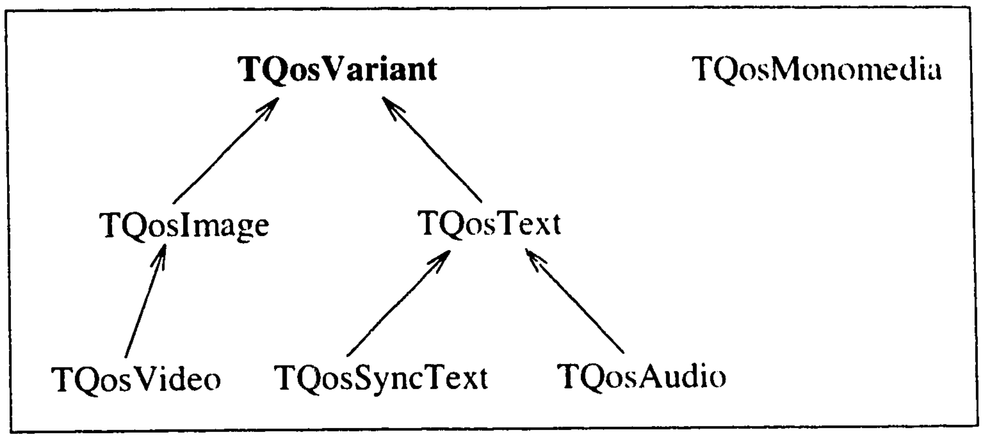

Figure 5.3: Quality-of-Service Types

and QoS types.

TQosMonomedia specifies the quality-of-service information for monomedia types. It has the two data members `type` and `price`. `price` is a static value that is attached to a monomedia object independently of its variants. It is used to calculate the total display cost of a monomedia object. The total display cost consists of a static part (price) and a dynamic part, which is calculated by the negotiation protocol. `type` specifies the type of the monomedia object, for example IMAGE or AUDIO. Strictly speaking, this is redundant since each monomedia object already contains implicit type information (e.g. an ImageMedia object is of "type" IMAGE). However, C++ does not maintain type information at run-time and this feature is required by some modules of the multimedia system developed at the University of Montreal.

5.4 The Element Type System

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

As previously stated, the approach that is followed in designing the type system is to introduce some built-in types that constitute the core of the type system. The built-

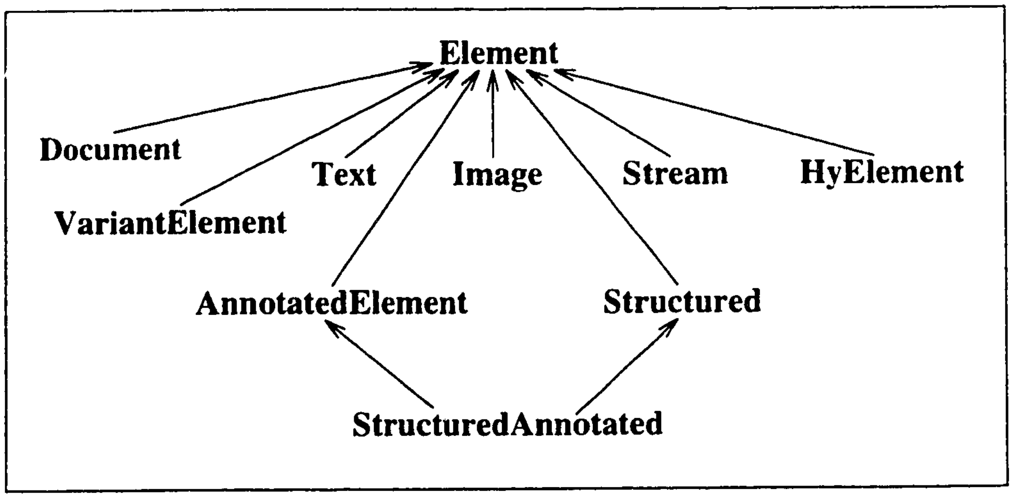

Figure 5.4: Built-in Element Types

in element types, some of which are adopted from the news-on-demand type system, model characteristics that are common to (some or) all DTD elements. Figures 5.4, 5.5, and 5.6 show the built-in types of the generic element type system. Whenever support for a new DTD is added to the system, new element types that model the characteristics specific to that DTD must be created and dynamically added to the type system. All element types that are automatically generated will be derived from one of the predefined abstract element types (built-in types).

5.4.1 The Root Element

Type Element is the supertype of all element types. It contains the data members **document**, **parent** and **type**, and member functions to access them. **document** points to the document the element belongs to. The document object has type Document to allow for different document types. See Section 5.4.7 for a detailed discussion of type Document. The attribute **document** has been introduced in type Element because it is useful for an element to know its document. For example, the query: "Display the titles of all documents that have images." could first retrieve all images stored in the database and then access the documents from these image elements.

Since SGML documents have a tree structure, each element, except the root element, has a parent. The data member **parent** models this structure by pointing to the element's parent element.[4] The type of **parent** is **Structured** (described in the next section), since all parent elements must be structured elements, that is, they have child elements.

For application programming, it is very useful for an object to know its type. To achieve that, the data member **type** has been introduced. It points to a type object that specifies the type of the element. Each element type has a type class (meta-type) associated with it (e.g. type **Article** has the meta-type **ArticleType**). Such a type object contains the name of the element as it appears in the DTD and other useful information that can be queried. (Refer to Section 7.5 for a discussion of meta-types.)

Type **Element** is subtyped into eight more specialized abstract types, **Structured**, **AnnotatedElement**, **HyElement**, **VariantElement**, **Text**, **Image**, **Stream**, and **Document**.

5.4.2 Structured Elements

Type **Structured** is a supertype for all elements in the DTD that are non-leaf nodes in the document tree. Such elements have a complex content model. By complex content model, we mean a content model that is not either **EMPTY**, or **(#PCDATA)**. Elements that are defined as **EMPTY** or **(#PCDATA)** are always leaf nodes in the document tree, and therefore unstructured.

Elements with a complex content model always have child elements. That is why structured elements must maintain references to their child elements. Type **Structured** contains a **childList** that keeps track of these references, and member functions to access the **childList** (such as **GetNth** to retrieve the n^{th} element in the **childList**). The **childList** contains objects of type **Element** because this is the common supertype of all possible child elements.

[4]In the root element, **parent** will be set to NULL.

5.4.3 Annotated Elements

For efficiency reasons, all textual components of a document are stored together as one text string. Each textual component has an *annotation* associated with it that indicates the start and end index of the object's text in the text string. To keep annotations is not only useful for elements that contain text (#PCDATA) but also for other elements that have to be located within the text string to display them properly. For example, an image should be displayed in the document at the position where it was defined and not separated from the document. The annotation start and end index for an image have the same value, the location in the text where the image should appear. Similarly, if we define a HyTime link, we want to be able to display the link in the document, for example by underlining the word or phrase that corresponds to the link. The annotation defines this text range. The only element types that do not have annotations are **Element**, **Structured**, all the HyTime types except the types derived from **Ilink_AF**, and all MM types except the types derived from **Image** and **Text**. These types do not have annotations because this information is not useful for any purpose.

Besides the data member `absoluteAnnotation` that stores the element's annotation, **AnnotatedElement** has the member functions `GetAbsoluteAnnotation` to query the element's annotation, and `GetString` to retrieve the appropriate substring representation of the element's annotation.

5.4.4 Structured Annotated Elements

Elements which are structured and have annotations are derived from the abstract supertype **StructuredAnnotated**. **StructuredAnnotated** is a subtype of both **Structured** and **AnnotatedElement** and it inherits `childList` and `absoluteAnnotation`, respectively. It also inherits all of the member functions that are defined in these two types.

5.4.5 HyTime Elements

Figure 5.5 shows the built-in HyTime types. **HyElement** is the supertype of all HyTime types in the element type system. Its immediate subtypes correspond to the architectural forms defined in HyTime. These subtypes are only derived from **HyElement**. The HyTime types are used to model spatio-temporal relationships between multimedia objects in order to synchronize their presentation.

Following the HyTime standard, all HyTime elements in the DTD must have `ID` and `HyTime` attributes. The `ID` is used as a unique identifier to make element references possible. The HyTime attributes specify the architectural form to which the element belongs as well as spatial, temporal, and location information. Some of the HyTime attributes are required, while some are implied.

HyElement contains the `id` required for each HyTime element as a data member. The HyTime types representing the HyTime architectural forms contain the static data member `aFormName`, set to the appropriate architectural form name. Furthermore, they have data members for all attributes that are required or implied by the HyTime standard for the particular architectural form. Appendix A describes all attributes that are defined for the built-in HyTime types.

Types **Video**, **SText**, and **Audio** are directly derived from the HyTime type **Event_AF**. They use HyTime events for synchronization purposes.

5.4.6 MM Elements

The **MM** types (monomedia types) have been introduced to provide a consistent way of handling DTD elements that refer to atomic objects.[5]

Each concrete type in the atomic type system has an MM type in the element type system associated with it.

If there is an element in the DTD that refers to an atomic type, the element type modeling this element will be subtyped from one of the MM types. Using this

[5] Any type derived from an MM type is also considered an MM type.

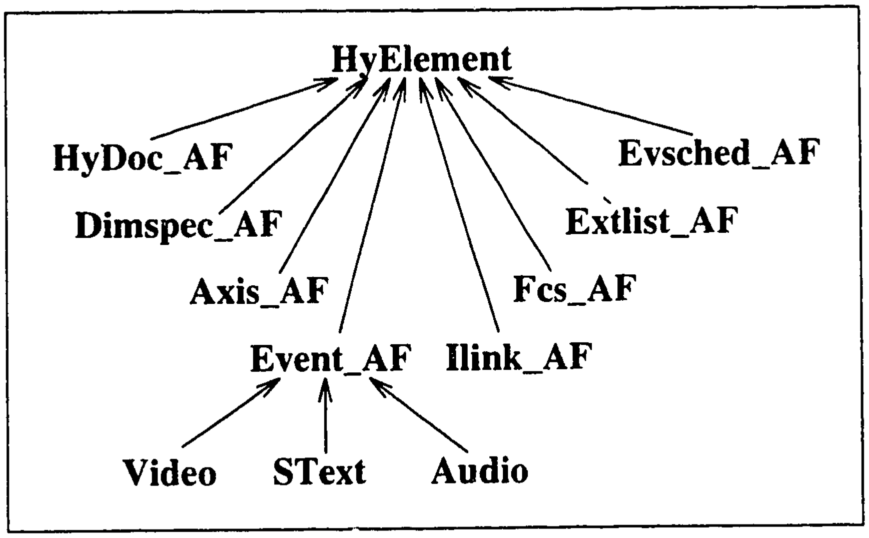

Figure 5.5: Built-in HyTime Types

concept, we can maintain the one-to-one correspondence between concrete element types and elements defined in the DTD. This indirection is used so that no element will be derived directly from an atomic type. Why is this useful?

- First of all, there is a clear interface between atomic types and element types. Atomic types model the primitive multimedia objects such as images or videos, whereas element types model elements in a DTD. Some of these elements just happen to be atomic, in the sense that they refer to atomic types. This *reference* is expressed using the MM element types.

- Second, this clear separation of atomic and element types simplifies the work of our project partners who deal, for example, with the quality-of-service negotiation or the storage of streams on the continuous media file server. They are concerned only with the atomic type system and not with the document structure. The document structure, on the other hand, is important for the display of the documents (e.g. links are underlined) and for the query engine (e.g. "retrieve all authors").

There is one drawback to this approach. The type system is more complex since the MM element types mirror the atomic types.

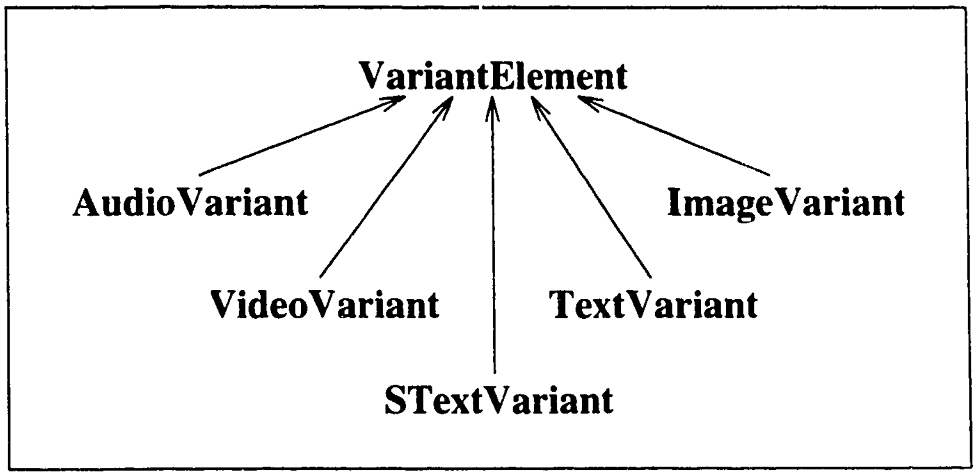

Figure 5.6: Built-in MM Variant Types

Each MM type contains a pointer to an object of the corresponding atomic type. For example, MM type ImageVariant maintains the data member `image` pointing to an object of type AtomicImage.

Figure 5.6 shows the predefined MM variant types. AudioVariant, VideoVariant, STextVariant, TextVariant, and ImageVariant are the MM types for AtomicAudio, AtomicVideo, AtomicSText, AtomicText, and AtomicImage, respectively. The types Text, Image, and Stream in Figure 5.4 are the MM types for the atomic types TextMedia, ImageMedia, and Stream. The attentive reader will notice that there are still three atomic types missing, AudioMedia, VideoMedia, and STextMedia. The MM types for these are Audio, Video, and SText. They are displayed as part of the HyTime type hierarchy in Figure 5.5. These three types are special in the sense that not only are they MM types modeling the reference to atomic types, but they are also HyTime types at the same time. The reason why they are defined as HyTime types is that they have spatial and temporal components that must be modeled using HyTime.

Figure 5.7 shows the relationships between the atomic types and the MM element types for the example of the Audio types. The relationships of the Text, Image, SText, and Video types are similar. In the example, we assume MyAudio, MyAudioVariant, and MyStream are user defined concrete element types and `aMyAudio`, `aMyAudioVariant`,

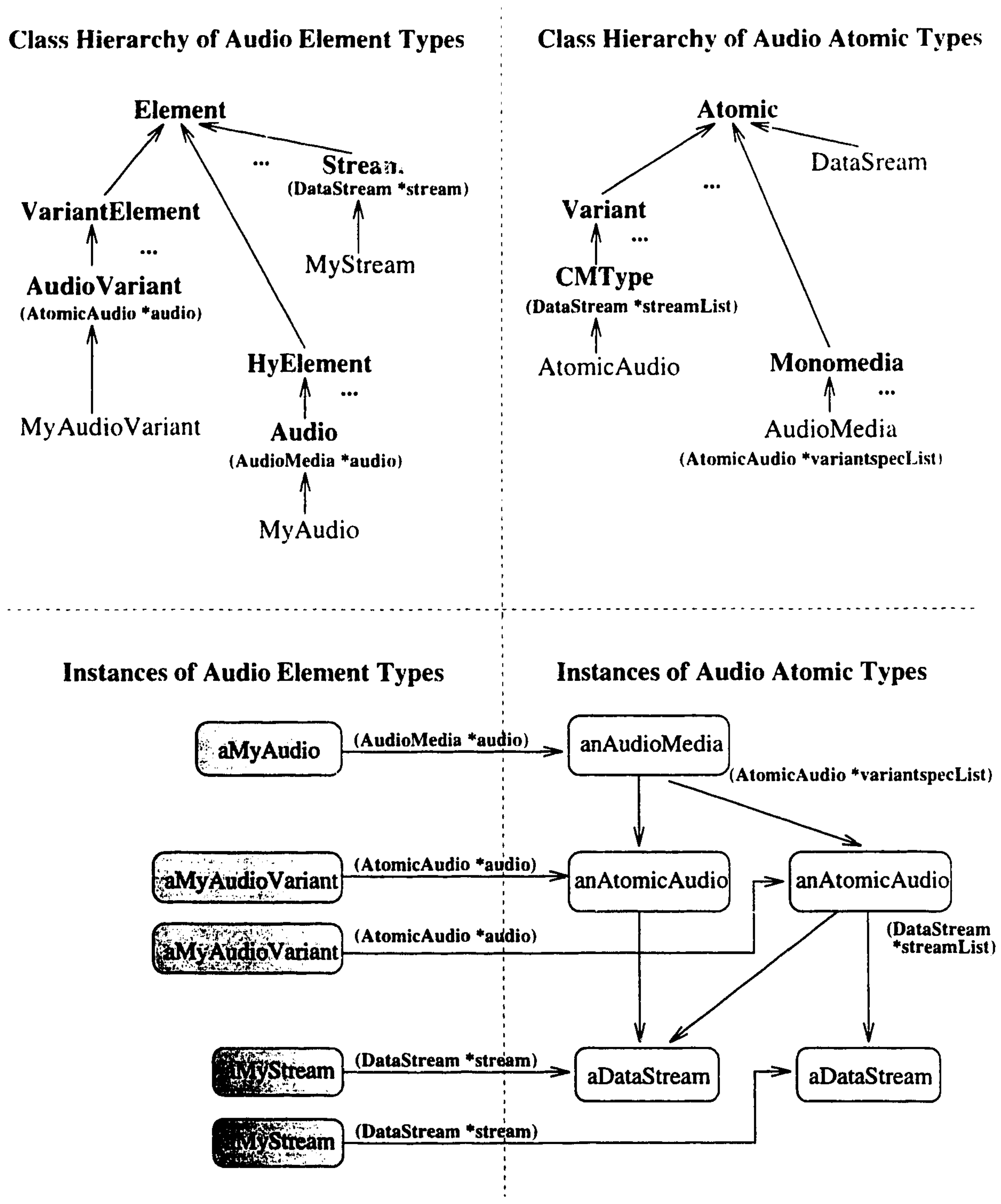

Figure 5.7: Example for MM Audio Types

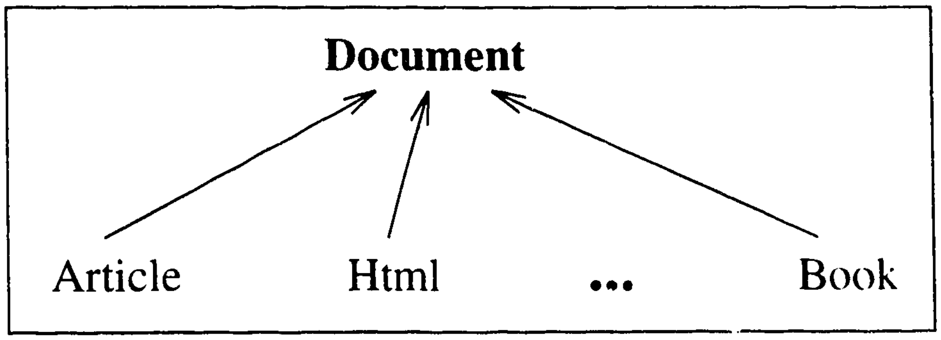

Figure 5.8: Type Document and its Subtypes

and aMyStream are instances of these types. anAudioMedia, anAtomicAudio, and aDataStream are instances of the atomic types AudioMedia, AtomicAudio, and DataStream, respectively.

5.4.7 Document Types

It was mentioned before that an object of type Element knows the document it belongs to. Its data member document points to an object of type Document. Document is a general built-in abstract supertype which models document types in the type system. Since Element is the supertype of all DTD elements, it must use types just as general for its data members.

Specific document types are derived from type Document. Figure 5.8 illustrates some sample document types. An instance of a concrete element type will always point to an object of a specific document type, such as Article or Html. For example, all elements which belong to an "article" document point to an object of type Article.

5.4.8 Helper Types

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

Type Annotation

As mentioned in Section 5.4.3, the entire text of a document is stored in a single object. Annotations are used to indicate where the composit[illegible]al elements start and end, relative to the beginning of the text string. All eleme[illegible]at are part of the text string (compositional elements) are annotated elements. E[illegible]tated element has an annotation associated with it. This annotation is of typ[illegible]ation. Type **Annotation** has the two data members `startPosition` and `endPos`. [illegible]ind[illegible] the start and end position of the element's text in the document's te..

Type DocumentRoot

Every document needs a place to store its text string. Thus, type **DocumentRoot** has been introduced. **DocumentRoot** contains a document's text string as an object of type **AtomicText**. Furthermore, it contains lists of all **Monomedia** objects (e.g. lists for **ImageMedia**, **AudioMedia** etc.) and member functions for accessing these lists. The lists of atomic media instances are stored here to improve access efficiency because atomic objects are queried quite frequently by other system components, e.g. for the QoS negotiation.

To achieve fast access to components of the document's text string, it is advantageous to store a document's list of annotations together with its text string. When displaying the document, a browser can scan these lists efficiently to determine the presentation of the text. Since annotations are specific to the elements defined in a particular DTD, the annotation lists cannot be part of the built-in type **Document-Root**. To achieve the desired effect, for each DTD supported by the system, a root type is created as a subtype of **DocumentRoot**. Figure 5.9 illustrates this idea. These root types inherit all data members and member functions from **DocumentRoot**, but also contain annotation lists for all annotated elements in the DTD.

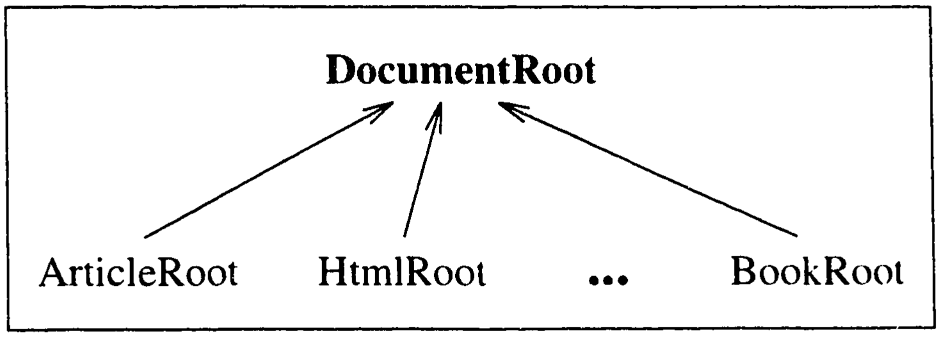

Figure 5.9: Type DocumentRoot and its Subtypes

Dtd

The concrete type **Dtd** has been introduced to store DTD objects in the database. An instance of **Dtd** contains the name of the DTD and a character string that represents the DTD text. The DTD must be kept in the system because it is not possible to recreate the original DTD from the type information. The functions `GetDtdName` and `GetDtdString` are defined to access the DTD name (document type) and the DTD string. Additionally, **Dtd** has an attribute called `typesList`, which contains a type object for each element type defined for that particular DTD. These type objects are needed for automatic insertion of document instances into the database.

Type **Dtd** can be extended at any time, if necessary, to hold more [illegible]mation by adding new data members to it. For example, a counter indicating the number of documents stored in the database that belong to this DTD might be a good idea, or a document list pointing to all these documents.

ElementType Types

ElementType types are used for the automatic insertion of documents into the database. They define functions to instantiate database objects that correspond to the document instance. They also store some meta-information about the DTD element types such as element names, attributes, and supertypes. This information is necessary for instantiating the appropriate database objects and setting their at-

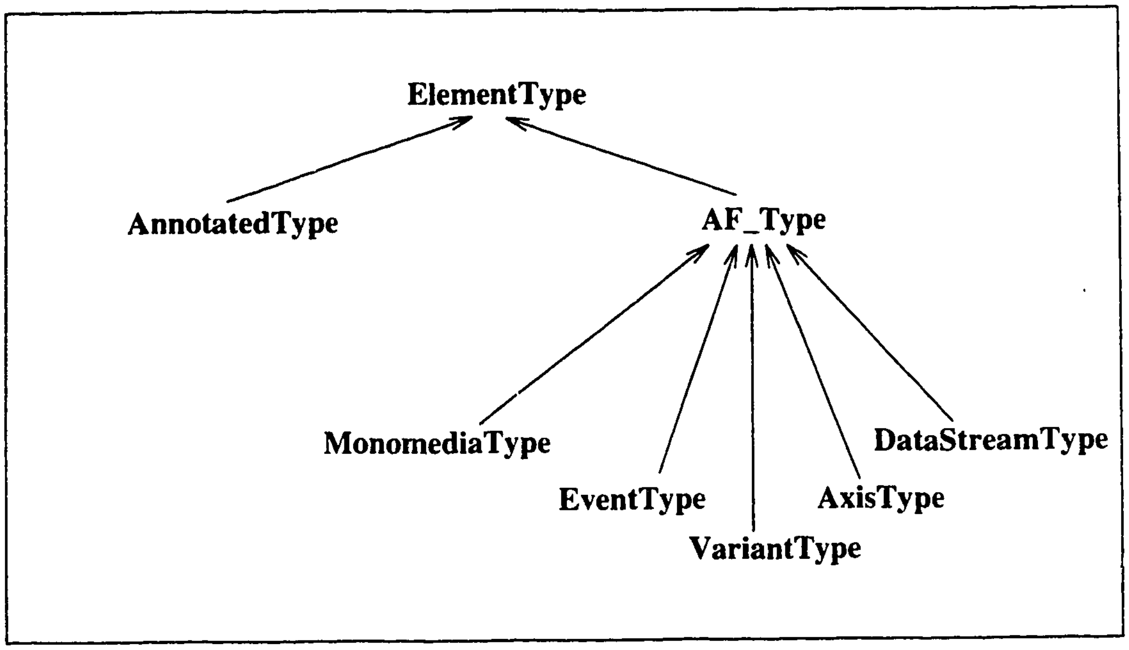

Figure 5.10: Top-level Hierarchy of the ElementType Types

tributes.

There are a number of built-in ElementTypes defined in our system. Figure 5.10 shows the top-level hierarchy of the ElementTypes. For each logical DTD element, there is an Element type and an ElementType type associated with it. The design of the built-in ElementTypes and the automatic generation of the ElementTypes defined for the specific DTD elements are discussed in detail in [EM96].

5.5 Extending the Element Type System

All the predefined types discussed above are generic types. None is specific to a particular DTD or application. They model characteristics and behavior that are common to all DTD elements.

In order to support different document types in the multimedia database system, we must extend the basic database schema with element types that model the specific characteristics of the elements defined in the particular DTDs to be supported.

5.5.1 What Types are Necessary to Support a New DTD?

In the element type system, there will be one element type defined for each element in the DTD. Following the approach described in Section 4.4, all DTD specific element types are added to the element type hierarchy as leaf nodes. They will be subtypes of one or more of the built-in element types.

For each DTD added to the system, a helper type is created that is a subtype of **DocumentRoot**. These subtypes keep track of element annotations and document text strings.

The element type that represents the root element of a DTD will be a subtype of type **Document**, and type **StructuredAnnotated**.[6]

Element types modeling the HyTime elements defined in the DTD are subtypes of the built-in HyTime types. In addition, if the HyTime elements have a complex content model, their element types are derived from either **StructuredAnnotated**, in case of the architectural forms HyDoc and Ilink, or **Structured**, otherwise.

Types representing the MM elements of a DTD will be subtypes of the predefined MM types. MM elements always have an **EMPTY** content model. Hence, these element types are not derived from the structured types. MM types, except types derived from **Image** and **Text**, will not have annotations associated with them either.

All other element types to be defined will be derived from **StructuredAnnotated**, if they have a complex content model, or from **AnnotatedElement** otherwise.

[6]This [illegible] that the DTD consists of more than one element and that the root element, in case it is a [illegible]ime element, is of the architectural form HyDoc.

Chapter 6

The DTD Parser

To support multiple DTDs in the database system, a tool is needed which analyses a DTD. This tool must perform three tasks: check the DTD for correctness, gather information that is needed for automatic type creation (e.g. the names of the elements defined in the DTD, their content model, and attributes), and invoke automatic type creation.

For the DTD Parser, a decision had to be made between:

- developing a new DTD Parser,
- using an existing DTD parser, or
- using an existing SGML document parser.

Since SGML is complex, it would be very difficult and time consuming to develop a new DTD Parser. Thus, the possibility of using an exiting DTD parser was explored. In spring 1995, when this project started, there was only one publicly available DTD parser, *dpp*. This parser was developed by C. M. Sperberg-McQueen at the University of Illinois. It can be found at the ftp site:

`ftp://ftp-tei.uic.edu/pub/tei/grammar/dpp/dpp.1`

The parser was written in C and used flex and bison (lex/yacc replacement). It fulfills the first requirement of our DTD Parser, to check the correctness of the DTD.

But the parser, in its original form, provided no facilities for gathering information about the DTD and for invoking the automatic type creation, our second and third requirements. Nevertheless, *dpp* was chosen as the basis for our DTD Parser, since the availability of the code made extensions and modifications to the parser possible.

Using an SGML document parser for parsing DTDs would have been a valid approach as well. It has been adopted in some projects reported in the literature, e.g. in [ABH94], and [CACS94]. But, since the SGML syntax for documents and DTDs are different, DTDs must be transformed into document syntax to be parsed by a document parser. This approach has extra overhead compared to the approach we have followed. In the near future, some newer versions of SGML document parsers will contain a full-fledged DTD parser. In particular, the new version of the *nsgmls* parser, the document parser used in our project, is supposed to contain a full-fledged DTD parser. Using this DTD parser instead of *dpp* would have the advantage of maintaining a uniform parser for both parsing the DTDs and the documents.

In the following sections, the necessary modifications to the *dpp* parser are described. Furthermore, the features of the parser and its limitations are discussed.

6.1 Modifications to the Original Version of dpp

As stated previously, the three important tasks of the DTD Parser are:

- checking the correctness of the DTD,
- gathering information for type generation, and
- invoking type generation.

The original version of *dpp* checks the correctness of the DTD by checking its conformance to a meta-DTD. This meta-DTD describes a grammar for defining DTDs.

`flex` divides the structured input, in our case the DTD, into "meaningful" units, called *tokens*. For example, in the context of SGML: `CDATA`, `PCDATA`, `<!`, and `ELEMENT` are tokens. `flex` uses a *flex specification*, that is a set of descriptions of possible

tokens, to produce the C function yylex(), called a *lexical analyzer* or *scanner*. The descriptions of the tokens are given in the form of regular expressions.

`bison` establishes relationships between the input tokens generated by yylex(). This task is known as *parsing*. The set of *rules* that define the relationships between the tokens is called a *grammar*. In the DTD Parser, this grammar specifies all definitions the DTD contains, and it can therefore be viewed as a meta-DTD. `bison` takes a concise description of this grammar and generates the C function yyparse(), which is known as the *parser*. The parser automatically detects whenever a sequence of input tokens matches one of the rules in the grammar. When the parser recognizes a rule, it executes user C code associated with the rule that is called the rule's *action*.

The original DTD Parser did not build data structures while validating the DTD. During the parsing of each definition in the DTD, strings were created that indicated which actions were performed. After a definition was successfully parsed, the string was displayed. To generate the information necessary for automatic type creation, *dpp* has been modified to build up data structures instead of strings. These data structures are described in Section 6.2.

Furthermore, changes have been made to the way the parser is called. The original parser took the DTD from the standard input. This is not sufficient anymore, since we need information (e.g. the name of the DTD file) for further processing. The modified parser is called in the following way:

```
dpp -f<dtd file> -d<database name> [-r<root element>]
```

where `-f` and `-d` are required command line arguments that indicate the filename of the file containing the DTD, and the database where the new types should be created. `-r` is an optional command line argument to indicate the root element of the DTD. The root element is not necessarily the natural root element of the DTD since it can be any element in the DTD.[1]

Other modifications to the parser are:

[1]This is valid SGML.

- Parsing no longer stops when the first error has been found. Rather, *all* error messages are printed *after* parsing the entire DTD.
- After *successfully* parsing the DTD, the Type Generator is invoked with information gathered during validation of the DTD.

6.2 Data Structures

dpp was modified to create data structures during DTD validation that store the information necessary for type generation. Exactly what information is needed? To create a type for an element defined in the DTD, the following information is required:

- the element's name, since it is used to name the new type (See Section 7.2.).
- the element's content model, since it specifies which data members must be created for the new type (See Section 7.3.),
- the attributes defined for the element (including attribute name, value type, and default value), since they will be modeled as data members as well (See Section 7.3.),
- the inclusions defined for the element, since they specify possible child elements of the element (See Section 6.3 for a discussion about inclusions.), and
- additional information required by the Type Generator e.g. the element's supertypes, and predefined attributes.

On the other hand, there is information generated by the parser which is not relevant for type generation (e.g. information about entity and comment declarations, data tag minimization). Thus, this information does *not* need to be stored.

Data structures have been developed that are capable of storing all necessary information based on these requirements. The main data structure is the element list. Whenever an element definition is found in the DTD, an entry in the element list is created. Each element list 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 declaration, **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. **incl** is a pointer to the element's list of included elements. **reachable**, **visited**, **mult_elem**, **isList**, **isStructured**, and **supertype** are initialized by the DTD Parser, but they are set and used only by the Type Generator (See Chapter 7.).

Whenever an attribute definition is found in the DTD, an entry is created in the attribute list of the element that contains the attribute. If the attribute definition appears in the DTD before its associated element is defined, an entry for the referenced element is created in the element list. The other element information is filled in later when the element definition is parsed. Each entry of the attribute list has the following structure:

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

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. `predef` is initialized by the DTD Parser, but set and used only by the Type Generator (See Section 7.3.1.). Appendix B contains an example illustrating how the element list is constructed by the parser.

6.3 Supported Features

The parser can handle all basic concepts of SGML including entities and complex content models, where a model group is inside of another model group such as `(a,(b|c|d)+,e)*`. Entity references are resolved automatically by the parser. Whenever an entity definition is parsed, it is stored in a table. If this entity is referenced somewhere else, *dpp* searches the table for the entity definition and replaces the entity reference with it.

dpp supports *inclusions* and *exclusions* which can be summarized as *exceptions*. In SGML, exceptions are shorthand methods for defining a content model that can avoid the need for complex model groups throughout an element's subelements. Elements named in an *inclusion* can occur *anywhere* in the element being defined and *anywhere* in its subelements. For example, suppose that a footnote (`fn`) is allowed anywhere within an `article` an unlimited number of times. The definition would be:

```
<!ELEMENT article   - -   (frontmatter, async, sync) +(fn)>
```

Exclusions (e.g. **-(fn)**) are defined to exclude elements from showing up in all the subelements of a given element. This is done to avoid unwanted recursions.

dpp, in its original version, checks only the syntax of element definitions. It is not capable of propagating the exceptions down to all elements for which they are defined. Since this feature is necessary for ccrrect type generation, it has been added to *dpp*. The propagation of the exceptions is done after the DTD is successfully parsed by iterating over the element list and inserting the information in the appropriate element entries.

6.4 Limitations

A limitation of *dpp* is that it is only able to detect *syntactic errors* and not *semantic errors*. For example, if a content model contains an element that is not defined in the DTD, this will not cause a problem. Similarly, attribute definitions are allowed even if the element that contains the attribute is not defined. Such errors must be detected by the Type Generator because they will cause problems during code generation. Some of the semantic errors could be detected by a two-phase DTD Parser, but we do not use this approach here.

dpp does not support short references, DATATAGs, rank groups and other optional features of SGML. It does not check identifier uniqueness, and it fails to allow parameter entity references in system identifiers and in attribute value literals given as default values in attribute definitions.

The original version of *dpp* contained a rule to allow an element to have the content model `ANY`:

```
<!ELEMENT body     - -    ANY>
```

If an element's content is ANY, it can contain `#PCDATA` or any of the elements defined in the DTD, in any order. SGML literature suggests that `ANY` should only be used for debugging purposes since it leads to unstructured documents. Therefore, the use of `ANY` is not allowed in this implementation. Our DTD Parser is not an isolated tool that checks the correctness of a DTD. It is part of a system and not a debugging tool.

Chapter 7

The Type Generator

The Type Generator is the system component that dynamically creates new Object Store types which model the characteristics specific to a class of documents, whenever support for that class of documents is added to the system. The Type Generator is invoked by the DTD Parser after a successful validation of the DTD. The element list, created by the DTD Parser, is used to automatically generate C++ code that defines a new ObjectStore type for each element in the DTD. As mentioned in Section 6.2, this element list contains the information necessary for dynamic type creation. To perform dynamic type creation in the most efficient way, the following design questions needed to be answered:

1. How can support for new DTDs be added to the system with as little impact as possible on running applications?
2. Can types be shared between different DTDs?
3. How can the new types be named to achieve type name uniqueness throughout the system?
4. How should the DTD components and their inter-relationships be modeled?
5. How can the Type Generator automatically detect where in the type hierarchy a new type should be introduced (What are its supertypes?)?

6. How should the interface between other system components (e.g. DTD Manager and Instance Generator) be designed?

To answer the first question, an analysis of the schema evolution facilities of ObjectStore is necessary. Basically, ObjectStore provides two techniques for dynamically adding types to the database schema. First, a type can be added as a *leaf node* in the inheritance hierarchy. Second, ObjectStore's schema evolution function can be used to add a type *anywhere* in the type hierarchy. As indicated in Chapter 4, the second technique is very time consuming and requires relinking all applications that work with the database after the schema change. Therefore, the approach of creating new types only as leaf nodes was chosen.

Design questions two, three, and six are discussed in Sections 7.1, 7.2, and 7.5, respectively. Questions four and five are addressed in Section 7.3.

7.1 Abstraction and Reusability

Sharing common type definitions throughout the system would reduce the complexity of the type system. In this context, there are two abstraction problems to deal with:

- the abstraction of common element definitions in one DTD, and
- the abstraction of common element definitions across different DTDs.

If two or more elements in the *same* DTD share common features such as common attributes or child elements, then these features could be automatically extracted by the Type Generator and promoted to an abstract supertype. This is a desired goal of the research project but it has been left out of the implementation because of time constraints and the nontriviality of the problem. Some of the problems that must be tackled in this context are:

- Naming conventions: Type names must be unique. Section 7.2 discusses the naming conventions of the system in the conventional sense. If we create abstract supertypes to factor out common characteristics, these types must have *unique* and *meaningful* names. There is no easy solution to this problem, since concatenating the names of the types derived from these abstract types will, in general, result in type names which are too long. Of course, a numbering system could be introduced (e.g. `Book_Supertype1`, `Book_Supertype2` etc.) to solve this problem, but the resulting names are not particularly meaningful. Meaningful type names are useful especially for application programs that make use of the new types.

- Automatic extraction of common features: The Type Generator has to iterate over the element list to detect common features. The question arises whether or not one common attribute or child element satisfies the overhead of creating a supertype. If not, what is the smallest meaningful unit?

If element definitions across different DTDs [illegible] quivalent, they should be represented by a common type in the type system. Ev[illegible] this approach sounds reasonable, it leads to the well-known semantic heterogeneity problem that has been studied extensively within the multi-database community [OSI+95]. Briefly, the problem is one of being able to determine whether two elements are semantically equivalent. Different definitions for type equivalence need to be introduced. Two types are *name equivalent*, if they have the same name. On the other hand, *structural equivalence* indicates that the type components recursively have the same names and types. Both definitions are not sufficient to ensure semantic type equivalence. For example, a `subtitle` in a book-DTD may be different than a `subtitle` in a `movie`-DTD. Alternately, elements having the same content definition (e.g. `#PCDATA`) may have different meanings (e.g. the `author` of a `book` and the `producer` of a `movie`). Since this problem is very complicated, we decided to give up some abstraction in favor of a semantically consistent type system. Future research will address the reuse of element types across different DTDs. For example, it might be possible to introduce the notion of "reusable types" and let the DTD designer decide which types should be reusable. This would require the extension of SGML in some way, either using an architectural form or a keyword to indicate reusable and reused types in DTDs.

7.2 Name Conventions

In the database schema, type names must be unique, since there is a certain structure and behavior associated with each type. Like other object-oriented database systems, ObjectStore does not allow the definition of two distinct types with the same name. Thus, when new types are automatically added to the database schema, type name uniqueness must be preserved. How can this be achieved?

Keeping the multimedia databases semantically consistent requires that document types (DTDs) are unique throughout the system. If the type of a document is `book`, the markup is specified by a unique `book` DTD entered into the system. Even if we have classes of documents with very similar structure, we still need to define different DTDs for them (e.g. `book1`, `book2`, ...). The DTD uniqueness is achieved in the following way. All files that are created during code generation have the *document type*[1] of the DTD as a *prefix* in their file names. For example, `<doctype>.hh` (e.g. `book1.hh`) and `<doctype>.C` (e.g. `book1.C`) contain the type definitions (e.g. for `book1`), and `<doctype>_init.C` (e.g. `book1_init.C`) defines functions to initialize the type extents of the new types. If the code for a particular DTD already exists and the DTD is stored in the database, attempts to insert the DTD again will fail. In this way, we maintain the uniqueness of DTDs in the system.

A similar approach is used to enforce type name uniqueness. According to SGML, in a particular DTD, all element names must be unique. This is not true for elements defined in different DTDs. For example, the two DTDs `book1` and `book2` might have an element defined which is called `chapter`. Therefore, we need a way to distinguish between both elements. Since document types are unique throughout the system, a combination of document type and element name must result in a unique type name. Thus, type names are built in the following way:

[1] Note that, the document type of a DTD is equal to the DTD name, normally the root element of the DTD, whereas the DTD filename refers to the file in which the DTD is stored.

- The type name for the root element of a DTD is the document type of that DTD with the first letter upper case and all other letters lower case (e.g. Book1).
- All other type names are a concatenation of the document type and the element name capitalized (e.g. Book1Chapter, Book2Chapter).

As we have seen, the document type of the DTD must be known to the Type Generator to name the files which contain the generated code and to name the element types. In general, there are two ways to obtain this information. First, the user can pass the root element name as a parameter when calling the DTD Parser. This information then gets passed to the Type Generator. Second, if no root element name is specified by the user, the root element, if unique, is automatically detected by the Type Generator. In any case, the element list is searched for all elements that are reachable from the root element, since code generation needs to be done only for these elements.

7.3 Type Generation for the Elements of a DTD

DTDs define the logical elements of documents of a particular type. In our system, for each DTD element a type is created that models the particular element. These types are created as part of the element type system, since the element type system is a uniform representation of DTD elements and their hierarchical relationships, whereas the atomic type system models primitive objects that are the basic components of multimedia documents.

At document insertion time, an instance of the appropriate element type is created for each element in the document. Hence, with this approach, the decision of how the documents are stored in the database is partially shifted to the DTD designer. But, how the translation of an element definition in the DTD to a type in the type system is done depends on the design of the Type Generator. Elements must be modeled in a way that expresses their structure and the relationships between them. Furthermore, interesting questions arise such as, which behavior should be associated with each type (What functions should be defined for each type?) and which characteristics of already existing types should be inherited.

7.3.1 Modeling of the Element Structure

Structured elements, elements with complex content models, are non-leaf nodes in the DTD composition hierarchy. Their subelements (child elements) can be modeled as data members in the element type.

A DTD specifies constraints for its structured elements. These are constraints on the type, the number, and the order of an element's subelements. Modeling these constraints is difficult, since ObjectStore provides little or no support for such an activity. In the following section, the specific problems and solutions are discussed.

Number of Subelements

The content model of an element specifies how often a child element can appear in it. With the "*", "+", and "?" occurrence indicators and the possibility of defining inclusions, a wide variety of constraints on the number of subelements can be specified.

```
<!ELEMENT section  - -  (title, (paragraph|list)*) +(footnote)>
```

For example, the DTD declaration of the element `section` given above specifies that there must be exactly one `title` element, zero or more `paragraph` and `list` elements, and zero or more `footnote` elements in the content of a `section` element. In C++/ObjectStore, there is no direct way of supporting some of these constraints. For example, there is no possibility of specifying that a data member cannot have a `NULL` value which is necessary to model the "+" (one or more) occurre[illegible]icator. This constraint could be enforced by the type's constructor which w[illegible]low instance creation only when at least the reference to one child element is given. Even though this solution works, it is not very efficient. Thus, the question arises whether the type system should enforce the constraints. Under the circumstances, the responsibility of maintaining the constraints can be shifted to the SGML document parser. Since the parser checks the document's conformance to a DTD, it will always enforce the number constraints. However, it is still the responsibility of the type system to provide enough storage space to keep information about all child elements, no matter how many there are. How can this be achieved?

The simplest solution would be to always create the data members as lists of pointers to objects of the appropriate types (element types of the child elements). This approach creates a large overhead. Managing lists in ObjectStore is very "expensive". ObjectStore lists are instances of the class os_list or its parameterized subclasses, classes in the ObjectStore class library. These classes have data members (e.g. for storing the cardinality of the list) and member functions (e.g. for inserting, removing, and retrieving elements). Thus, the storage requirements for a data member of type os_list are much higher than for a data member containing only information about one object. Moreover, the explosion of templates with parameterized classes leads to very large executables. Therefore, ObjectStore lists should only be used if they are really needed. To achieve this, the Type Generator must be designed well enough to automatically detect whether or not a data member should be created as a list. The criterion is whether or not a child element can occur more than once in an element's content model. For the example described above, the following code is generated (assuming the document type is Article):

```
class ArticleSection : public StructuredAnnotated {
public:
        ArticleTitle                 *title;
        os_List<ArticleParagraph*>  *paragraphList;
        os_List<ArticleList*>       *listList;
        os_List<ArticleFootnote*>   *footnoteList;
                  ...
}
```

Modeling the child elements as data members in the parent element specifies the hierarchical relationships between these elements. A `section` *contains* a `title` element, `paragraph` elements, `list` elements, and `footnote` elements.

As explained in Section 5.5, for each DTD to be supported by the system, one helper type will be created by the Type Generator as a subtype of the built-in type

DocumentRoot (e.g. ArticleRoot in the above example). Among other things, this type keeps track of the element's annotations. It has a data member for each annotated element of the DTD. The decision whether or not these data members should be created as lists is made in a similar way as in the case of element type creation. If an annotated element can occur more than once in a document, we need to store more than one annotation for that element. Thus the Type Generator must create the annotation data member as a list. To make this decision, the Type Generator must check the content models of all elements declared in a DTD for an element's occurrence.

Order of Subelements

In the DTD, a constraint specifying the order of subelements is introduced by the "," connector. If two elements are connected by this connector, they must appear in a document in the same order as they are defined in an element's content model. In the example used in the last section, we have this order of child elements for the `section` element: `title` is *followed* by any number of `list` and `paragraph` elements in any order. `footnote` elements can appear anywhere in the content.

To capture this constraint in the type system, a mechanism is needed to order the child elements. Since this feature is not present in ObjectStore, an implicit ordering of the child elements is assumed. The abstract supertype **Structured**, from which all structured elements are subtyped, contains a `childList` and the method `GetNth` to access the `childList`. After document instantiation, the `childList` contains pointers to all child elements of an element in the order they appeared in the document. Suppose, method `GetNth`, which is zero based, is applied to the element with "2" as a parameter. The element's "third" child will be returned. Note that, this method does not enforce the order constraint. It assumes that the elements that are stored in the `childList` are ordered.

Modeling Attributes

In general, attributes of an element become data members in the element type. If the element is a HyTime or a monomedia element, some of its attributes are already defined in its abstract supertypes. These are the attributes that are common to all element types which are derived from these types. For example, all ilink HyTime elements have a link-end associated with them. Therefore, the attribute `linkend` has been defined in the built-in HyTime type **Ilink_AF**. Thus, the Type Generator must know before creating the data members for attributes if the element is a HyTime or monomedia element. In that case, data members should be created only for the user defined attributes, that are the attributes *not* already defined in the system.

The type of an attribute data member depends on the value type defined for the attribute. If the value type is NUMBER/NUMBERS or IDREF/IDREFS, the type of the data member will be `integer` or a pointer to `Element`, respectively. All other attributes are stored in character strings.

A different solution would be to store all attribute values as character strings. We did not follow this approach since it makes the access to the attribute values less efficient. For example, following an element reference stored in a string (element ID) is much harder than following a pointer. Note that the approach used in this thesis requires the resolution of the ID references at document insertion time.

The number constraints for attributes can be handled in a similar way as for elements. If an attribute can consist of more than one value, an ObjectStore list is created for this attribute. This is the case when the value type of the attribute is either NUMBERS or IDREFS.

7.3.2 Modeling the Element Behavior

As explained in Section 5.4, most of the built-in element types define methods to model the behavior of their elements. When element types are subtyped from these types, they inherit their behavior. For example, type **Element** defines the function

GetDocument, since it is very likely that an element will be asked for its document (e.g. to obtain the document's text string or its ID). Similarly, type Structured contains access functions to access the element's child list (e.g. GetNth to retrieve the nth child element).

In the concrete element types, all data members are made public, thus making it unnecessary to define access functions for them. This solution has been chosen since it avoids the creation of huge amounts of code (two functions per data member (get/set)).

Each concrete element type has a constructor and a destructor function. Constructors usually involve memory allocation and initialization for a new instance of the type. Destructors involve cleanup and deallocation of memory for that object. If no constructor or destructor is provided, the compiler uses a default implementation for them.

Furthermore, the method SetChild is defined for each concrete structured type to set the element's child elements. Concrete element types that have attributes contain the method SetSpecificAttribute to set the element's attributes. The method ResolveRef is defined for each concrete element type that has ID references (IDREF/IDREFS) to resolve these references. These methods are called by the Instance Generator. They are discussed in detail in [EM96].

7.3.3 Automatic Detection of Supertypes

Depending on the structure and the semantics of an element in the DTD, the element type modeling the element will be subtyped from one or more of the built-in types of the element type system. The rules for when an element type should be subtyped from a built-in type are defined in Section 5.5. For example, the element type representing the root element of a DTD will always be subtyped from the built-in abstract supertypes Document, and StructuredAnnotated. Additionally, if the DTD makes use of the HyTime architectural forms, the type will be derived from HyDoc_AF, the built-in abstract supertype modeling HyTime documents.

The Type Generator is responsible for automatically detecting the supertypes of an element type to be created. Using the information in the element list, which is generated by the DTD Parser, the Type Generator finds out:

- what the root element of the DTD is,
- if an element is structured or not,
- if an element is a HyTime or MM element or not, and if so, to which architectural form it belongs (See Section 7.3.3.), and
- if an element should be annotated or not.

For example, to detect whether or not an element is a HyTime element, the attribute list of that element can be searched for an attribute named HyTime. The value of this attribute is set to the architectural form used (e.g. to HyDoc). (See Section 6.2 for details about the element list data structure.)

Depending on the test results, the appropriate supertypes are chosen. Supposing an element is structured and needs annotations, the element type is then derived from StructuredAnnotated.

The only difficulty is when monomedia elements are defined in the DTD. Monomedia elements, such as images, variants or data streams, make use of the atomic types defined in the atomic type system. As described in Section 5.4.6, these types should be subtyped from the appropriate MM types which are built-in abstract supertypes in the element type system. These MM types contain references to the atomic types (See Section 5.4.6.).

An Extension to SGML/HyTime

In standard SGML/HyTime there is no way of specifying that an element in the DTD should be considered a monomedia element. If we are not able to express these semantics in the DTD, the Type Generator cannot automatically detect whether or not an element is a monomedia element. To solve this problem, we have extended

SGML/HyTime with a feature that allows us to specify these semantics. There are two approaches:

Keywords: SGML can be extended with a keyword for each monomedia type defined in the type system. When an element is declared in the DTD, the keywords can be used to indicate the monomedia type of the element. Suppose, a DTD contains two monomedia elements, myimage and myaudio, which are of the MM types Image and Audio. Using the keyword approach, the element declarations would look as follows:

```
<!ELEMENT myimage  - - IMAGE>
<!ELEMENT myaudio  - - AUDIO>
```

Here, `IMAGE` and `AUDIO` are the keywords for the MM types Image and `Audio`, respectively. Since monomedia elements always have an empty content model, the keywords can be placed in the element's content declaration.

Architectural forms: Architectural forms have been introduced by the HyTime standard to specify certain HyTime specific element characteristics. A DTD element can be defined as a HyTime element by giving it an attribute with the name `HyTime` and a value indicating the HyTime architectural form used.

```
<!ATTLIST link
          HyTime  NAME      #FIXED ilink>
```

In this example, `link` is a HyTime element of type `ilink`.

A similar concept can be applied to specify monomedia elements in a DTD. Instead of HyTime architectural forms, **MM (multimedia) architectural forms** can be used. The element declarations look as follows (using the same example as for the keyword approach):

```
<!ATTLIST myimage
          MM      NAME      #FIXED image>
<!ATTLIST myaudio
          MM      NAME      #FIXED audio>
```

The attribute name `MM` indicates a monomedia element and the value of the `MM` attributes specifies the MM type.

In our project, we have adopted the second approach. We believe that this approach has two main advantages over the keyword approach. First, it is very similar to HyTime, since both use architectural forms to express certain semantics. Second, only small extensions to SGML/HyTime are necessary (the introduction of the `MM` key attribute and its architectural forms, one for each MM type). The keyword approach, on the other hand, requires the extension of SGML/HyTime with one keyword for each MM type defined in the type system.

The concept of MM architectural forms to indicate monomedia elements has been incorporated in the design of a DTD for multimedia news articles. The DTD has been developed as part of this thesis. It can be found in Appendix C. This DTD also contains data streams and variants which were introduced to define different quality-of-service levels for the quality-of-service negotiation.

7.4 Object Persistence

In ObjectStore, any C++ object can be made persistent. These objects can be handled in the same way as non-persistent objects. Persistent objects can be accessed if they are directly associated with a *database root*, or if they can be reached by navigation from a database root.

ObjectStore does not maintain type extents automatically. Extents are important for queries that search over instances of a certain type. For example, the following query "Select all documents that have the word **`politics`** in their headline" requires searching the extent defined for the DTD element **`headline`**.

In this project, type extents are automatically maintained as parameterized ObjectStore sets with the type as parameter. These sets are created as database roots. This solution is not optimal, since it creates an explosion of database roots if the system supports multiple DTDs. (In ObjectStore, the optimal number of database roots is less than ten, since the search over the database roots is done sequentially.) Thus, future effort should be put into limiting the number of database roots in the system, e.g having one database entry point per DTD. This approach has not been implemented yet, since it requires changes in the query engine, a project component that has been developed at the University of Waterloo.

When an instance of an element type is created, the type's constructor updates the type extent with a reference to the new object. Similarly, the type's destructor deletes the object reference from the type extent.

Before the type extents can be used, they must be created and initialized. This is done by the Type Generator. The Type Generator not only generates C++ code which defines the new element types, it generates an application program that creates and initializes the type extents in the database as well. After the generated code is compiled (new types as well as the initialization program) and the database schema is updated to reflect the new types, the initialization program is invoked by the Type Generator.

7.5 The Interface with Generic Document Instantiation

The Type Generater does not only create new element types by modeling the elements defined in a DTD; it also executes certain tasks that are needed for the automatic document instantiation. These are:

- the creation of some member functions for the element types that are called during document instantiation. These include the method **SetChild**, defined for each structured element (type), to set the element's child elements, the method **SetSpecificAttribute**, defined for each element that has attributes to set these attributes, and the method **ResolveRef**, defined for each element that has **ID** references (**IDREF/IDREFS**) to resolve these references.

- the creation of meta-types for the newly created element types. Each object in the database knows its type. Each concrete element type in the type system has a "type class" associ[illegible]d with it (e.g. the meta-type of type `ArticleSection` is type `ArticleSectionType`). A meta-type co[illegible]ains meta information about its corr[illegible]ponding element type (e.g. the name and types of the element's child elements and attributes). Among others, it defines the function `CreateObject` that creates an object of the corresponding element type.

The sub-components of the Type Generat[illegible]r that fulfill these tasks have been developed as part of the thesis described in [EM96]. They use the information stored in the el[illegible]ment list, gathered by the DTD Parser and the other components of the Type Generator, for code generation.

Chapter 8

The DTD Manager

The DTD Manager is invoked by the Type Generator *after* type generation of both the element types and the meta-types, and *after* the generated code is compiled. This is when all requirements are met for inserting a document that conforms to the particular DTD into the database.

8.1 The DTD as an Object

The DTD Manager stores the DTD as an object of the built-in type **Dtd** in the database. As soon as the DTD object is stored, documents conforming to that DTD can be inserted into the database.

Type **Dtd** has the data members `name`, indicating the name of the DTD, and `dtdstring`, a character string in which the DTD text is stored. It is necessary to store the original DTD, since it is not possible to recreate the original DTD from the type information. For example, the DTD Parser resolves entity references automatically, making it unnecessary to keep any information about entities in the system. The SGML Parser, on the other hand, needs to work with the original DTD in order to parse SGML documents correctly.

Both `name` and `dtdstring` are parameters in the constuctor function of type **Dtd**. They must be known at object creation time. Thus, the DTD Manager has to be provided with the following information:

- the name of the DTD, which is equivalent to the document type of the DTD,
- the DTD file name (The content of the DTD file has to be read into `dtdstring`.), and
- the name of the database in which the DTD and the documents conforming to that particular DTD will be stored. Before objects can be inserted into a database, the database needs to be opened.

From where can this information be obtained? The solution is simple. The DTD Parser and the Type Generator gather this information to fulfill their tasks. In particular, the DTD file is needed by the DTD Parser to parse the DTD, the DTD document type and the database name are needed by the Type Generator to name the new element types and to initialize the type extents, respectively. Thus, there is no need for obtaining this information again. It can simply be passed as parameters by the Type Generator when the DTD Manager is called.

8.1.1 Advantages of Storing the DTD as an Object

Storing the DTD as an object has many advantages:

- Whenever a **Dtd** object is found for a certain DTD in the database, we know that the DTD is supported by the system.
- The uniqueness of DTDs (document types) can be enforced. Before a **Dtd** object is inserted into a database, the database is queried for a **Dtd** object that has the same **name** (document type) as the object to be inserted. If the query fails[1], the **Dtd** object is inserted into the database. Otherwise, an error is generated.

[1]This is, if no **Dtd** object with the same **name** was found.

- Much useful information can be kept about the DTD that can be queried at any time. For example, the DTD object could store references to all documents conforming to that DTD. That way, access to all documents of a particular type would be very fast.

8.2 The Type Objects

The DTD Manager creates a type object for each meta-type defined for a particular DTD (See Section 7.5.). These type objects are stored in the **Dtd** object besides the DTD name and the DTD string. The creation of the meta-type objects is described in more detail in [EM96]. The meta-objects are used by the Instance Generator to automatically instantiate SGML documents in the database. Each meta-type object can be asked to create an object of its corresponding element type by calling the member function `CreateObject`.

Chapter 9

Related Work

Since SGML is becoming the de facto standard for structured document handling, there is a lot of ongoing research in this area. In the past, structured documents were often stored in file systems which do not support document management extensively. Now, there is a trend to develop database systems, especially object-oriented database systems, for storing SGML documents. Many researchers believe that document management can benefit from the use of database management facilities such as recovery, concurrency control, or query capabilities. In this chapter, some approaches reported in literature are described and compared to the approach taken in this thesis. Special attention is paid towards object-oriented database systems that support SGML and HyTime.

D-STREAT

In [ABH94] a database application for SGML documents, called D-STREAT, is described that is based on the OODBMS VODAK. The system aims to be flexible and generic in order to administer documents of arbitrary types. [ABH94] points out that, for the management of structured document bases, it is very important not to be restricted to a fixed number of DTDs. It should be possible to insert and modify DTDs in the database system without affecting running applications.

Since our project group realized the need of supporting arbitrary DTDs as well, a generic type system has been developed in this thesis that can handle any DTD. New DTDs can be inserted in the database at any time without affecting running applications, since dynamic type creation is done without using the ObjectStore schema evolution facility. Contrary to D-STREAT, we did not examine the possibility of changing a DTD definition already stored in the database. Compared to the multimedia database system described in this thesis, D-STREAT is clearly the most similar system found in the literature.

In D-STREAT, adding support for a new DTD is achieved in the following way:

- First, the DTD is parsed by a parser generator to generate the DTD as a document instance of a particular DTD, the so-called *super-DTD*. The generated super-DTD instance differs from the original DTD only syntactically. The super-DTD describes the definition of DTDs as defined by SGML.

- Second, the DTD-document created in step one is parsed by a parser for super-DTD instances. This parser is an extension of the publicly available ASP ("Amsterdam Parser"). The transformation in step one has been done in order to use this parser for both parsing the DTD and the SGML documents. During parsing, the DTD's conformance to the super-DTD is checked and the DTD is instantiated in the database. The database schema contains three built-in classes that represent the components of the DTD. These classes are ELEMENT, ATTRIBUTE, and ENTITY to store the element, attribute and entity definitions of the DTD, respectively.

- Third, using the newly created instances of ELEMENT, ATTRIBUTE, and ENTITY, new classes are created for the logical elements in the DTD (e.g section, chapter etc.). This is achieved by using the *Metaclass* feature of VODAK. Instances of metaclasses are classes themselves. Metaclasses have an instance creation method, whose invocation leads to the creation of a new class. Thus, system shutdown is not necessary to extend the database schema with new element classes. Note that, there is a differentiation between *terminal* and *non-*

terminal element types. Terminal element types, such as PCDATA or CDATA, are part of the SGML definition. Since they are not specific to a DTD, the classes representing them are predefined by the system as instances of the metaclass TERMINAL. On the other hand, nonterminal element types are element types that are specific to a DTD. Classes representing them can only be created when the DTD is known to the system, that is, when the DTD is inserted (see step one and two). These classes are created as instances of the metaclass NONTERMINAL.

Once the new classes for the logical elements in the DTD are created, documents conforming to that DTD can be inserted into the database. As mentioned before, the parsing of the SGML documents is done with the "Amsterdam Parser". The parser checks the document's conformance to its DTD and invokes methods to create the objects in the database that represent the logical document components.

Contrary to D-STREAT, we use a DTD parser (which understands DTD syntax) and not a document parser (which understands document syntax) to check the DTD's conformance to the super-DTD. Thus, we do not need a parser generator (as described in step one for D-STREAT) to create a document instance of a meta-DTD that can be parsed using an ordinary SGML document parser.

Adding HyTime support to D-STREAT is achieved by defining a metaclass for each HyTime architectural form. They are called HyTime metaclasses. Each HyTime element in the document has *two* objects in the database associated with it: an instance of the element-type class containing the element-type specific features of the element, and an instance of the HyTime element-type class storing the HyTime information. This is necessary, since VODAK does not support multiple inheritance. In our system, only one object is associated with each HyTime element. This object contains all element-type-specific and all HyTime-specific information, since it inherits from both an element type, such as StructuredAnnotated, and an HyTime type, such as Ilink_AF.

In [BO], an improvement to the first D-STREAT prototype is described. The system is now configurable. The DTD can be enriched with semantic information that helps to handle the management of the DTD and its documents in a more efficient way. One can choose between physical representations for elements (e.g. *flat* vs. *non-flat*). several indexing structures. and diverse mechanisms supporting access to the documents' secondary structure. This is achieved by extending SGML with certain attribute definitions. For example. to allow for indexing. the super-DTD must contain the following definition:

```
...
<!ELEMENT ELEM          ...>
<!ATTLIST ELEM          ...
          INDEX   (DIRECT|NOT)   NOT
...>
```

where INDEX is the keyword for indexing. and DIRECT stands for direct indexing. The DTD would then contain entries similar to these:

```
<!ATTLIST author
          INDEX   DIRECT>
```

In D-STREAT, this "keyword" approach is used to extend SGML/HyTime to make the system configurable. Contrary to that, in our system, we use an architectural form extension to SGML/HyTime. which is similar to HyTime. to achieve the same system behavior (See Section 7.3.3.). We can define any number of MM architectural forms to configure the system. e.g. an architectural form to allow for indexing. or to indicate reusable elements. The difference to D-STREAT is that we use only *one* key attribute, the MM attribute. D-STREAT extends SGML with one key attribute for each configurable characteristic.

It is important to point out the difference between *flat* and *non-flat* elements. In the first D-STREAT prototype, for each element in a DTD an element type was created. The text content of a document was not stored as single text string as in our system. Rather, each element type instance contained a particular portion of the text. This approach proved to be rather slow for many access operations. Thus, in [BO] a hybrid approach for physical document representation was used. Now, parts of the document can be left *flat*. An instance of class **FLAT** will be created for each flat document portion. That way, the text content of a number of document components can be stored in one object. This improves the speed for text based search. For *non-flat* document components an instance of the corresponding element type is created. as before. The decision as to which element types should be flat is left to the DTD designer.

To summarize, D-STREAT and the system described in this thesis are very similar. They are both based on SGML/HyTime, they support arbitrary DTDs, and they achieve the dynamic addition of the DTDs to the type system with as little impact on the existing applications as possible. But, they are different in the underlying database system used (VODAK for D-STREAT, ObjectStore for our project) and therefore, in the way they perform the dynamic addition of the element types to the type system. Furthermore, D-STREAT's database schema is essentially a flat class hierarchy, with depth one. In comparison to that, our system is more structured having the abstract supertypes **Element**, **Structured**, **AnnotatedElement**, **StructuredAnnotated** and so forth. This makes it more complicated to generate new types, since information must be generated that helps to decide from which of these types to subtype. On the other hand, this structured type system is more efficient in factoring out common type characteristics compared to the flat type system of D-STREAT.

VERSO

In the VERSO project at INRIA, France, an object-oriented database system for the storage of SGML documents is currently being developed on top of the O_2 DBMS. In [CACS94], two extensions to the O_2 data model are proposed to facilitate the mapping from SGML documents to O_2 instances. These extensions are union types and ordered tuples. Union types are used to describe alternative content models of elements ("|" connector), and ordered tuples model the "," connector. Like our system, the system described in [CACS94] supports multiple DTDs. DTDs are mapped into O_2 schemas and documents into corresponding objects and values by using an extended version of the Euroclid SGML parser. The parser generates a BNF grammar. This grammar is automatically annotated with the appropriate semantic actions that interact with the database. [CACS94] also describes an extension to the O_2 query language (O_2SQL) to deal with the O_2 data model extensions and with the requirements of document retrieval such as querying data with incomplete structure knowledge. The authors point out that this language is particularly suited for the current extensions of SGML to multimedia and hypermedia documents (e.g. using HyTime).

HyOctane

At the University of Massachusetts, a distributed multimedia information system is being developed which uses HyTime as its data model and interchange format. The system described in [KRRK93] consists of an SGML parser, an ObjectStore database for storing the HyTime documents, and the HyTime engine HyOctane for accessing and presenting the document instances. The database schema is based on one specific DTD, the HDTD – a HyTime-conforming document type definition for interactive multimedia presentations. Contrary to the multimedia system described in this thesis, the system does not support the automatic addition of DTDs to the type system. But, the authors claim that their type system can be easily extended to support other HDTDs.

The design of the system is layered. It has an SGML layer, a HyTime layer, and an application layer. The SGML layer consists of three types: **document**, **element**, and **attribute**. When a document is parsed, an instance of type **document** is created. This instance has references to all element instances. For each element in the document, an instance of type **element** is created. Each element instance knows its parent element, its children, and its attributes (instances of type **attribute**).

After the objects in the SGML layer are instantiated, the HyOctane engine queries the SGML layer to obtain the data needed to initialize the HyTime layer. Each element instance belonging to an architectural form is processed into an instance of that architectural form. In the HyTime layer, each architectural form has a type associated with it. Each type has a data member for each of its HyTime attributes and for its data content. While the attribute values and the data content are just character strings at the SGML layer, at the HyTime layer this information is converted to the appropriate data types.

Once HyOctane has finished its processing, the application process is invoked which instantiates the application layer by querying the SGML and HyTime layers. The application layer contains a type for each element defined in the DTD. The application process works with this layer only. All changes to the objects of the application layer are propagated down to the HyTime and SGML layers.

Requirements of Multimedia Applications

[WKL90] and [Thu92] are ground-breaking papers in the area of multimedia database systems. They highlight the requirements of multimedia applications and investigate which database systems are best suited to support these requirements. The requirements include a data model that is sufficiently flexible to allow a very natural definition and evolution of the database schema that represents multimedia documents. The data model must also include presentation information and capture the complex relationships between the document components. Other requirements are, for example, support for versioning, management of large data volumes, and access to the stored documents based on document structure, document type, and document content.

Both papers are very general in their considerations, since they do not adopt any specific document standard. They conclude that object-oriented database management systems are most appropriate for fulfilling the requirements imposed by multimedia applications.

Chapter 10

Conclusion and Future Work

This thesis describes the design of an object-oriented multimedia database system. The main contribution of this work is the development of a flexible and extensible type system that is general enough to support different classes of multimedia documents. Hence, this thesis is a generalization of the work described in [Vit95], where a multimedia DBMS for a news-on-demand application was developed. The database system described here, stores multimedia objects as well as meta information about them. The type system is designed in strict adherence to the SGML/HyTime document standard.

To summarize, the important features of this work are:

- A generic and structured type system was developed. This type system consists of two separated parts, the Atomic Type System, modeling the basic multimedia objects with different variants composed of multiple data streams, and the Element Type System, modeling the document components and their interrelationships.

- An interface between the Atomic and the Element Type System based upon the MM types was developed.

- Different kinds of multimedia applications are supported by the system. Adding support for a new class of documents is done by dynamically adding new types to the Element Type System. The types are inserted into the database schema without costly schema evolution.

- Three system components were developed that realize the dynamic type creation. These are the DTD Parser, the Type Generator, and the DTD Manager.
- The DTD was modeled as an object in the database.
- New architectural forms (`MM forms`), similar to HyTime, were introduced that refer to built-in element types to make them reusable as supertypes of the types to be created.
- A new DTD for multimedia news articles, incorporating data streams and variants, was developed.
- An interface to the system components (described in [EM96]) that provide facilities for the automatic insertion of documents into the database was developed.

In the future, some possible enhancements to the work described in this thesis, and to the multimedia database system itself are:

- The reuse of types within a single DTD should be supported. If two or more elements in a DTD share common features, such as common child elements or attributes, then these features could be automatically extracted and promoted to an abstract superclass. This was one of the design decisions made in this thesis but has been left out in the implementation because of the time constraints given and the nontriviality of the problem.
- The reuse of types across multiple DTDs should be supported. In general, common element definitions across different DTDs should be represented by common types. This is not done in our system due to the problem of semantic heterogeneity. With a better understanding of this problem, it might be possible to achieve at least partial reusablility. In that context, it might be interesting to investigate the possibility of introducing a new architectural form that identifies

reusable types in a DTD. Alternately, some types that are known to be used ite frequently (such as **Author**, **Date**, and **Title**) could be defined as built-in types in the Element Type System and reused using the **MM** architectural form (or any other new architectural form).

- The system should make use of indexes. In the current system, no indexes are created to improve access efficiency. The main reason is the lack of knowledge about which elements in a certain document class are accessed quite frequently. In the news-on-demand database system, the indexes were built using intuition only. Now, an automatic way must be found to identify the elements in a DTD that would benefit most from the use of indexes. This can be achieved, for example, by introducing an **INDEX** architectural form, leaving the decision where to build an index to the DTD developer.

- Currently, two different parsers are used in the system: the DTD Parser (bison/flex implementation) for parsing the DTD, and the SGML Parser (C++ implementation) for parsing SGML documents. The new release (fall 1995) of the SGML Parser is supposed to have all features necessary for parsing a DTD. If this proves to be true, this parser can replace the DTD Parser currently used.

- Updates should be supported by the system. The database system, as it is now, supports only a read only environment for documents. Updates (e.g. adding one line of text to a paragraph) are not easily achieved, since they involve updating the annotations of all annotated elements positioned in the document after the updated element. Currently, there is no automatic way of doing that.

- Relative indexing should be used for annotations instead of absolute indexing. In the database, the content of a document is stored as a single text string. Annotations are defined for each (annotated) element in the document to indicate the start and end index of that element in the document's text string. These indexes are absolute values starting from the begin of the text string. If we support new document types, such as books, in our database, the content is too large to be effectively stored as a single string in the database. How can we divide the text string into substrings with as little impact as possible? One possible solution might be to annotate the DTD. The DTD object could store the start and end indexes of the substrings as absolute values. The annotations of the elements in the substrings could be given relative to the beginning of the substrings.

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

- Future work will focus on the development of a query processor and optimizer that handle a distributed objectbase and support content-based queries of images and videos.

- ObjectStore, the database system currently used, will be replaced with TIGUKAT. TIGUKAT is an extensible ODBMS that has inherent support for multimedia applications. It is currently prototyped at the Laboratory for Database Systems Research of the University of Alberta.

Appendix A

HyTime Attributes

Axis_AF

`axismeas`	SMU (standard measurement unit)
`axisdim`	size of the axis in MDUs (measurement domain units)
`axismduList`	axis measurement domain unit, SMU to MDU ratio

Dimspec_AF

`mark1`	first marker of the dimension specification A marker is an integer other than zero that represents a position within an addressable range of quanta on coordinate axis.
`mark2`	second marker of the dimension specification

Ilink_AF

`linkendsList`	link ends that point to some objects
`anchrole`	anchor roles for each link end, an anchor role is a declared name that can be used to communicate the meaning or significance of the anchor

Event_AF

`pls2granList`	pulse to granule ratio. pulses are "meaningful repetitions" that can occur along an axis
`exspecList`	extent specification, associates scheduled extents with an event

Evsched_AF

`axisord`	ordered set of axis names of the finite coordinate space
`apporder`	declares whether the order of the SGML representation of the elements in the schedule is significant to the application and must be preserved
`sorted`	declares whether the order of the SGML representation of the elements in the schedule is sorted
`basegran`	base measurement granule for each axis (e.g. minutes, seconds)
`gran2hmuList`	base granule to HMU (HyTime measurement granule) ratio for each axis
`pls2granList`	pulse to granule ratio
`overrun`	determines outcome if dimensions specify nonexistent offsets into the axes used, e.g. error, ignore, truncate

Fcs_AF

`axisdefs`	lists the name of the axes that define the finite coordinate space
`fcsmdu`	finite coordinate space measurement domain unit, SMU to MDU ratio

Appendix B

A Example for the DTD Parser

The following example illustrates, how the data structures are built by the DTD Parser while validating the DTD. Suppose, the DTD contains the following element definition:

```
<!ELEMENT (loc|source|subject)  - -  (#PCDATA)>
```

When the DTD Parser parses this definition, it resolves the following rule.[1]:

```
elementdecl: MDELEMENT elemtype minimiz contentdecl MDC {          [1]
               Insert_NameGrp($2);
               Copy_ContentDecl($2, $4);
             };
```

MDELEMENT and MDC are tokens for the expressions <!ELEMENT and >, respectively. After this rule is resolved, elemtype ($2) contains a NameGrp data structure, which itself contains a list of element names (loc, source, subject) and the indicator OR_G indicating that the elements of this NameGrp are connected by "|". Similarly, contentdecl ($4) will hold a data structure that stores all information about the content model. minimiz contains no information. As specified in Section 6.2, no information is needed to be stored about data tag minimization. Therefore, the rules to resolve minimiz are executed without taking any action.

[1]To explain the example, the rules have been numbered using [1], [2], ..., on the right-hand side of the rules.

After rule [1] is successfully resolved, the user action (enclosed in {}) is performed. First, the function `Insert_NameGrp($2)` creates an entry in the element list for each element in `NameGrp` (if an entry did not already exist). Second, `Copy_ContentDecl` copies the information about the content declaration in the element list entries of all elements in `NameGrp`. But, how do the data structures get created in `elemtype` and `contentdecl`? To resolve rule [1], the parser tries to match rules for `elemtype`, `minimiz`, and `contentdecl`. In the following, the data structure creation is shown on the example of `elemtype`:

```
elemtype: namesym {
            $$ = Create_NameGrp(NO_G, Add_List_End(NULL, $1));
          }
          | GRPO namegrp GRPC {                                     [2]
            $$ = $2;
          };

 namegrp: seqnames {
            $$ = Create_NameGrp(SEQ_G, $1);
          }
          | ornames {                                               [3]
            $$ = Create_NameGrp(OR_G, $1);
          }
          | andnames {
            $$ = Create_NameGrp(AND_G, $1);
          };

 ornames: namesym OR namesym {
            $$ = Add_List_End(NULL, $1);
            Add_List_End($$, $1);
          }                                                         [4]
          |
            ornames OR namesym {
            Add_List_End($1, $3);
          };
```

When the `orname` rule is matched ([4]), a list with the three element names, that are part of the element definition, is created. This list is backed up to the rule for `namegrp` ([3]). There a `NameGrp` structure is created that includes the list of element names and the group indicator (`OR_G`). Now, this structure is backed up to the next higher rule level to resolve rule [2], and from there to rule [1].

Appendix C

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>
```

Bibli graphy

[ABH94] K. Aberer, K. Böhm, and C. Hüser. The prospects of publishing using advanced database concepts. *Electronic Publishing*, 6(4):469–480, 1994.

[BO] K. Böhm. Building a configurable database application for structured documents. GMD-IPSI Darmstadt, submitted for publication.

[BKKK87] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD RECORD*, 16(3):311–322, 1987.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *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.

[EM96] S. El-Medani. Support for document entry in the multimedia database. Master's thesis, University of Alberta, Department of Computing Science, 1996. (forthcoming).

[KRRK93] J. F. Koege L. W. Rutledge, J. L. Rutledge, and C. Keskin. HyOctane: A HyTime engine for an MMIS. In *Proceedings of the ACM Multimedia Conference '93*, pages 129–135, 1993.

[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.

[Obj94] ObjectStore Release [illegible] for OS/2 and AIX/xlC Systems. Object Design, Inc. [illegible] 4.

[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.

[OSI+95] M.T. Özsu, D. Szafron, P. Iglinski, G. El-Medani, S. El-Medani, M. Schöne, and C. Vittal. Database management support for a news-on-demand application. In *Proc. First International Symposium on Technologies and Systems for Voice, Video, and Data Communications - Multimedia: Full-Service Impact on Business, Education, and the Home*, volume 2617, October 1995.

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

[PS87] D. J. Penney and J. Stein. Class modification in the gemstone object-oriented dbms. In *OOPSLA '87 Proceedings*, pages 111–117, October 1987.

[SZ86] A. H. Skarra and S. B. Zdonik. The management of changing types in an object-oriented database. In *OOPSLA '86 Proceedings*, pages 483–495, September 1986.

[Thu92] B. Thuraisingham. On developing multimedia database management systems using the object-oriented approach. *Multimedia Review*, 3(2):11–18, 1992.

[TK89] L. Tan and T. Katayama. Meta operations for type management in object-oriented databases. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 241–258, 1989.

[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.

[WKL90] D. Woelk, W. Kim, and W. Luther. An object-oriented approach to multimedia databases. *Communications of the ACM*, 33(9):90–103, 1990.