Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

DISSERTATION

– PTDOM –

A Persistent Typed Document Object Model for the Management of MPEG-7 Media Descriptions

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften

unter der Leitung von

Univ.-Prof. Dr. Wolfgang Klas

Institutsnummer 384
am Institut für Informatik und Wirtschaftsinformatik
der Universität Wien

Zweitgutachter

Univ.-Prof. Dr. Christian Breiteneder

Institutsnummer E188
am Institut für Softwaretechnik und Interaktive Systeme
der Technischen Universität Wien

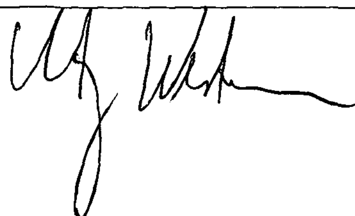
eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Inf. Gerd Utz Westermann

Matrikelnummer 0027225
Hebragasse 5/25
1090 Wien, Österreich

Wien, am 10.2.2004



Zusammenfassung

MPEG-7 ist ein vielversprechender Metadatenstandard, der eine umfassende Beschreibung multimedialer Inhalte ermöglicht. Mit einer zunehmenden Zahl an verfügbaren MPEG-7-Medienbeschreibungen wird eine angemessene Datenbankunterstützung zur Verwaltung solcher Beschreibungen immer wichtiger. Da MPEG-7-Medienbeschreibungen im wesentlichen XML-Dokumente sind, welche Medienbeschreibungsschemata folgen, die in einer Erweiterung der XML Schema-Sprache namens MPEG-7 DDL verfaßt sind, ist es ein naheliegender Gedanke, XML-Datenbanklösungen für diese Aufgabe einzusetzen.

Die vorliegende Dissertation will diesen Ansatz näher untersuchen. Sie stellt zunächst einen ausführlichen Anforderungskatalog auf, der von einer zur Verwaltung von MPEG-7 Medienbeschreibungen eingesetzten XML-Datenbanklösung erfüllt werden sollte. Gegen diese Anforderungen werden 21 repräsentative, dem Stand der Technik entsprechende XML-Datenbanklösungen gründlich geprüft: native XML-Datenbanklösungen und XML-Erweiterungen klassischer Datenbank-Management-Systeme – kommerzielle Systeme, Forschungsprototypen ebenso wie Open-Source-Projekte. Die Untersuchung fördert beträchtliche Defizite heutiger XML-Datenbanklösungen zutage, die ernsthaft deren Eignung zur Verwaltung von MPEG-7-Medienbeschreibungen beeinträchtigen. Ein zentrales Problem der betrachteten Lösungen ist, daß sie zum größtenteils die in Medienbeschreibungsschemata vorhandene Schema- und Typinformation bei der Speicherung von MPEG-7-Medienbeschreibungen ignorieren. Als Konsequenz werden die großen Mengen nicht-textueller Daten wie bspw. Frequenzspektren, Farbverteilungen und Bewegungsvektoren von Bildobjekten, die üblicherweise in MPEG-7-Medienbeschreibungen enthalten sind, unzulänglicherweise als Text gespeichert. Dies behindert den sinnvollen Zugriff auf diese Daten und deren Verarbeitung.

Angesichts solcher Probleme ist es das Ziel dieser Arbeit, eine XML-Datenbanklösung zu entwickeln, welche den Anforderungen zur Verwaltung von MPEG-7-Medienbeschreibungen besser genügt. Diesbezüglich leistet die Arbeit drei wesentliche Beiträge:

Erstens stellt sie das Typed Document Object Model (TDOM) vor, ein generisches Datenmodell für XML-Dokumente mit besonderem Augenmerk auf die Repräsentation nicht-textueller Inhalte. TDOM bietet sogenannte typisierten Repräsentationen zur adequaten Darstellung von XML-Dokumentinhalten gemäß ihres jeweiligen, in der Schemadefinition des Dokuments spezifizierten Typs. Auf Basis von TDOM können Anwendungen auch auf nicht-textuelle Daten innerhalb von MPEG-7-Medienbeschreibungen geeignet zugreifen und diese typkonform verarbeiten.

Zweitens stellt diese Arbeit Typisierungsautomaten vor, ein sprachunabhängiger Formalismus zur Zwischenrepräsentation von Schemadefinitionen für XML-Dokumente. Ein Typisierungsautomat ist nicht nur in der Lage, mittels TDOM dargestellte XML-Dokumente gegen die von ihm repräsentierte Schemadefinition zu validieren. Er ist ebenfalls in der Lage, geeignete typisierte Repräsentationen der Inhalte dieser Dokument herzuleiten und zu erzeugen. Da der Mechanismus der Typisierungsautomaten bis zur Ausdrucksstärke von MPEG-7 DDL erweiterbar ist, stellt er eine adequate Grundlage zur Zwischenrepräsentation beliebiger MPEG-7-Medienbeschreibungsschemata dar.

Drittens beschreibt die Arbeit die Implementierung des Persistent Typed Document Object Model (PTDOM), einer prototypischen XML-Datenbanklösung. Diese verfügt über einen MPEG-7 DDL-konformen Schemakatalog, der Typisierungsautomaten verwendet, und eine Dokument-Management-Komponente, die auf TDOM zur Repräsentation von XML-Dokumenten aufbaut. Unter Verwendung des Schemakatalogs nutzt PTDOM die Vorteile von TDOM zur typgerechten Speicherung von XML-Dokumentinhalten. Berücksichtigt man zudem die weitreichende Unterstützung von Datentypen, benutzerdefinierten Routinen und Indexstrukturen sowie wie seine durchgehende Erweiterbarkeit, so stellt PTDOM eine XML-Datenbanklösung dar, welche die meisten Anforderungen zur Verwaltung von MPEG-7-Medienbeschreibungen erfüllt und sich deshalb hervorragend zu diesem Zweck eignet.

Die Relevanz der Beiträge dieser Dissertation ist nicht nur auf das Gebiet von MPEG-7 beschränkt. Da TDOM, Typisierungsautomaten und der implementierte Prototyp prinzipiell zur Verwaltung beliebiger XML-Dokumente verwendbar sind, legt diese Arbeit auch einen Grundstein für eine neue Generation allgemeiner XML-Datenbanklösungen, die verfügbare Schema- und Typinformation zur Speicherung von XML-Dokumenten nutzen. Die Verfügbarkeit solcher Lösungen ist in all jenen Anwendungsgebieten äußerst wünschenswert, in denen datenzentrierte XML-Dokumente mit großen Anteilen nicht-textueller Daten auftreten.

Abstract

MPEG-7 is a promising metadata standard for the extensive description of multimedia content. The amount of MPEG-7 media descriptions available is continuously increasing and adequate database support for the management of larger numbers of such descriptions is gaining more and more importance. Since MPEG-7 media descriptions essentially are XML documents following media description schemes defined with an extension of XML Schema named MPEG-7 DDL, employing XML database solutions for their management is an idea that lies close at hand.

It is the aim of the present thesis to explore this idea in detail. It develops an extensive set of requirements that should be met by any XML database solution employed for the management of MPEG-7 media descriptions. Against these requirements, it thoroughly examines 21 representative state-of-the-art XML database solutions: native XML database solutions as well as XML extensions of traditional database management systems – commercial systems, research prototypes, as well as open source projects. The examination unveils considerable deficiencies of existing XML database solutions that seriously limit their suitability for the management of MPEG-7 media descriptions. One of the major problems is that the analyzed solutions largely ignore schema and type information that is available within media description schemes for the storage of MPEG-7 media descriptions. As a consequence, large amounts of complex non-textual data typically contained in MPEG-7 media descriptions, such as frequency spectrums, color distributions, and object motion vectors, are inadequately stored and represented as text hindering reasonable access to these data and their appropriate processing.

Facing these problems, this thesis sets out to develop an XML database solution which suits the needs of the management of MPEG-7 media descriptions. In this regard, it makes several important contributions:

Firstly, the thesis proposes the Typed Document Object Model (TDOM) as a generic data model for XML documents that pays particular attention to the representation of non-textual contents. TDOM provides the notion of typed representations to treat the contents of an XML document in a way that is adequate to the respective content type specified in the schema definition to which the document complies. On the basis of TDOM, applications can reasonably access and process even complex non-textual data contained in MPEG-7 media descriptions.

Secondly, the thesis proposes typing automata as a formalism for the intermediary representation of schema definitions for XML documents that is independent of any particular XML schema definition language. A typing automaton is not only able to validate TDOM-represented XML documents against a schema definition; it is also able to infer and produce appropriate typed representations of the contents of these documents. As typing automata can be extended up to the expressiveness of MPEG-7 DDL, they constitute an adequate foundation for the intermediary representation of arbitrary MPEG-7 media description schemes.

Thirdly, the thesis describes the implementation of the Persistent Typed Document Object Model (PTDOM), a prototypical XML database solution. PTDOM features an MPEG-7 DDL-compliant schema catalog that is based on typing automata and a document management component that employs TDOM for the representation of XML documents. Using the schema catalog, PTDOM takes advantage of TDOM for the typed storage of XML document contents. Furthermore considering its profound support for datatypes, user-defined routines, and index structures and its profound extensibility, PTDOM satisfies most of the requirements for the management of MPEG-7 media descriptions and is thus highly suited for this purpose.

The impact of this thesis is not just limited to the domain of MPEG-7. Since TDOM, typing automata, and the implemented prototype are usable for the representation and management of arbitrary XML documents in principle, the thesis lays the foundations for a new generation of general XML database solutions that exploit available schema and type information for the adequate storage of XML documents. The availability of such solutions is very desirable in any application domain in which data-centric XML documents with large amounts of non-textual data are encountered.

Contents

1 Introduction **1**
- 1.1 Aims . . . 2
- 1.2 Contributions . . . 2
- 1.3 Organization . . . 3
- 1.4 Prerequisites . . . 4

2 MPEG-7 **5**
- 2.1 Metadata for Digital Media . . . 5
- 2.2 Metadata Standards . . . 6
- 2.3 Overview of MPEG-7 . . . 9
- 2.4 MPEG-7 Media Descriptions . . . 10
- 2.5 Basic Observations . . . 12

3 Requirements **15**
- 3.1 Representation of Media Descriptions . . . 15
- 3.2 Access to Media Descriptions . . . 17
- 3.3 Media Description Schemes . . . 20
- 3.4 Extensibility . . . 22
- 3.5 Classic DBMS Functionality . . . 24

4 XML Database Solutions **25**
- 4.1 Native Database Solutions . . . 26
- 4.2 Database Extensions . . . 27

5 Analysis **29**
- 5.1 Representation of Media Descriptions . . . 29
- 5.2 Access to Media Descriptions . . . 32
- 5.3 Media Description Schemes . . . 37
- 5.4 Extensibility . . . 39
- 5.5 Classic DBMS Functionality . . . 40
- 5.6 Summary . . . 42

6 The Typed Document Object Model **44**
- 6.1 Data Models for XML Documents . . . 44
- 6.2 TDOM in Six Points . . . 45

7 Typing **62**
- 7.1 Basic Considerations . . . 63
- 7.2 Typing Automata . . . 67
- 7.3 Computational Complexity . . . 81

7.4 Optimizations 83
7.5 Extensions 90

8 Implementation **104**
8.1 Simple Type Framework 106
8.2 Document Manager 106
8.3 Schema Catalog 107
8.4 Routine Framework 110
8.5 Index Framework 112
8.6 Query Evaluator 114
8.7 Experimental Results 120

9 Conclusion **124**

List of Figures

2.1 Standardized description schemes 7
2.2 Standardized metadata frameworks 8
2.3 Predefined MPEG-7 media description schemes 9
2.4 `Melody` media description scheme 12
2.5 Example MPEG-7 media description 13

4.1 XML database solutions 25

5.1 Analysis results 30

6.1 TDOM representation of XML document structure 46
6.2 Structural representation of an MPEG-7 media description 47
6.3 TDOM representation of elements and attribute values 49
6.4 Typed representation example 50
6.5 Unyped representation example 51
6.6 TDOM simple type framework 54
6.7 Example simple type support 55
6.8 Switching between corresponding representations 58

7.1 Typing problem 64
7.2 Example tree automaton 65
7.3 Tree automaton application 67
7.4 Example element type IDs 69
7.5 Typing automaton structure 70
7.6 Example transition rules 72
7.7 Typing phase 76
7.8 Local document typing 85
7.9 Failing local document 86
7.10 Secondary conditions 92
7.11 Example mapping of unmixed element content declaration 92
7.12 Example mapping of complex type derivation hierarchy 95
7.13 Attribute condition structure 98
7.14 Example mapping of attribute declarations 100

8.1 PTDOM architecture 104
8.2 Simple type framework component 106
8.3 Document manager overview 107
8.4 Schema catalog overview 108
8.5 Routine framework overview 111
8.6 Index framework overview 113

8.7 Query evaluator component overview 115
8.8 PTDOM query algebra overview 116
8.9 Query algebra example 117
8.10 Heuristic XPath translation overview 119
8.11 Results of schema definition import 120
8.12 Results of document import 121
8.13 Results of document querying 122

Chapter 1

Introduction

The Multimedia Content Description Interface (MPEG-7) [ISO01a, ISO99, MSS02, MKP02, Mar02, NL99a, NL99b] is an ISO standardization effort driven by major broadcasting companies, consumer electronics manufacturers, and telecommunication service providers that is targeted at the development of a metadata standard for multimedia content. Starting back in 1996 and having reached a mature state by November 2001, MPEG-7 provides an exhaustive tool set for the detailed description of audiovisual media for a broad variety of applications, ranging from classic multimedia archives, search engines, journalism, education, and entertainment applications to multimedia production support.

Given its strong backing and broad applicability, MPEG-7 is receiving much attention from the multimedia community. As more and more tools and applications producing and processing MPEG-7-compliant media descriptions are emerging (e.g., [FK01, YBL+01, KW01]), the amount of MPEG-7 media descriptions is increasing continuously.

In the light of this development, there will certainly be the need for adequate database support for the management of MPEG-7 media descriptions. Adequate database support including a sophisticated query language, efficient index structures, transactions, concurrency control, access control, reliable means for backup and recovery, etc. would greatly alleviate the implementation of MPEG-7 applications that have to work with larger numbers of media descriptions such as multimedia archives, multimedia search engines, and multimedia production environments. Since MPEG-7 media descriptions are XML documents [BPSMM00] which conform to schema definitions expressed with the XML Schema variant MPEG-7 DDL [TBM+01, BM01, ISO01b], it is a self-suggesting idea to employ XML database solutions for their management, as proposed for instance by [Kos02, FK01].

Closer reflection on this natural idea reveals, however, that the decision to use XML database solutions for the management of MPEG-7 media descriptions is not as simple as it might sound at first glance. For example, it must be considered that MPEG-7 media descriptions largely consist of non-textual data typically covering rather low-level technical aspects of multimedia content, such as frequency spectrums, color distributions, object shapes, object motion vectors, etc. A suitable database solutions should offer adequate means to access and index such non-textual data. But many XML database solutions have a classic document management background and are therefore mainly targeted at the treatment of textual data.

Moreover, there exists a confusing variety of XML database solutions with different degrees of maturity and capabilities: "native" XML database solutions as

well as XML extensions of traditional database management systems, commercial products as well as open source projects and research prototypes [Bou02]. In order to be able to reasonably decide for an XML database solution for the management of MPEG-7 media descriptions, it is necessary to have an overview of current XML database solutions and their capabilities and limitations.

1.1 Aims

The present thesis wants to explore the issue of applying XML database solutions for the management of MPEG-7 media descriptions in detail. An extensive catalog of requirements shall be developed that should be met by any XML database solution employed for the management of MPEG-7 media descriptions. Against this catalog, a representative set of current XML database solutions with commercial, research, and open source backgrounds are to be evaluated. Given a positive outcome of this evaluation, an appropriate XML database solution for the management of MPEG-7 media descriptions shall be suggested. Given a negative outcome, the foundations for a new XML database solution that suits the imposed requirements for the management of MPEG-7 media descriptions shall be layed. A prototype of such a solution shall be designed and implemented.

1.2 Contributions

In pursuing these aims, the thesis makes several substantial contributions:

- The thesis performs an extensive analysis of state-of-the art XML database solutions. It compares a broad array of 21 native XML database solutions and XML extensions of traditional database management systems (DBMSs) including commercial systems, research prototypes, as well as open source projects against a wealth of 18 requirements covering the representation of and the access to MPEG-7 media descriptions, the handling of media description schemes, extensibility, as well as traditional DBMS functionality. An analysis of the capabilities and limitations of current XML database solutions with such an extent and level of detail is unprecedented. The analysis has also been published in [WK03b, WK02b].

- In face of considerable deficiencies unveiled by the analysis of current XML database solutions, the thesis proposes the Typed Document Object Model (TDOM) for XML documents as a foundation for XML database solutions that better suit the needs of MPEG-7 applications. Compared to other data models for XML documents, TDOM is focused on exploiting available schema and type information – as carried by MPEG-7 media description schemes, for example – to represent non-textual XML document contents in a way that is appropriate to the particular content type. On the basis of TDOM, applications can reasonably access and process even complex non-textual data such as frequency spectrums and color distributions often contained in MPEG-7 media descriptions. TDOM has also been published in [WK04, WK02a].

- The thesis proposes typing automata as a formalism for the intermediary representation of schema definitions for XML documents which is independent of any particular XML schema definition language. Typing automata are able to validate XML documents in TDOM representation and to infer and create

appropriate typed representations of their contents. As they are extensible up the expressiveness of MPEG-7 DDL, typing automata constitute an adequate intermediary representation for MPEG-7 media description schemes. Typing automata have also been published in [WK03a].

- The thesis describes the prototypical implementation of the Persistent Typed Document Object Model (PTDOM), an XML database solution developed on top of the object-oriented DBMS ObjectStore that is highly suitable for the management of MPEG-7 media descriptions. PTDOM employs TDOM for the representation of XML documents and features an MPEG-7 DDL-compliant schema catalog based on typing automata. The typing automata maintained by the schema catalog are used to store XML document contents in an appropriately typed manner thereby giving applications adequate access to these contents. Further benefits of PTDOM are its broad support of data types, index structures, and server-side routines, as well as its profound extensibility. The described prototype has also been published in [WK03c].

1.3 Organization

The present thesis is structured as follows: Chapter 2 introduces the domain of metadata for digital media and corresponding metadata standards and gives an overview of the MPEG-7 standard. It interrelates MPEG-7 to other metadata standards and performs some basic observations on the nature of MPEG-7 media descriptions and media description schemes that are relevant for their management in a database. Based on these observations, Chapter 3 presents a comprehensive catalog of requirements that should be fulfilled by an XML database solution employed for the management of MPEG-7 media descriptions. Chapter 4 introduces different types of XML database solutions and presents an extensive and representative set of such solutions on which the investigations of this thesis are restricted. Chapter 5 performs a detailed analysis of these database solutions with regard to the requirements of Chapter 3.

Given the significant weaknesses of current XML database solutions uncovered by the analysis, the thesis continues with developing a new XML database solution that better suits the needs of MPEG-7 applications. As a foundation of such a solution, Chapter 6 proposes, after highlighting the deficiencies of existing XML data models, TDOM as a data model for XML documents that is suitable for the representation of MPEG-7 media descriptions. As another foundation, Chapter 7 proposes typing automata for the intermediary representation of XML schema definitions for the validation and typing of TDOM-represented XML documents. It examines the computational complexity of the behavior of typing automata, suggests optimizations, and proposes enhancements that increase the expressiveness typing automata up to the level of MPEG-7 DDL. Equipped with TDOM and typing automata, Chapter 8 describes the implementation of the PTDOM prototype of an XML database solution that satisfies most of the requirements for the management of MPEG-7 media descriptions given by Chapter 3. Chapter 9 concludes the thesis with a summary of its results and gives an outlook to possible future research work.

1.4 Prerequisites

This thesis assumes basic knowledge of XML and XML Schema. There exist many excellent introductions to this topic, e.g., [LBK02] Chapter 17, [Fal01], and [HS02]. The thesis further assumes basic knowledge of database management systems. For an introduction to this topic, the reader is referred to one of the numerous database text books, such as [LBK02] or [EN94].

Chapter 2

MPEG-7

This chapter gives an introduction to MPEG-7 and takes a closer look on MPEG-7 media descriptions and their characteristics. It begins with an introduction into the subject of metadata for digital media (2.1) and related metadata standards (2.2). This is followed by an overview of the MPEG-7 standard, its aims, anatomy, and position to other metadata standards (2.3). The form of typical MPEG-7 media description schemes and complying descriptions (2.4) is illustrated and several observations concerning basic properties that have to be considered for their management are performed (2.5).

2.1 Metadata for Digital Media

A central problem of digital media management is that digital media formats – such as MPEG-2 for videos, MP3 for audios, and SMIL for multimedia presentations – primarily encode the presentation of content for use by media players but not the information content conveys to consumers. It is more and more recognized that *metadata*, i.e., data that describe media, are the key to an effective media management [CFS99, SK98, KSS95].

Metadata may address different aspects of media [Gil98]: apart from pure *description of media content*, e.g., the score and lyrics of a song or the persons occurring in a video, metadata can also cope with *administrative aspects* of media, e.g., the location of media and copyright information, *preservation aspects*, e.g., the tools with which media where produced and the physical condition of storage media, *technical aspects*, e.g., applied encoding formats and encoding parameters, and *usage aspects*, e.g., user tracking and information about reuse. Rich metadata covering these aspects open up the way to a variety of advanced multimedia applications (see [Gre00, SK98] for illustrative examples) ranging from search engines and archives allowing content-based search and retrieval of media, content-based filtering of media on broadcast channels, (semi-)automated composition support in multimedia authoring, and automatic generation of personalized content in accordance with user interests to administrative support for the media production process.

In the literature, several classification schemes for metadata for digital media have been elaborated [BKS98, Gil98, KSS95, BR94]. Though diverse in detail and bias, the proposed schemes agree on a common core of basic characteristics along which metadata can be categorized: firstly, metadata can be classified according to the *level* on which they describe media. Media description may occur on a *technical*

level or on a *semantic* level. Metadata on a technical level constitute technical features describing lower-level aspects of media, such as frequency spectrums for audios and color distributions for images. Technical features are effective for the realization of similarity searches on media collections. Metadata on a semantic level are concerned with the information media conveys on a higher level of abstraction. Examples would be the score of a song, the persons occurring in a video, etc. As such metadata are close to human thinking, they are very valueable for the realization of content-based access.

Secondly, metadata can be classified according to their *producibility*: production of metadata can either be *automatic* or it may require *manual* human intervention. As human intervention for the production of metadata may be costly, automatic producibility is a very desirable property of metadata from an economic point of view. It is noteworthy that the level of metadata affects their producibility: low-level, technical metadata can usually be generated automatically from media with little effort while semantic metadata describing the information conveyed by media content typically require domain knowledge to be brought in by humans.

Thirdly, metadata can be categorized according to *dependencies*: metadata can be *domain-dependent*. For example, the position of a tumor is likely to be of use for medical applications only whereas the color distribution of an image may be useful for a variety of application domains. Also, metadata can be *media type-dependent*. For instance, color distributions apply to visual media only while the creation date is applicable to any type of media.

2.2 Metadata Standards

In the wake of increasing awareness of the importance of metadata for digital media management, a multitude of metadata standardization initiatives have been called into being throughout the recent years. Indeed, standardization of metadata for digital media bears several appealing prospects: metadata standards allow content providers to engage in electronic business by sharing standardized descriptions of their media with other providers. Standardized media descriptions also enable third parties to establish value-added services spanning multiple providers like media search engines or market places. Finally, metadata standards pave the way to interoperable commercial off-the-shelf software for the creation and processing of media descriptions such as media annotation and automated metadata extraction tools.

Coarsely, two classes of metadata standards can be distinguished: *standardized description schemes* and *standardized metadata frameworks*. Standardized description schemes comprise the many standardization efforts of various communities that are targeted at the definition of fixed, ready-to-use attribute sets for the common description of media tailored to the needs of the communities' particular application domains together with appropriate formats for exchanging descriptions.

The table of Figure 2.1 presents a selection of seven influential and frequently cited representatives of this category of metadata standards: Machine-Readable Cataloging (MARC) [Net02] and Dublin Core Metadata for Resource Discovery (Dublin Core) [DCM99] for the bibliographic description of arbitrary media; Categories for the Description of Works of Art (CDWA) [AIT00] and VRA Core Categories [VRA02] for the description of artworks and images depicting artworks; Content Standard for Digital Geospatial Metadata (CSDGM) [Met98] for the description of geographic media and data sets; the Data Dictionary (Z39.87) [NIS02]

		Standardized description schemes						
		MARC	Dublin Core	CDWA	VRA Core Categories	CSDGM	Z39.87	LOM
General information	Standardization body	Library of Congress	Dublin Core Metadata Initiative (DCMI)	Art Information Task Force (AITF)	Visual Resources Association (VRA)	Federal Geographic Data Committee (FGDC)	National Information Standards Organization (NISO)	IEEE Learning Technoglogy Standards Committee (LTSC)
	Year	Late 60's – current version MARC 21 since 1999	1998 – current Version 1.1. since 1999	Mid 90's – current Version 2.0 since 2000	Current Version 3.0 since 2002	1994 – updated version since 1998	2002	2002
Characteristics of standardized metadata	Domain	Bibliographic media description	Bibliographic media description	Description of artworks	Description of images of artworks	Description of geographic media	Description of still images	Description of educational media
	Media types	Any	Any	Any	Images	Any	Images	Any
	Level	Largely semantic	Largely semantic	Largely semantic	Largely semantic	Semantic and technical	Technical	Largely semantic
	Producibility	Mainly manual	Mainly manual	Mainly manual	Mainly manual	Manual and automatic	Mainly automatic	Mainly manual

Figure 2.1: Standardized description schemes

for the technical description of images taken with digital cameras or digitized with scanners; and Learning Object Metadata (LOM) [IEE02] for the description of educational material.

Accompanied by some general information, the table classifies these standardized description schemes according to the basic characteristics of the metadata they define. One can observe that the standardized description schemes treat media with regard to the respective application domains largely on a semantic level. Most of the high-level metadata defined by these schemes will have to be manually produced by human catalogers. The exceptions to this are CSDGM and Z39.87 which define significant amounts of low-level technical metadata that are automatically extractable. One can furthermore observe that the applicability of the depicted standardized description schemes – except VRA Core Categories and Z39.87 which focus on image description – is usually not restricted to any specific media types. On that score, however, it should be considered that most of the description schemes pursue a coarse-grained look onto media: they do not open up the internal, type-dependent structure of media for the detailed description of individual media parts.

There are some problems concerning the use of standardized description schemes. As they constitute fixed attribute sets highly-specialized on certain application domains, they are difficult to apply to other domains or to adapt to the needs of a particular application. Moreover, the data models and exchange formats that underly the various standardized description schemes differ so that the combination of several description schemes or the realization of applications supporting more than one description scheme is complicated.

In the light of these difficulties, standardized metadata frameworks have evolved as another class of metadata standards. Metadata frameworks do not predefine any domain-specific description schemes. Instead, they provide rich generic data models for media descriptions together with schema definition languages allowing to define description schemes for arbitrary application domains as well as exchange formats for both description schemes and complying media descriptions. Standardized metadata frameworks thus constitute common foundations for media description that can

be flexibly tailored to the particular needs of an application domain or an application. They promise simpler interoperability between as well as simpler combination of different description schemes and permit the creation of generic tools for the processing of media descriptions.

		Standardized metadata framework				
		PICS	MCF	RDF	Topic Maps	XTM
General information	Standardization body	World Wide Web Consortium (W3C)	World Wide Web Consortium (W3C)	World Wide Web Consortium (W3C)	International Organization for Standardization (ISO)	TopicMaps.org Consortium
	Year	1996	1997	1999	2000	2001
	Data model	Attribute-based	Semantic network-based/ directed labeled graphs	Semantic network-based/ directed labeled graphs	Semantic network-based/ labeled graphs	Semantic network-based/ labeled graphs
	Schema definition language	Rating Systems	Standard Vocabulary	RDF Schema	Topic Map Constraint Language (TMCL) under development	Published Subject Identifiers, Topic Map Constraint Language (TMCL) under development
Characteristics of standardized metadata	Domain	Qualitative rating of web resources	Machine-processable semantic description of web resources	Machine-processable semantic description of web resources	Navigatable, semantic mapping of information resources	Navigatable, semantic mapping of web resources
	Level	Semantic	Semantic	Semantic	Semantic	Semantic
	Media types	Any	Any	Any	Any	Any

Figure 2.2: Standardized metadata frameworks

Figure 2.2 presents a selection of five representative standardized metadata frameworks: the Platform for Internet Content Selection (PICS) [KMRT96] for the qualitative rating of web resources; the Resource Description Framework (RDF) [LS99] and its precursor Meta Content Framework (MCF) [GB97] aimed at the machine-processable semantic description of web resources; Topic Maps [ISO00] and their close relatives XML Topic Maps (XTM) [PM01] targeted at the navigatable mapping of information and web resources according to their semantics.

Along with some general information regarding the data models and schema definition languages supported, the figure classifies the depicted standardized metadata frameworks according to basic characteristics of the metadata that can be created with them (producibility is neglected in the figure because it depends on the description schemes that are actually defined with the schema definition languages). It is noteworthy that all frameworks are primarily targeted at describing media at a semantic level in very broad application domains. The data models offered by the frameworks do not support more intricate numeric data like vectors or matrices that might be necessary to reasonably represent complex technical metadata such as color distributions or object shapes. It can be furthermore observed the frameworks generally permit the description of arbitrary types of media. However, it should be considered that this independency of media types is the result of maintaining largely a coarse-grained view on media.

2.3 Overview of MPEG-7

The Multimedia Content Description Interface (MPEG-7) [ISO01a, MSS02, MKP02, Mar02, NL99a, NL99b] is an ISO metadata standard aimed at providing a means for the sophisticated description of multimedia content that is of use for a broad spectrum of applications, including – but not limited to – multimedia archives, search engines, media production support, education, and entertainment.

To achieve the ambitious goal of broad applicability, MPEG-7 essentially standardizes two things: the *Description Definition Language* (MPEG-7 DDL) [ISO01b] for the definition of schemes for the description of media, and, defined with MPEG-7 DDL, a comprehensive set of *media description schemes* that are expected to be useful for many applications.[1] The predefined media description schemes comprise schemes for describing visual [ISO01c] and audible media [ISO01d] and multimedia description schemes that are of general use [ISO01e]. With these description schemes, MPEG-7 permits an extensive description of multimedia content and parts of multimedia content not only on a technical, feature-oriented level but also on a high semantic level. Using MPEG-7, for example, one can describe the frequency spectrum of a song recording as well as the song's lyrics and musical score – all within a single media description.

Standardized MPEG-7 media description schemes		
Visual	Audio	Multimedia
Color: • color space • dominant colors • color quantization • ...	Audio framework: • audio waveform • audio power • audio spectrum • ...	Content management: • creation information • creation tool • creator • ...
Texture: • edge histogram • homogeneous texture • texture browsing • ...	Timbre: • harmonic instrument timbre • percussive instrument timbre • ...	Content semantics: • classification scheme • text annotation • graph • ...
Shape: • object region-based shape • contour-based shape • 3D shape • ...	Sound recognition and indexing: • sound model • sound classification model • sound model state path • ...	Navigation and summarization: • hierarchical summary • visual summary component • audio summary component • ...
Motion: • camera motion • object motion trajectory • motion activity • ...	Melody: • melody contour • melody sequence • ...	Content organization: • collection • classification model • cluster model • ...
Localization: • region-locator • spatio-temporal locator • ...	Spoken content: • spoken content lattice • spoken content header • ...	User interaction: • usage history • user preferences • ...

Figure 2.3: Predefined MPEG-7 media description schemes

Figure 2.3 gives an overview of some of the media description schemes shipping with MPEG-7. As it can be observed, the description schemes profoundly cover the

[1]MPEG-7 originally distinguishes between descriptors, which are basic descriptive features of media, and description schemes, which are more complex descriptional units made up of other description schemes and descriptors. In practice, however, this distinction is rather arbitrary: descriptors may have a complex structure as well. For simplicity, both descriptors and description schemes are thus referred to as media description schemes in the following.

different aspects that can be addressed by metadata for digital media: there exist plenty of schemes for describing audiovisual content on technical and semantic levels, schemes for the management and organization of content addressing administrative, preservation, and technical aspects of media, as well as schemes addressing usage aspects of media such as user preferences and usage history.

Applications do not need to stick to the predefined media description schemes. They can flexibly create new description schemes with MPEG-7 DDL, either from scratch or by extending or combining existing schemes. This inherent extensibility further contributes to the standard's aim of broad applicability.

Relating MPEG-7 to other metadata standards available for digital media makes clear why MPEG-7 is so attractive to the multimedia community. As MPEG-7 defines a collection of ready-to-use schemes for the description of media, it can be classified into the group of standardized description schemes. In contrast to other standardized description schemes like MARC, CDWA, and LOM, however, MPEG-7 has not been developed with a restricted application domain in mind but is intended to be applicable to a wide range of multimedia applications. Furthermore, the wealth of media description schemes predefined with MPEG-7 allows to describe multimedia content or parts of multimedia content with an unprecedented degree of detail not only on a semantic but also on a technical level while not unduly restricting the supported media types: the standard applies to arbitrary audiovisual material. In comparison to other standardized description schemes, it is also interesting to observe that many of the media description schemes offered by MPEG-7 – especially those addressing the technical level of media – are automatically producible. But what distinguishes MPEG-7 most from other standardized description schemes is that it can be tailored to the needs of a specific application domain or application by means of MPEG-7 DDL allowing to define new or to extend existing media description schemes.

With the ability to define media description schemes by means of MPEG-7 DDL, MPEG-7 can be equally regarded as a standardized metadata framework. However, MPEG-7 distinguishes itself from other standardized metadata frameworks like RDF and Topic Maps not only in that it already comes with an extensive set of ready-to-use media description schemes. Also, MPEG-7 is targeted at the semantic description of media as well as at their technical description. Consequently, it provides adequate support for intricate numeric data necessary for the representation of more complex technical metadata like color histograms, shapes, etc.

2.4 MPEG-7 Media Descriptions

MPEG-7 is strongly committed to XML [BPSMM00] and related standards. MPEG-7 DDL, the language used for the definition of media description schemes, is a superset of XML Schema [TBM+01, BM01], the W3C schema definition language for XML documents. Certain extensions to XML Schema were considered neccessary to better cope with the peculiarities of multimedia data. In particular, support for array and matrix data types as well as additional temporal data types were added to XML Schema.

Being an extended XML Schema, MPEG-7 DDL can be considered as another general-purpose schema definition language for XML documents. Since MPEG-7 media descriptions instantiate media description schemes defined with MPEG-7 DDL, an MPEG-7 media description is consequently an XML document that is valid with regard to the MPEG-7 DDL schema definition containing the media

description scheme to which the description complies.

Within an MPEG-7 DDL schema definition, a media description scheme is defined by means of a complex type. Complex types, a concept already introduced by XML Schema, essentially constitute named complex content models. A complex types can be referenced by an element type declaration in order to specify the elements and attribute values that are valid to appear as the contents of the elements instantiating the element type declared.

An important feature of complex types of which MPEG-7 makes extensive use is that they can be derived from each other – either by extending the content model of a base type with further elements or attribute values or by restricting the content model of the base type to a limited subset. Every MPEG-7 media description scheme is directly or indirectly derived from the complex type `DSType` predefined by MPEG-7 (or from the complex type `DType` in case that a media description scheme merely constitutes a descriptor). In that manner, MPEG-7 media description schemes are organized in a deeply nested derivation hierarchy.

Apart from the pure aesthetics of letting related media description schemes share common characteristics via common base types, organizing MPEG-7's media description schemes in a deep complex type derivation hierarchy permits the flexible combination of different description schemes. By complex type derivation, MPEG-7 DDL enables a simple form of polymorphism within XML documents: the content of an element does not necessarily need to be filled in according to the content model given by the complex type referenced by its element type declaration; the element can also be filled in according to the content model of a complex type derived from the referenced type as long as the derived type used is declared by augmenting the element with an additional `xsi:type` attribute value.

Figure 2.4 illustrates the definition of media description schemes with MPEG-7 DDL. The figure shows a slightly simplified fragment of the `Melody` media description scheme [ISO01d], a representative scheme for the description of melodies that can be employed for the realization of query-by-humming applications.

The `Melody` media description scheme is defined by the complex type `MelodyType` to the upper left of the figure. By extending the predefined type `AudioDSType` which in turn extends the predefined type `DSType`, it is expressed that `MelodyType` defines an MPEG-7 audio description scheme.

The declaration of `MelodyType` states that a melody can be described by its meter and its melody contour using optional elements of type `Meter` and `MelodyContour`. The meter of a melody, according to the complex type `MeterType` to the upper right of the figure providing the content model for `Meter` elements, is a fraction consisting of numerator and denominator (element types `Numerator` and `Denominator`). There is the restriction that the numerator must be an integer value in the interval from 1 to 128, while the denominator must be a power of two in the same interval.

A melody contour is divided into a contour and a beat. This is expressed with the element types `Contour` and `Beat` given by the complex type `MelodyContourType` in the lower left column which defines the content model for `MelodyContour` elements. Since `MelodyContourType` is derived from `AudioDSType`, it also defines an MPEG-7 audio description scheme, i.e., the `MelodyContour` media description scheme. The contour of a melody is a list of integer values giving a measure for the distance between every two consecutive notes of a melody while the beat is a list of integer values associating every note of the melody with its position in the beat.

Finally, the figure shows the declaration of the element type `AudioDescriptionScheme` to the lower right. The contents valid for ele-

```
...

<complexType name="MelodyType">
  <complexContent>
    <extension base="mpeg7:AudioDSType">
      <sequence>
        <element name="Meter"
          type="mpeg7:MeterType"
          minOccurs="0"/>
        <element name="MelodyContour"
          type="mpeg7:MelodyContourType"
          minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<complexType name="MelodyContourType">
  <complexContent>
    <extension base="mpeg7:AudioDSType">
      <sequence>
        <element name="Contour">
          <simpleType>
            <list itemType="integer"/>
          </simpleType>
        </element>
        <element name="Beat">
          <simpleType>
            <list itemType="integer"/>
          </simpleType>
        </element>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

```
<complexType name="MeterType">
  <complexContent>
    <extension base="mpeg7:AudioDType">
      <sequence>
        <element name="Numerator">
          <simpleType>
            <restriction base="integer">
              <minInclusive value="1"/>
              <maxInclusive value="128"/>
            </restriction>
          </simpleType>
        </element>
        <element name="Denominator">
          <simpleType>
            <restriction base="integer">
              <enumeration value="1"/>
              <enumeration value="2"/>
              <enumeration value="4"/>
              <enumeration value="8"/>
              <enumeration value="16"/>
              <enumeration value="32"/>
              <enumeration value="64"/>
              <enumeration value="128"/>
            </restriction>
          </simpleType>
        </element>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<element name="AudioDescriptionScheme"
  type="mpeg7:AudioDSType"/>

...
```

Figure 2.4: `Melody` media description scheme

ments of that type are given by the complex type `AudioDSType`. Since every audio description scheme predefined by MPEG-7 is ultimately derived from `AudioDSType`, `AudioDescriptionScheme` elements can hold descriptions complying to any audio description scheme by exploiting complex type polymorphism via the `xsi:type` attribute. In particular, an `AudioDescriptionScheme` element can be filled according to both the `Melody` and `MelodyContour` media description schemes if `xsi:type` attribute values are provided addressing the complex types `MelodyType` and `MelodyContourType`, respectively.

Figure 2.5 depicts an MPEG-7 media description complying to the `Melody` media description scheme. The description covers a small fraction of the melody of the song "Moon River" by Henry Mancini (taken from [ISO01d], page 101). An `AudioDescriptionScheme` element constitutes the entry point to the description whose content is marked to be compliant to the `Melody` media description scheme by means of an `xsi:type` attribute value.

2.5 Basic Observations

Several elementary characteristics of the MPEG-7 standard can be observed that should be considered for the management of MPEG-7 media descriptions:

The first very basic observation is that MPEG-7 media description schemes are schema definitions for XML documents written in the MPEG-7 DDL schema def-

```
<!-- Melody description of 8 notes taken from "Moon River" by Henry Mancini -->

<AudioDescriptionScheme xmlns="http://www.mpeg7.org/..."
                        xmlns:xsi="http://www.w3.org/..."
                        xsi:type="MelodyType">
    <Meter>
        <Numerator>3</Numerator>
        <Denominator>4</Denominator>
    </Meter>
    <MelodyContour>
        <!-- Distance between two notes -->

        <Contour>2 -1 -1 -1 -1 -1 1</Contour>

        <!-- Beat position of notes -->

        <Beat>1 4 5 7 8 9 9 10</Beat>
    </MelodyContour>
</AudioDescriptionScheme>
```

Figure 2.5: Example MPEG-7 media description

inition language; MPEG-7 media descriptions are XML documents valid to these schema definitions. Even though this observation appears banal at first glance, it unveils a fundamental characteristic of the standard that has considerable implications on the management of MPEG-7 media descriptions: MPEG-7 does not define an XML-independent conceptual model for MPEG-7 media descriptions that could serve as a basis for the development of an MPEG-7 database schema. By defining media description schemes with a schema definition language for XML documents, MPEG-7 instead coalesces the XML representation of media descriptions with their conceptual model. As a consequence of this inseparability of XML representation and conceptual model, managing MPEG-7 media descriptions effectively means managing XML documents.

Of course, an XML-independent conceptual model for MPEG-7 media descriptions could be reverse-engineered from the media description schemes predefined with the standard. But considering the multitude and complexity of the standardized media description schemes, this constitutes an effort that is likely to be prohibitive in practice unless limited to a small subset of the predefined schemes.

As a second observation, one can find that the set of media description schemes supported by MPEG-7 is not fixed. The standard already comes with comprehensive ready-to-use media description schemes for audiovisual content but applications are always permitted to create new, to extend, or to recombine existing description schemes with MPEG-7 DDL whenever necessary. Since MPEG-7 DDL is a general-purpose schema definition language for XML documents allowing to mask almost arbitrary XML schema definitions as MPEG-7 media description schemes, this observation implies that a generally applicable solution for the management of MPEG-7 media descriptions cannot be a solution that is just capable of managing XML documents complying to the predefined media description schemes. Instead, it has to be a solution capable of managing arbitrary XML documents.

The inherent extensibility of MPEG-7 is another argument against the idea of reverse-engineering a generally applicable conceptual model for MPEG-7 media descriptions as the foundation for a dedicated MPEG-7 database schema: such a model will inevitably neglect application-specific media description schemes since these are typically not known in advance.

Thirdly, as illustrated before, MPEG-7 makes heavy use of complex type derivation to organize the standardized media description schemes in a deep derivation hierarchy and extensively exploits this derivation hierarchy for the flexible combination of description schemes by means of complex type polymorphism. A management of MPEG-7 media descriptions will have to deal with the complexity introduced by both concepts.

Finally, it can be observed that much of the information encoded within typical MPEG-7 media descriptions is not of a textual nature. Large portions of the information consist of numbers and complex numeric structures such as vectors and matrices usually representing technical metadata. The `Melody` media description scheme presented earlier in Figure 2.4 is a prime example for this observation since the information defined by that scheme is solely of non-textual nature, such as numbers for measuring the meter of a song and lists of integer values for capturing the song's contour and beat. As a matter of fact, more than 80% of the media description schemes predefined by MPEG-7 for the description of visual and audible content in [ISO01c] and [ISO01d] consist primarily of non-textual data. Thus, a management of MPEG-7 media descriptions must be prepared to adequately handle large amounts of complex, non-textual data.

Chapter 3

Requirements

Given its broad applicability, inherent extensibility, and the wealth of predefined media description schemes, MPEG-7 can be expected to face wide-spread use in a large variety of future multimedia applications. Soon, there will be the need for adequate solutions for managing MPEG-7 media descriptions.

As observed previously, MPEG-7 media descriptions are XML documents complying to schema definitions written in MPEG-7 DDL. Considering the lack of an XML-independent conceptual model for MPEG-7 media descriptions and the difficulties to reverse-engineer such a model, the problem of managing MPEG-7 media descriptions in general constitutes a problem of managing XML documents. Thus, the idea to employ XML database solutions lies close at hand. This chapter presents requirements such solutions should satisfy in order to be suitable for the management of MPEG-7 media descriptions.

General requirements for XML database solutions have already been addressed in the literature [ST01]. This chapter, however, does not take an abstract XML database management viewpoint. Instead, outgoing from the basic observations about MPEG-7 in Chapter 2, it derives and motivates requirements for XML database solutions from the concrete perspective of applications that need adequate database support for MPEG-7 media descriptions. These certainly include requirements that are specific to the management of MPEG-7 media descriptions and do not necessarily apply to the management of other kinds of XML documents. Nevertheless, these equally include general requirements that should apply to the management of almost any kinds of XML documents.

For the subsequent presentation, requirements are organized into requirements concerning the representation of media descriptions (3.1), the access to media descriptions (3.2), media description schemes (3.3), extensibility (3.4), and classic database management system (DBMS) functionality (3.5).

3.1 Representation of Media Descriptions

The basic characteristics of MPEG-7 media descriptions considerably affect the way they should be represented in an XML database solution. As it will be set out in the following, the nature of MPEG-7 especially calls for the *fine-grained representation* of a media description's structure as well as for the *typed representation* of its basic contents.

Fine-grained representation MPEG-7 allows the definition of arbitrarily complex media description schemes with MPEG-7 DDL, capturing media content from possibly very different points of view and levels of abstraction. Regarding this potential complexity, typical applications cannot be expected to process the full scope of an MPEG-7 media description. Instead, applications will likely process only those parts necessary to achieve their particular tasks. For instance, an MPEG-7-based query-by-humming song retrieval engine will be mainly interested in melody contours kept within `MelodyContour` elements as defined by the `Melody` media description scheme; it will likely ignore other elements that might be contained in an MPEG-7 media description treating other aspects of the song.

As a result, it is important for an XML database solution used for the management of MPEG-7 media descriptions to store and represent the structure of such descriptions with fine granularity. A fine-grained storage representation that in detail models the hierarchy of elements and attribute values of which a media description consists allows a database solution to efficiently provide applications with exactly those parts of a description that they are interested in. The price to pay is that effort must be spent for the decomposition of media descriptions when they are imported into a database and for their reassembly when they are exported.

If a database solution represents MPEG-7 media descriptions as coarse-grained, unstructured objects, in contrast, it must load, parse, and decompose the descriptions every time they are accessed by applications. Apart from performance concerns regarding the overhead if applications want to access small fractions of descriptions only as it is likely in typical MPEG-7 application scenarios, a coarse-grained storage representation of media descriptions hinders not only the realization of fine-grained updates and concurrency control but also fine-grained access control – also very desirable properties for the management of MPEG-7 media descriptions.

A variety of data models for XML documents that could be used for the fine-grained representation of MPEG-7 media descriptions have been proposed in the literature, e.g., [JLS99, GMW99, GSN99, SYU99]. Moreover, several fine-grained data models for XML documents have appeared in the context of standardization efforts, such as the DOM Structure Model which is specified along with the DOM API by the DOM standard [LLW+00], the XPath Data Model which provides the basis for the evaluation of XPath expressions [CD99], the XML Information Set [CT01], and the XQuery 1.0 and XPath 2.0 Data Model [FMN02], a working draft currently being defined in connection with the XQuery standardization effort for a common XML query language [BCF+02].

Typed representation As already exemplified by the means of the `Melody` media description scheme in Chapter 2, large portions of the information contained in MPEG-7 media descriptions typically consist of non-textual data. Since MPEG-7 media descriptions are XML documents and, as such, a form of text documents, all these data are, self-evidently, encoded as text.

While this might be appropriate for the platform-independent exchange of media descriptions, the textual representation of non-textual data is inadequate for the storage of media descriptions within a database. Usually, textual representations of non-textual data not only consume more storage space than corresponding binary representations; typically, they are also less efficient and more complicate to handle. It is plausible, for example, that handling the list of integer values constituting the content of the `Contour` element in Figure 2.5 on the basis of the depicted textual representation – i.e., using string operations – is rather cumbersome compared to a data structure more appropriate for lists, e.g., an array. Finally, the textual

representation of non-textual information is not always adequate to the semantics of the data. As an example, the alphanumeric order of the textual representation of integer values differs from their inherent numeric order. This complicates, for instance, meaningful indexing of integer values.

As a consequence, a suitable XML database solution should keep the basic contents of an MPEG-7 media description – more precisely, simple element content and the content of attribute values occurring in a media description – in typed representation and not just as text. Typed representation means that these contents are encoded in data structures that suit the particular content type. In this regard, an eligible solution should support the rich set of simple data types predefined by MPEG-7 DDL [ISO01b, BM01] as well as the numerous derivation methods for simple types with which MPEG-7 DDL allows to flexibly derive new simple types from existing ones.

A database solution that keeps non-textual contents of MPEG-7 media descriptions as text, in contrast, burdens applications with the responsibility to constantly and explicitly convert accessed non-textual contents to representations better suiting their types for further processing. Apart from that this is cumbersome and error-prone, the sheer number of conversions necessary during the processing of media descriptions might induce a performance overhead that can pile up considerably.

Some of the data models for XML documents that have been proposed in the literature, e.g., [GMW99, GSN99], support – at least to some degree – the typed representation of a document's basic contents and could therefore be suitable foundations for representing MPEG-7 media descriptions. Among the standard data models for XML documents, the current working draft of the XQuery 1.0 and XPath 2.0 Data Model to some extent permits typed representation of simple element content and the content of attribute values as well.

3.2 Access to Media Descriptions

A fundamental service of an XML database solution used to store MPEG-7 media descriptions is to provide applications with adequate means for accessing such descriptions contained in a database. In this section, it is argued that a suitable solution should allow *fine-grained access* to the structure of a media description, *typed access* to the basic contents carried by the description, and *fine-grained updates* of media descriptions. Moreover, the availability of sophisticated index structures enabling efficient access even to large collections of media descriptions is considered an important requirement. In the following, it is particularly demanded that a suitable XML database solution provides powerful *value index structures*, *text index structures*, and *path index structures*.

Fine-grained access As explained before, MPEG-7-based applications will typically not process the full scope of a media description but rather selectively access only those parts necessary to fulfil their particular tasks. An XML database solution storing MPEG-7 media descriptions should therefore not only represent the structure of media descriptions with a fine granularity; naturally, it should also provide applications with adequate means to perform fine-grained, selective access to these descriptions. With an XML database solution that does not provide means for fine-grained access, applications have to tediously load entire media descriptions for each access and then find and navigate to those parts inside a description that they are actually interested in themselves.

The spectrum of possible means for fine-grained access to the constituents of an MPEG-7 media description ranges from APIs like the DOM API [LLW+00] that facilitate the programming of navigational access to XML documents to declarative query languages for XML documents. Several XML query languages have been proposed in the literature, such as XML-QL [DFF+98] and Quilt [CRF00] to name two prominent examples. Furthermore, XML query languages with varying degrees of expressivenes have been created in the context of standardization efforts like XPath expressions [CD99], XQL [RLS98], and XQuery [BCF+02].

The tradeoff between navigational APIs and declarative query languages for database access is long since understood and also applies to XML database solutions: navigational APIs are a low-level yet powerful means that allow to program any desired form of database access but leave it completely up to the programmer to manually perform optimizations; query languages are a high-level, declarative means that allow to compactly specify database access and enable automatic optimization by a query optimizer but are constrained in expressiveness.

Typed access Non-textual information making up significant portions of typical MPEG-7 media descriptions should be accessible to applications in a way that is appropriate for the particular data type. An XML database solution allowing only textual access to the basic contents of media descriptions seriously hinders applications from accessing non-textual contents of such descriptions in an adequate manner. For example, accessing the list of integer values making up the content of the `Contour` element of Figure 2.5 as a string has considerable drawbacks: apart from that there is no indication that this string actually represents a list of integer values, an application must either access the list by means of string operations – which is not adequate – or explicitly cast the string to a list of integer values before processing – which is cumbersome, error-prone and implies an overhead.

It is therefore highly desirable that an XML database solution gives typed access to the basic contents of an MPEG-7 media description: simple element content and the content of attribute values should be accessible via appropriate typed representations and not as text. Also, a rich set of type-specific operations should be provided so that these contents can be reasonably processed. The type-specific operations offered should not only cover the simple types predefined with MPEG-7 DDL but also those simple types that can be constructed within media description schemes using simple type derivation methods. Picking up once more the example list of integer values, reasonable type-specific operations could be functions for accessing and retrieving the single elements of the lists or for querying list's size.

An extensive set of type-specific operations for typed access within an XML query language has been recently proposed in the context of the XQuery standardization effort [MMRW02]. The proposed operations, however, only cover a restricted subset of predefined simple types and simple type derivation methods of XML Schema and would have to be extended in order to be fully adequate to MPEG-7, e.g., with matrix operations.

Fine-grained updates Multimedia content production can be seen as a repetitive process in which media descriptions, just as the content they describe, are continuously evolving and subject to change. As a consequence, an XML database solution should provide adequate means for updating MPEG-7 media descriptions.

The call for fine-grained access to media descriptions is thus extended from mere read access to cover fine-grained update operations. If a database solution does not support fine-grained updates of media descriptions, a change of (a potentially small

fraction of) a media description must be performed by unloading the complete description from the database, modifying it outside the database, and reinserting it back into the database. Depending on the extent of the update, this can be inefficient and might unnecessarily hamper concurrent access.

As for read access, the span of available means to support fine-grained updates ranges from DOM-like APIs allowing to program updates of XML documents to declarative update languages. Compared to the amount of available XML query languages, however, the domain of declarative update languages for XML documents has largely been neglected so far. An extension of XQuery with update functionality has been proposed by [TIHW01] and the XML:DB Initiative is currently standardizing the XUpdate language [LM00].

Value index structures To facilitate the realization of efficient multimedia retrieval applications on the basis of MPEG-7 media descriptions kept in an XML database solution, the availability of value index structures is indispensable. Even for a large number of media descriptions, a value index maintained on the elements of a given type or on the values of a given attribute permits the efficient lookup of all those out of the indexed elements or attribute values whose contents fulfil a certain criteria. For instance, a value index defined on the `Numerator` and `Denominator` elements declared by the `Melody` media description scheme could help an application to immediately find all song descriptions with a beat of $\frac{3}{4}$. Without value indexing support, the same application would have to access every media description contained in the database and check the beat itself.

The value index structures supported by an XML database solution should at least include classic one-dimensional value index structures such as B-Trees that have long proved their effectiveness for traditional database applications. However, MPEG-7 media descriptions often carry complex multimedia data that cannot be effectively indexed by one-dimensional value index structures. What are reasonable "less than" and "equal to" relations according to which, e.g., the `Contour` elements defined by the `Melody` media description scheme could be organized in an ordered, one-dimensional index structure like a B-Tree? In this case, indexing according to topological relations is more appropriate: for instance, a query-by-humming application needs to efficiently find out whether the contour of a melody fragment that has been hummed by a user is "contained" in the melody contour of a song that is stored in the database. Multidimensional value index structures [GG98] like R-Trees or extended k-d-Trees can support such queries. Therefore, an XML database solution suitable for MPEG-7 should also come with multidimensional *index structures to permit adequate indexing of multimedia data.*

Text index structures So far, it has been stressed that non-textual content contributes significantly to the content of typical MPEG-7 media descriptions. Nevertheless, a considerable portion of the content still consists of textual information. To support the realization of versatile search engines for songs, for example, it would make perfect sense to capture not only the melody of a song using the `Melody` media description scheme but also the song's lyrics.

To facilitate efficient retrieval of elements and attribute values according to textual content, e.g., to retrieve all songs with lyrics containing a certain phrase, an XML database solution should therefore, in excess to conventional value index strucures, have sophisticated text index structures at the disposal. Suitable structures for text indexing [FBY92] are well-known from the domain of information retrieval, e.g., inverted files and PAT-Trees.

Path index structures Since applications seldom process complete MPEG-7 media descriptions but rather selected parts of media descriptions only, applications accessing an MPEG-7 database will often need to efficiently extract these parts from the descriptions stored in the database. The naive approach, i.e., accessing every media description in the database and traversing the description to reach the parts of interest employing the means for fine-grained access offered by the underlying XML database solution, might turn out to be inefficient for large databases. Not only could the traversal times for all accessed descriptions pile up prohibitively for complex traversals. Also, media descriptions will be accessed unnecessarily if they do not contain the part of interest because, for instance, it has been declared optional in the media description scheme.

To reduce traversal times of frequently followed access paths and to prevent unnecessary access to media descriptions, the availability of structures for path indexing is an essential requirement for an XML database solution applied for the storage of MPEG-7 media descriptions. Applicable path index structures are known from the domains of object-oriented databases, e.g., Multiindexes [BK89], semi-structured databases, e.g., DataGuides [GW97] and T-Indexes [MS99], and XML databases, e.g., extended Access Support Relations [FM00] and SphinX [PH01].

3.3 Media Description Schemes

Media description schemes specifying the allowable forms of media descriptions are a central concept of the MPEG-7 standard. An XML database solution managing MPEG-7 media descriptions should therefore provide an *MPEG-7-DDL-compliant schema catalog* for the storage of media description schemes. Based on this catalog, a suitable solution should exploit media description schemes for the *validation of media descriptions*, for the *inference of typed representations*, and for *access optimization.*

MPEG-7-DDL-compliant schema catalog It has been discussed in Chapter 2 that MPEG-7 provides the schema definition language MPEG-7 DDL for the specification of media description schemes which define the allowable structure of media descriptions and the types of their basic contents. It should be no surprise that this information is valuable for the effective management of MPEG-7 media descriptions. Also, this information is of value for applications that require knowledge of the nature of media descriptions, such as editors for media descriptions.

To facilitate the utilization of schema and type information carried in media description schemes, an XML database solution adequate to MPEG-7 should provide an MPEG-7-DDL-compliant schema catalog for the management of the media description schemes to which the media descriptions contained in a database comply. For this purpose, the catalog should be able to parse media description schemes written in MPEG-7 DDL, to check their correctness and integrity, and to bring them into an appropriate representation for later utilization.

A schema catalog based on an XML schema definition language different from MPEG-7 DDL might also be suitable for the management of MPEG-7 media description schemes. In such a case, media description schemes have to be translated to the catalog's schema definition language prior to storage. The suitability of the schema catalog for the management media description schemes is thus determined by the expressiveness of its schema definition language compared to MPEG-7 DDL.

An analysis of XML schema definition languages [LC00] has revealed, however,

that XML Schema – being a large subset of MPEG-7 DDL – already exceeds the expressiveness of other common schema definition languages like DTDs [BPSMM00], XDR [FT98], DSD [Møl03], and Schematron [Jel02], especially with regard to complex type derivation, complex type polymorphism, and supported simple types – concepts on which MPEG-7 heavily relies. As a consequence, MPEG-7 media description schemes cannot be reasonably translated to these schema definition languages in general. And even though compliance to XML Schema will be sufficient for the schema catalog to cope with many description schemes such as the example `Melody` media description scheme, an understanding of the MPEG-7-DDL-specific extensions is nevertheless indispensable to deal with every media description scheme that might occur in practice.

There exist data models for the detailed representation of schema definitions written in XML Schema. These might serve as a basis for the representation of media description schemes in a schema catalog. The XML Schema standard itself provides the XML Schema Component Data Model [TBM+01]; Abstract Schemas [CLKR02] have been proposed in the context of the current DOM Level 3 standardization [LLW+03] as a data model for representing schema definitions for XML documents in a schema dialect-neutral way.

Validation of media descriptions It is widely accepted that databases should ensure data consistency. Similarly, an XML database solution should ensure the consistency of MPEG-7 media descriptions stored with it. For that purpose, a database solution should utilize the media description schemes kept in the schema catalog to validate the correctness of media descriptions.

A database solution that does not support the validation of MPEG-7 media descriptions defers the task of ensuring data consistency to the applications working with that solution. Apart from that it can be cumbersome to applications to constantly perform even most basic consistency checks, leaving the important responsibility of ensuring the consistency of media descriptions in the hands of applications can actually be dangerous and error-prone.

Validation of media description should be performed during the import of a media description into a database to prohibit the insertion of inconsistent media descriptions. Validation of media descriptions is also necessary during updates: if an update of a media description violates its description scheme, it should be rejected.

Inference of typed representations It has already been motivated that the provision of typed representations of the basic contents of a media description is an important requirement for an XML database solution suitable for MPEG-7. However, MPEG-7 media descriptions themselves do not contain any type information for their basic contents that could be used to create appropriate typed representations; this information is contained in the media description scheme associated with the description. Without the `Melody` media description scheme of Figure 2.4, for instance, there is no indication that the content of the `Contour` element in the media description shown in Figure 2.5 constitutes a list of integer values and not just arbitrary text.

To be able to automatically provide typed representations of the basic contents of an MPEG-7 media description, an XML database solution should exploit the type information available with the media description schemes of the schema catalog. Employing this information, the solution can infer the types and, by these types, create appropriate typed representations of the basic contents of the description.

Access optimization If an application accesses MPEG-7 media descriptions stored with an XML database solution via a high-level declarative query language such as XQuery and not via a low-level API for programmed navigational access like DOM, the access is open to automatic optimization by the database solution's query processor.

A prerequisite to sophisticated optimization of access to a database's contents is the availability of schema information. The information carried by the media description schemes contained in a schema catalog can be utilized for the optimization of declarative queries in a number of ways that may significantly reduce query evaluation times [BAOG98, FS98, Woo99]. For instance, expressions contained in a query that, according to a media description scheme, can never yield a result or that are redundant to other expressions can be cut off thereby reducing the complexity of queries. Furthermore, path traversals specified inside a query can be simplified according to those access paths that are actually permitted by the media description scheme. Using the media description scheme, queries against media descriptions can also be equivalently rewritten such that existing indexes can be exploited for query evaluation.

Without schema information carried by media description schemes providing an accurrate summary of the structure of the media descriptions contained in a database as a basis for decisions, however, sophisticated optimizations are difficult to perform. At best, a query processor can apply transformation rules that are generally valid for every potential media description according to some heuristics and statistics to rewrite queries into an equivalent form which is hoped to be evaluatable more efficiently.

An XML database solution whose query processor does not perform any automatic access optimization at all relies on application programmers to have detailed knowledge of the structure of the media descriptions that will be stored in a database and the indexes that will be available for accelerating access to these descriptions. Using this knowledge, programmers have to formulate queries inside their applications in such a way that they can be evaluated efficiently without any rewriting by the query processor. When the structure and available indexes change, however, an application's queries might need to be changed as well to still be performant.

3.4 Extensibility

Extensibility is a desirable property of an XML database solution managing MPEG-7 media descriptions. As motivated in the following, an ideal database solution should offer *extensibility with functionality* and *extensibility with index structures.*

Extensibility with functionality MPEG-7 basically constitutes a means for the definition of schemes for the description of multimedia content that is supplemented by a profound library of standardized, ready-to-use description schemes. However, the standard does not prescribe how MPEG-7-compliant media descriptions are to be processed by applications. For example, MPEG-7 defines a way how the melody of a song can be described with the `Melody` media description scheme; but it is left to an application how descriptions following the `Melody` media description scheme are employed, e.g., to compare two songs for similarity.

Given this situation, it is difficult to provide meaningful operations for the treatment of MPEG-7 media descriptions that are of use for all applications. Coming back to the example, different applications might even need different measures of

similarity for the comparison of songs on the basis of the `Melody` media description scheme.

Therefore, it is desirable that an XML database solution is extensible with functionality. Applications should be able to plug in procedures and functions for the processing of media descriptions and parts of media descriptions that suit their particular purposes. This corresponds to the concept of stored procedures and functions well-known from relational DBMSs. The functional extensions should be callable with the particular means for accessing media descriptions offered by a database solution, such as an API or a declarative query language. An example of a query language for XML documents that explicitly considers functional extensions is XQuery: the XQuery language allows the definition of custom functions which can then be used in queries just like any other function built-in with the language.

With a database solution that cannot be extended with functionality, in contrast, missing functionality usually has to be integrated within the application programs (if one neglects the possibility of modifying and extending the kernel of the database solution: this typically is a challenging and complex endeavor demanding that the solution's source code is available). In a client-server scenario, this conceptual unpleasantness may even constitute a performance problem. As an example, consider a song retrieval application acting as the client of an MPEG-7 database on a server. If the implementation of a reasonable similarity measure for melody contours has to be integrated into the application program, all media descriptions have to be transferred from the server to the client in order to select all those descriptions with contours similar to a given one according to the defined measure. The resulting network traffic is likely to be prohibitive for large databases.

Extensibility with index structures Since the MPEG-7 standard does not prescribe how media descriptions are to be processed, it is difficult to assess what index structures should be available with an XML database solution to best fit the needs of different MPEG-7 applications that are going to work with the solution. For instance, the availability of an R-Tree index structure for the indexing of media descriptions could perfectly suit the needs of a particular application while an extended k-d-Tree index structure could prove more effective for other applications.

Moreover, specialized index structures might exist focused on the needs of a specific application that are not useful for other applications. For example, one could imagine a highly specialized value index structure specifically designed for a classic music archive application enabling very efficient retrieval of classic music based on the `Contour` element of media descriptions complying to the `Melody` media description scheme. Such an index structure does not need to be effective for the retrieval of pop songs as well.

To flexibly accomodate the different indexation needs of MPEG-7 applications, we believe that an XML database solution should be extensible with new index structures. It should provide an open interface facilitating the integration of new value, text, or path index structures without the need to modify the system's kernel. Interfaces for the integration of new index structures exist in the context of object-relational DBMSs. For example, Oracle features the Extensible Indexing API allowing the provision of new index structures [GD02] and the microkernel of the Monet research prototype has been specifically designed to be flexibly extensible by kernel modules which may introduce, among others, new index structures into the system [BK95].

If there is no interface supporting the flexible integration of new index structures, a new index structure can be brought into a database solution (again neglecting the

possibility of modifying the solution's kernel) by extending the solution with functions (provided that this is possible) for constructing, maintaining, and querying the new structure which then have to be explicitly invoked by applications. The index structure could be stored just like any other data in the database. This approach, however, hides the existence of the index structure from the database solution so that, for example, it cannot be considered for automatic access optimization; also, *indexing is not transparent to applications.*

3.5 Classic DBMS Functionality

There is some classic but nevertheless important DBMS functionality which most database applications nowadays take for granted. It is state of the art that DBMSs provide support for *transactions* to guarantee atomic, consistent, isolated, and durable access to database content, exert *fine-grained concurrency control* to achieve a high concurrency between transactions, exercise *fine-grained access control* to grant and enforce access rights of users down to single pieces of data contained in a database, and come with reliable means for *backup and recovery* to shield applications from aborted transactions, system crashes, and media failures.

Since such features are also desirable for applications working with a database of MPEG-7 media descriptions, a suitable XML database solution should provide classic DBMS functionality as well. Given that the desirability of this functionality is neither specific to MPEG-7 database applications nor even specific to XML database applications, we abstain from giving a detailed justification of this demand from an MPEG-7 point of view. Nevertheless, XML database solutions will be analyzed with regard to classic DBMS functionality later in Chapter 5. As we will see, *support of classic DBMS functionality is no matter of course.*

Chapter 4

XML Database Solutions

The area of XML database solutions is still very active. There exists a confusing variety of systems at different development stages and with different degrees of maturity that are concerned with the management of XML documents in a database [Bou02]. To permit an analysis of XML database solutions with regard to the management of MPEG-7 media descriptions within the scope of this thesis, the investigations have been restricted on a set of prominent and representative XML database solutions sufficiently mature for reasonable examination.

	Native database solutions	Database extensions		
		Unstructured storage	Structured storage	Mapping
Commercial	eXcelon XIS GoXML DB Infonyte-DB Tamino TEXTML X-Hive/DB	IBM DB2 XML Extender Microsoft SQLXML Oracle XML DB		Oracle XML DB/ Structured Mapping
Open source	dbXML eXist Xindice		ozone/XML	
Research	Lore Natix PDOM TIMBER		Monet XML Shimura et al. XML Cartridge	

Figure 4.1: XML database solutions

As shown in Figure 4.1, the selected set of XML database solutions consists of commercial products, open source projects, and research prototypes roughly categorized according to common terminology into *native* XML database solutions and XML database *extensions* enabling the storage of XML documents within conventional, usually relational or object-oriented DBMSs. Even though it is frequently encountered in conjunction with XML databases and appears to be intuitively clear at first glance, the notion of a native XML database solution is cloudy and there exists no widely accepted definition. In practice, one has to realize that the term "native" is mainly employed by vendors of specialized XML database solutions as a marketing brand to distinguish their products from the XML database extensions offered by the major conventional DBMS vendors.

To give an impression of the variety of interpretations of the term "native" around, one class of definitions considers the way in which XML documents are stored as the central aspect that distinguishes native XML database solutions from others [Bou02]. Some of these definitions regard as the essential characteristic of a native XML database solution that it stores XML documents in their original textual file format at the most applying compression techniques to reduce storage space and maintaining additional indexes to speed up retrieval operations. Other storage-oriented definitions soften up this relatively rigid point of view and allow a native database solution to define a logical data model such as DOM for the hierarchical structure of XML documents and to store documents according to this model employing any physical storage model desired [XML03]. Yet another class of definitions, e.g., [Thu02], takes a very abstract viewpoint and defines a native XML database solution to be a solution that has been specifically developed for the management of XML documents.

Facing this diversity of definitions, we would like to come up with a manageable criterion to distinguish between the categories of native XML database solutions and XML database extensions for the purpose of this analysis, which has proven to be compliant to the self-conception of most vendors of XML database products. We regard as the distinctive feature between both categories that a native XML database solution allows the modeling of data only by means of XML documents. An XML database extension, in contrast, still offers applications the modeling primitives of the data model of the extended DBMS. Hence, the predicate "native" does not imply that a native database solution has been developed specifically for the management of XML documents from the ground up; a native solution might very well base on conventional database technology as long as the data model of the underlying system is entirely hidden.

The rest of this chapter briefly introduces the native XML database solutions (4.1) and XML database extensions (4.2) that are given in Figure 4.1.

4.1 Native Database Solutions

A variety of commercial native XML database solutions have appeared on the market to serve the increasing need for the efficient management of large amounts of XML documents. Arguing that XML documents cannot be efficiently stored in conventional DBMSs because of their hierarchical and semistructured nature, several vendors have developed entire DBMSs specialized on the management of XML documents. Perhaps the most prominent representative of this group of native XML database solutions is Tamino [Sof01b]; commercial solutions falling in the same category are X-Hive/DB [X-H02] and GoXML DB [XML01]. Infonyte-DB [Inf02] which evolved from the research prototype PDOM [HMF99] has also been specifically developed for the management of XML documents. However, Infonyte-DB does not provide a full-fledged database server with all the standard DBMS functionality such as transaction management and concurrency control but rather constitutes a lightweight in-process storage solution for XML documents.

Furthermore, vendors of object-oriented DBMSs have seized the opportunity to get their slice of cake from the market of XML document management and reshaped their existing DBMSs to native XML database solutions. A prominent representative of this approach is eXcelon XIS [eXc01a] which internally makes use of the object-oriented DBMS ObjectStore. In a similar manner, vendors of document management systems have reshaped their existing technology as native

XML database solutions, e.g., TEXTML [IXI01].

Apart from these commercial solutions, several open source projects aiming at the development of native XML database solutions have emerged recently. The Apache XML Project has started to implement a specialized DBMS for XML documents called Xindice [Sta02] from the gound up. Xindice is the successor of dbXML [Sta01] previously driven by the dbXML Project. eXist [Mey02] is another example of an open source project implementing a native database solution. In contrast to Xindice, however, eXist is built on top of a relational database system, MySQL or PostgreSQL, which internally serves as the persistent storage backend.

Finally, there has also been considerable research concerning native XML database solutions. A prominent example is the semistructured DBMS research prototype Lore that has been developed towards a native XML database solution [GMW99]. The idea is to exploit Lore's ability to represent irregular graph structures (including hierarchies) and the system's powerful query language for the management of XML documents. Natix [FHK+02, KM99] and TIMBER [JAC+02] are further prominent research prototypes of native database solutions. Whereas the heart of of Natix is a dedicated storage manager supporting a flexible and efficient clustering of XML document trees and subtrees into physical disc records of limited sizes, TIMBER has been built on top of the multiple purpose storage manager Shore [CDF+94].

4.2 Database Extensions

Basically, three approaches for representing XML documents in conventional DBMSs can be distinguished. In the first approach, which is called unstructured storage in the following, an XML document is stored directly in its textual format in a character large object (CLOB). This is the approach supported by most major relational DBMSs today: these systems have been extended with CLOB-based data types for XML documents along with more or less sophisticated functions for accessing a document's content from SQL. Prominent representatives of relational database extensions supporting the unstructured storage of XML documents are Oracle XML DB [HAA+02], IBM DB2 XML Extender [IBM00], and Microsoft SQLXML [Mic00b].

In the second approach, which is called structured storage, a fine-grained metamodel of XML documents capable of representing the node trees of arbitrary XML documents is built by employing the modeling primitives of the underlying conventional DBMS. This opens up the structure and the contents of XML documents to the querying facilities provided with the DBMS. There has been considerable research in this area (see [FK99] for an overview) and a lot of research prototypes of database extensions for the structured storage of XML documents have been developed, mostly for relational DBMSs. This analysis focuses on the representative prototypes XML Cartridge [GSN99], Shimura et al. [SYU99], and Monet XML [SKW+00, BK95] extending the relational DBMSs Oracle, PostgreSQL, and Monet respectively with support for the management of XML documents.

There are also open source projects that provide structured storage extensions for conventional DBMSs. The analysis includes the open source project ozone/XML [DN01]. ozone/XML is a library of persistent-capable classes for the object-oriented DBMS ozone that implements the DOM standard for the representation of XML documents in a database.

Finally, in the third approach for representing XML documents in conventional

DBMSs, which is called mapping, the content of XML documents is mapped to database schemas specifically designed for that content. This, in principle, places all the modeling capabilities available with a conventional DBMS at the disposal to efficiently and adequately represent document content. There exists a large number of tools and formalisms for the specification of the mapping between an XML format and a database schema. For example, IBM DB2 XML Extender and Microsoft SQLXML provide the concepts of document access definitions (DAD) and annotated schemas respectively for this purpose. Nevertheless, such tools and formalisms are neglected for this analysis: with these, the design of a database schema appropriate for the content of an XML format and the specification of the mapping between the XML format and the database schema are elaborate manual tasks. Bearing in mind that MPEG-7 allows the definition of arbitrary media description schemes in excess to those predefined with the standard, the effort necessary to cope with a media description following a previously unknown description scheme would be prohibitive.

Recently, there has been considerable research concerning the automatic derivation of relational database schemas from schema definitions for XML documents and the automatic mapping between them [STH+99, TDCZ02, DFS99]. As these approaches are based on Document Type Definitions (DTDs) and not on schema definitions written in the far more complex MPEG-7 DDL, they do not qualify for the management of MPEG-7 media descriptions. However, in its latest release of the Oracle XML DB database extension, Oracle has picked up the basic approach and extended it for XML Schema to provide an additional storage option for XML documents in excess to unstructured storage. Oracle calls this storage option Structured Mapping [HAA+02]. Since XML Schema constitutes a large subset of MPEG-7 DDL, Oracle XML DB/Structured Mapping is taken into account for the analysis.

Chapter 5

Analysis

This chapter analyzes in detail to what extent the XML database solutions introduced in Chapter 4 are suitable for the management of MPEG-7 media descriptions. For that purpose, it evaluates these database solutions along the requirements for the management of MPEG-7 media descriptions outlined earlier in Chapter 3.

Accordingly, the remainder of the chapter examines the fulfillment of the requirements concerning the representation of media descriptions (5.1), the access to media descriptions (5.2), media description schemes (5.3), extensibility (5.4), and classic DBMS functionality (5.5). The chapter concludes with a summary of the findings of the analysis (5.6). For a better orientation in the ensuing discussions, the results of the analysis are visualized in Figure 5.1.

5.1 Representation of Media Descriptions

Fine-grained representation Native XML database solutions typically represent XML documents, and hence also MPEG-7 media descriptions, in a fine-grained manner. For storage, they internally decompose an XML document into the individual nodes of which it consists. eXcelon XIS, Infonyte-DB, PDOM, X-Hive/DB, Xindice, dbXML, and eXist represent these nodes according to the DOM Structure Model. GoXML DB employs the XQuery 1.0 and XPath 2.0 Data Model. The research prototypes Lore, Natix, and TIMBER define their own, non-standard models based on edge-labeled trees for the internal representation of XML documents.

In contrast, the native solutions Tamino and TEXTML basically store XML documents in their entirety in their original textual format, only applying compression techniques to reduce the consumption of storage space. This is comparable to the XML database extensions IBM DB2 XML Extender, Microsoft SQLXML, and Oracle XML DB which store XML documents in CLOBs in an unstructured manner as well. Tamino somewhat alleviates the disadvantages of such a coarse-grained storage representation by managing additional fine-grained structural information: a structure index maintains path information for the individual elements and attribute values occurring in a document.

Since XML database extensions that follow the structured storage approach apply the modeling primitives of the underlying DBMS for the definition of detailed metamodels for XML documents, they offer a fine-grained storage representation of MPEG-7 media descriptions by construction: the object-oriented database extension ozone/XML is a library of persistent classes that constitutes a 1:1 implementation of the DOM Structure Model and API; the relational extensions Shimura et al.

		Native database solutions											Database extensions							
		eXcelon XIS	GoXML DB	Infonyte-DB/ PDOM	Tamino	TEXTML	X-Hive/DB	Xindice/ dbXML	eXist	Lore	Natix	TIMBER	IBM DB2 XML Extender	Microsoft SQLXML	Oracle XML DB	ozone/XML	Monet XML	Shimura et al.	XML Cartridge	Oracle XML DB/ Structured Mapping
Representation of media descriptions	Fine-grained representation	■	■	■	□	–	■	■	■	■	■	■	–	–	–	■	■	■	■	□
	Typed representation	–	–	–	–	–	–	–	–	□	–	–	–	–	–	–	–	–	□	□
Access to media descriptions	Fine-grained access	■	■	■	■	–	■	■	■	■	■	■	■	■	■	■	■	■	■	■
	Typed access	□	–	–	□	□	–	–	–	□	–	–	–	–	–	–	–	–	□	□
	Fine-grained updates	■	■	■	–	–	■	■	–	■	■	■	–	–	–	■	■	■	■	■
	Value index structures	□	□	□/–	□	□	□	□	–	□	–	□	□	–	□	–	□	□	□	□
	Text index structures	■	■	–	■	■	■	–	■	■	■	■	■	■	■	–	–	–	■	■
	Path index structures	■	–	■	■	–	■	–	–	■	■	■	–	–	■	–	■	■	■	–
Media description schemes	MPEG-7-DDL-compliant schema catalog	□	□	–	□	–	–	–	–	–	–	–	–	–	□	–	–	–	–	□
	Validation of media descriptions	□	□	–	□	–	–	–	–	–	–	–	–	–	□	–	–	–	–	□
	Inference of typed representations	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	□
	Access optimization	–	–	–	■	–	–	–	–	–	–	–	–	–	–	–	–	–	–	■
Extensibility	Extensibility with functionality	■	–	–	■	–	–	■	–	–	–	–	■	■	■	■	■	■	■	■
	Extensibility with index structures	–	–	–	–	–	□	–	–	–	–	–	–	–	■	–	■	–	■	■
Classic DBMS functionality	Transactions	■	■	–	■	–	■	–	–	■	■	■	■	■	■	■	■	■	■	■
	Fine-grained concurrency control	■	■	–	–	–	–	–	–	■	■	■	–	–	–	■	□	■	■	□
	Fine-grained access control	–	–	–	■	–	–	–	–	–	–	–	–	–	–	■	–	–	–	□
	Backup and recovery	■	□	□	■	□	■	–	■	□	■	■	■	■	■	■	■	■	■	■

Figure 5.1: Analysis results (■ support, □ partial support, – no support)

and XML Cartridge represent XML documents by the means of edge-labeled trees which they store in a central edge table; Monet XML dynamically creates a table for every distinct path by which an element or attribute value occurring in an XML document contained in the database can be reached from the root of the respective document. Every element and attribute value of a document is then stored in the table created for the corresponding path.

The database extension Oracle XML DB/Structured Mapping in principle facilitates fine-grained storage of XML documents as well. The system's mapping scheme basically creates an SQL object type for every complex type declared in a schema definition written in XML Schema. Such an SQL object type has one field each for every element type and attribute of which the corresponding complex

type consists. In case that a field corresponds to an element type with simple content or an attribute, it is assigned the atomic SQL data type coming closest to the simple type defining the respective content. In case that a field corresponds to an element type with complex content, it is assigned the SQL object type created for the complex type defining the content of that element type. Multiple occurrences of element types are handled by collection-typed fields. As an entry point to this structure, a table is created for each root element type with a single column which is assigned the SQL data or object type suitable for the root element type's content. To store an XML document that complies to the given schema definition using this storage representation, Oracle XML DB/Structured Mapping maps the contents of that document in a fine-grained manner to the corresponding columns and fields of the tables and SQL object types created.

When it comes to the storage of MPEG-7 media descriptions, however, this mapping scheme unveils considerable limitations: elements with mixed content and elements with arbitrary content cannot be represented, as well as elements which make use of the `xsi:type` attribute to announce complex type polymorphism. Oracle XML DB/Structured Mapping circumvents these restrictions by augmenting the SQL object types with additional CLOB fields serving as overflow stores that keep those fractions of XML documents that cannot be represented otherwise textually in an unstructured manner. But as MPEG-7 makes considerable use of the problematic constructs mentioned above (especially of complex type polymorphism, see Chapter 2), significant portions of typical MPEG-7 media descriptions can be expected to be kept in such overflow stores. This notably reduces the granularity of the storage representation for MPEG-7 media descriptions. The example media description of Figure 2.5, for instance, would be stored completely in an overflow store, because the root `AudioDescriptionScheme` element applies complex type polymorphism via `xsi:type` already.

Typed representation The investigated XML database solutions have strong deficiencies with regard to the representation of non-textual information significantly contributing to the content of MPEG-7 media descriptions. Most of them store the basic contents of an XML document, i.e., simple element content and the content of attribute values, as text regardless of the particular content type.

In fact, only a few solutions even address the issue of typed representation of basic document contents. The native database solution GoXML DB, as it employs the XQuery 1.0 and XPath 2.0 Data Model for the storage of XML documents, could in principle exploit the model's capabilities regarding typed representation of simple element content and the content of attribute values. Experiments with Version 2.0.2 revealed, however, that this model feature has apparently not been implemented: such content is always stored and treated as text. The native database solution Lore as well as the database extension XML Cartridge to some extent implement typed representations: for the representation of the content of attribute values, Lore provides a set of atomic data types including integer, real, and string. Simple element content, however, is always represented as text. XML Cartridge keeps the basic contents of a document in specific leaf tables which use the atomic SQL data types provided by the underlying Oracle DBMS to store values, with one leaf table for every supported data type.

Whether this support for typed representations leads to an adequate storage of the basic contents of MPEG-7 media descriptions, however, is questionable at least. Apart from the fact that the data types supported by Lore and XML Cartridge only constitute a limited subset of the simple types that are predefined with MPEG-7

DDL, all the systems do not support the simple type derivation methods coming with MPEG-7 DDL. Among others, they therefore cannot adequately represent lists and matrices commonly occurring in MPEG-7 media descriptions. Moreover, it remains unclear how Lore and XML Cartridge are actually supposed to create typed representations, since they do not take schema definitions into account for the storage of XML documents that contain type information needed for that purpose.

Out of the investigated database solutions, Oracle XML DB/Structured Mapping is the one providing yet the most support for the typed representation of the basic contents of an MPEG-7 media description. According to the mapping scheme explained above, simple element content and the content of attribute values of an XML document are kept in fields of SQL object types which are assigned the atomic SQL data type best suiting the particular content type as given by the schema definition associated with the document.

Nevertheless, there are again considerable limitations. On the one hand, the mapping scheme does not support many of the simple type derivation methods provided with MPEG-7 DDL: among others, lists are simply mapped to text fields and the mapping of matrices is not supported at all since Oracle XML DB/Structured Mapping cannot cope with the extensions of MPEG-7 DDL to XML Schema. On the other hand, the typed representation of simple element content or the content of an attribute value depends on whether the respective element or attribute value is kept in an textual overflow store or not, because it occurs in an element with mixed or arbitrary content or in an element which employs complex type polymorphism via the `xsi:type` attribute. These factors are likely to result in the textual representation of large parts of non-textual data carried by MPEG-7 media descriptions.

5.2 Access to Media Descriptions

Fine-grained access With the exception of TEXTML which allows the retrieval of complete XML documents only, all of the XML database solutions covered by this analysis provide means that give applications fine-grained access to the constituents of MPEG-7 media descriptions. Many solutions offer navigational programming interfaces which can be used to program fine-grained traversals of the structure of media descriptions. The native solutions eXcelon XIS, Infonyte-DB, PDOM, and X-Hive/DB as well as the database extensions Oracle XML DB, ozone/XML, and Oracle XML DB/Structured Mapping all offer implementations of the DOM API. Natix offers a file system driver giving a file system view on the hierarchical structure of XML documents: document content can thus be traversed via file operations.

With XML database extensions that follow the structured storage and mapping approaches, applications can directly use the query mechanisms available with the DBMS underlying the extension to access the content of media descriptions in a fine-grained manner. These extensions explicitly represent the content of XML documents with the modeling primitives of the DBMS. For example, the tables finegrainedly representing XML document content with the relational database extensions Monet XML, Shimura et al., XML Cartridge, and Oracle XML DB/Structured Mapping can be directly queried with SQL.

Most of the examined database solutions also support some form of declarative XML query language or query algebra that facilitates fine-grained access to MPEG-7 media descriptions. The native solutions eXcelon XIS, Infonyte-DB, X-Hive/DB, Xindice, dbXML, and eXist are capable of evaluating XPath expressions. Tamino implements a proprietary, slightly extended variant of XPath expressions called

X-Query [Sof01a], which should not be confused with the XQuery standardization effort, whereas Infonyte-DB and PDOM support XQL. Lore supports the expressive Lorel query language [AQM+97] for semistructured data. GoXML DB implements a subset of the current working draft of the XQuery standard. Natix and TIMBER define the query algebras NPA [FM01] and TAX [JAC+02], respectively, powerful enough to represent XPath expressions as well as most queries expressible with XQuery and provide appropriate translators from XPath and XQuery.

The relational database extensions IBM DB2 XML Extender, Microsoft SQLXML, and Oracle XML DB following the unstructured storage approach are capable of evaluating variants of XPath expressions on XML documents contained in a CLOB. For this purpose, they offer dedicated functions for use within SQL queries that take a CLOB containing an XML document and an XPath expression as arguments. When invoked, these functions internally load the whole XML document from the CLOB into main memory, parse it, evaluate the expression, and return the result of the evaluation as a string.

As the querying of XML documents stored with relational database extensions that follow the structured storage and mapping approaches by the means of SQL can be complex and cumbersome, the database extensions XML Cartridge, Shimura et al., and Oracle XML DB/Structured Mapping offer additional functions that take path expressions, XQL queries, and XPath expressions respectively as arguments and rewrite them to equivalent SQL statements.

Typed access It has already been pointed out that the investigated XML database solutions are weak in representing the basic contents of MPEG-7 media descriptions in an appropriate typed manner. Thus, it is no surprise that applications have difficulties in accessing such contents in a way adequate to the particular content type with these systems. As a matter of fact, those XML database solutions that store the basic contents of an XML document textually per se give applications textual access to basic document contents only. To access and process non-textual content in a reasonable way, applications must manually transform the content to appropriate typed representations everytime the content is accessed, e.g., by explicitly performing type conversions or by exploiting implicit type coercion rules associated with the operators of a query language such as XPath or XQuery.

Some XML database solutions that store the basic contents of a document textually facilitate a form of typed access to all those contents for which value indexes have been defined. For the definition of a value index on the content of elements or attribute values, eXcelon XIS, Tamino, TEXTML, and Infonyte-DB allow to specify whether the content is to be interpreted as strings, numbers, or dates for the purpose of indexing. When an element or attribute value indexed in such a way is accessed in a query, it is treated according the data type specified.

This form of typed access, however, quickly meets its limitations with MPEG-7: the set of data types available for value indexing does not come even close to the broad variety of predefined simple types and simple type derivations methods available with MPEG-7 DDL. For instance, reasonable typed access to lists and matrices frequently occurring in media descriptions is not possible. Moreover, indexed content is still retrieved as text typically forcing applications to transform retrieved non-textual content to typed representations more appropriate for further internal processing. Finally, wide-spread typed access to the basic contents of a media description requires the definition of many value indexes which is likely to slow down update performance prohibitively.

Those XML database solutions that keep the basic contents of an XML docu-

ment in typed representation generally can also give applications typed access to these contents. The limitations of these solutions with regard to typed representation directly strike through to typed access though: just as it is questionable how Lore and XML Cartridge are supposed to obtain typed representations without employing type information contained in schema definitions, it remains unclear how both systems are supposed to realize typed access to the basic contents of XML documents in practice. Similarly, because of the already mentioned deficiencies of the applied mapping scheme, it will not always be possible with Oracle XML DB/Structured Mapping to access the basic contents of MPEG-7 media descriptions in a typed manner from within SQL queries (or XPath expressions that have been rewritten to SQL queries), depending on whether the particular content is of a type that is always represented textually such as a list or whether the content happens to be kept in a textual overflow store due to complex type polymorphism, mixed content, etc.

Fine-grained updates Many of the XML database solutions investigated in this thesis not only provide mechanisms that give applications fine-grained read access to MPEG-7 media descriptions stored with them; in several cases, they also permit fine-grained updates of media descriptions. A common means offered by the examined systems for this purpose are navigational APIs allowing to program fine-grained updates of XML documents. The native database solutions eXcelon XIS, Infonyte-DB, PDOM, and X-Hive/DB as well as the database extensions ozone/XML and Oracle XML DB/Structured Mapping all implement the update-related methods of the DOM API; the file system interface offered by Natix allows fine-grained updates as well by the means of file operations.

Since database extensions following the structured storage and mapping approaches in detail represent the content of MPEG-7 media descriptions with the data model of the extended DBMS, they can consequently employ update mechanisms available with the DBMS to perform fine-grained modifications of media descriptions as well: the relational database extensions Monet XML, Shimura et al., XML Cartridge, and Oracle XML DB/Structured Mapping allow fine-grained updates of XML documents using suitable SQL UPDATE statements.

Apart from these update mechanisms, many native database solutions enable fine-grained updates of media descriptions via dedicated XML update languages. eXcelon XIS supports proprietary Updategrams [eXc01a]; GoXML DB extends XQuery with update operations. The database solutions Xindice and dbXML offer implementations of the XUpdate language. Lore's query language Lorel and TIMBER's query algebra TAX both include operations for performing fine-grained updates. As fine-grained updates of XML documents may result in complex SQL UPDATE statements, the relational database extension Oracle XML DB/Structured Mapping offers a function that rewrites update requests specified in a simple XML update language based on XPath expressions to semantically equivalent UPDATE statements.

In contrast, the native database solutions Tamino, TEXTML, and eXist do not allow fine-grained updates of MPEG-7 media descriptions. With these systems, an update of even a small fraction of an XML document can only be performed by replacing the complete document with an updated version. This is also the case with the IBM DB2 XML Extender, Microsoft SQLXML, and Oracle XML DB database extensions that unstructuredly store XML documents in CLOBs.

Value index structures The support for value indexing offered by the covered XML database solutions is generally not sufficient for the reasonable indexing of MPEG-7 media descriptions. The examined native XML database solutions typically come with ordered, B-Tree-based one-dimensional value index structures that can be employed for the indexing of simple element content and the content of attribute values of XML documents. The exceptions to this are PDOM, eXist, and Natix which do not provide value index structures at all. For effective indexing of MPEG-7 media descriptions, however, the capabilities of the native database solutions are limited: none of the examined native solutions offers multidimensional index structures such as R-Trees that are certainly desirable for the indexing of complex multimedia data frequently contained in media descriptions.

XML database extensions can rely on value index structures already provided by the extended DBMS. Database extensions that implement the structured storage and mapping approaches for the representation of XML documents can directly use these index structures, because they explicitly represent XML documents and their basic contents with the modeling primitives of the underlying DBMS. Among the investigated database extensions, this solely does not apply to ozone/XML as the DBMS ozone does not offer any value index structures.

Concerning the database extensions that follow the unstructured storage approach for the representation of XML documents, the question comes up how the value index structures of the extended DBMS can be applied to MPEG-7 media descriptions buried inside CLOBs. In fact, this is not possible with Microsoft SQLXML at all. IBM DB2 XML Extender allows to map the content of an element or an attribute value of an XML document contained in a CLOB to columns of synchronized side table which can then be indexed with DB2's value index structures just as any other column. Oracle XML DB supports the indexing of basic document contents kept in a CLOB by means of functional indexes based on the functions for XPath evaluation offered by the system.

Albeit most XML database extension thus allow one-dimensional value indexing of the basic contents of MPEG-7 media descriptions, they – just like the examined native XML database solutions – fall short of reasonable indexing of complex multidimensional data commonly carried by such descriptions. Event though several of the underlying DBMSs offer multidimensional value index structures like grid index and R-Tree index structures [IBM01a, MAA+02b, The02] that could be exploited for that purpose in principle, it is questionable whether the multidimensional index structures and database extensions fit together. For instance, the grid index and R-Tree index structures shipping with IBM DB2 and Oracle can only index table columns which are assigned specific geometric data types for the representation of two-dimensional shapes. It is unclear, how instances of these geometric data types can be obtained from complex multimedia data such as melody contours contained in MPEG-7 media descriptions stored with one of the XML database extensions based on IBM DB2 and Oracle so that the data is indexed effectively.

Likewise, PostgreSQL offers an R-Tree index structure that can index columns of any data type as long as the data type implements a specific set of operations. But the XML database extension Shimura et al. that is based on PostgreSQL cannot benefit from this index structure: Shimura et al. stores all simple element content and the content of attribute values as text in one single string column. It is doubtable that the set of operations required by the R-Tree index structure can be implemented for strings in such a way that the content of all elements and attribute values is reasonably indexable.

Text index structures Quite a few XML database solutions offer dedicated text index structures that can be exploited by applications for the realization of efficient text retrieval on MPEG-7 media descriptions. Among the native database solutions, X-Hive/DB and Natix provide an index structure allowing the full-text indexing of entire XML documents while eXcelon XIS, GoXML DB, Tamino, TEXTML, eXist, Lore, and TIMBER possess index structures allowing to textually index the content of single elements and attribute values at any level in an XML document. The native solutions Infonyte-DB, PDOM, Xindice, and dbXML, in contrast, do not offer any special structures for text indexing.

As with value index structures, XML database extensions can benefit from existing text index structures of the underlying DBMS for the realization of efficient text retrieval on MPEG-7 media descriptions. As there are text index structures available for the relational DBMSs IBM DB2, Microsoft SQL Server, and Oracle that can be applied to CLOBs and character columns [IBM01b, Mic00a, MAA+02a], the XML database extensions IBM DB2 XML Extender, Microsoft SQLXML, and Oracle XML DB that keep XML documents in their original textual format in CLOBs can directly employ these structures for the full-text indexing of entire XML documents. The database extensions XML Cartridge and Oracle XML DB/Structured Mapping founding on the DBMS Oracle can exploit the available text index structures as well: the text index structure can be applied to the respective leaf table columns and SQL object type fields in which both extensions store textual simple element content and attribute values in a fine-grained fashion.

The database extensions ozone/XML, Monet XML, and Shimura et al., however, do not support text retrieval with dedicated index structures: they do not implement text index structures themselves and the underlying DBMSs ozone, Monet, and PostgreSQL do not provide any special text index structures either.

Path index structures Several of the investigated XML database solutions offer index structures that can accelerate the traversal of access paths within MPEG-7 media descriptions in miscellaneous ways. Some, such as the structure index offered by eXcelon XIS, precompute and maintain the results of evaluating a given path expression on all XML documents in a database. Whenever an application needs to evaluate the indexed path expression, it can directly rely on these precomputed results instead of having to evaluate the expression itself.

Another group of path index structures offered by several database solutions maintains the access paths by which the individual elements and attribute values contained in the XML documents of a database can be reached from the respective document roots. This kind of path information can be used to rule out documents or parts of documents which will certainly be missed by a given path traversal. This is the intention of the structure index offered by the native database solution Tamino that helps to filter out irrelevant XML documents during the evaluation of an X-Query expression. The signature index structure supported by Infonyte-DB and PDOM is even able to perform this kind of filtering for parts of XML documents.

Access path information for the individual elements and attribute values of XML documents can also be exploited to accelarate the calculation of traversal results. Proponents of this approach are the relational database extensions Shimura et al., XML Cartridge, and Monet XML. Shimura et al. and XML Cartridge. These systems maintain additional path tables which augment the elements and attribute values kept in the edge tables with a textual representation of the access paths by which they can be reached from their corresponding document root. Thereby, even complex path traversals can be broken down to relatively simple string matching

operations on the path table. Monet XML inherently supports path indexing by construction because it stores, as explained before, elements and attribute values in specific tables created for the access path by which they can be reached from the respective document roots. Traversal of an access path thus essentially consists of selecting the rows of the table representing that path.

Yet another approach to path indexing supported by some of the investigated XML database solutions is based on maintaing schematic summaries of the contents of the XML documents in a database. Such schematic summaries can be as simple as an element type name index which is available with the native database solutions X-Hive/DB and TIMBER. Given the name of an element type, an element type name index collects all those elements occurring the database whose types bear that name. An application interested in the elements of a certain type can exploit such an element type name index to directly obtain all elements of interest instead of having to access and traverse all documents to seek out these elements itself. DataGuides offered by the native database solution Lore constitute a more complex schematic summary of document contents. A DataGuide is a graph that summarizes all access paths that are possible in the XML documents contained in a database. A DataGuide not only helps to decide whether a path traversal can yield a result for any of the documents contained in a database. Also, as a DataGuide references all elements or attribute values which can be reached by traversing a given path, path traversals can be performed to a large extent on the DataGuide itself avoiding the need to access the documents in the database.

Finally, some of the examined database solutions feature full-text index structures that can also play the role of path index structures. This is the case with Natix, whose full-text index structure in combination with extended Access Support Relations can accelerate path traversals, as well as with Oracle XML DB, whose full-text index structure can help to filter out XML documents which are irrelevant for the evaluation of an XPath expression.

5.3 Media Description Schemes

MPEG-7-DDL-compliant schema catalog None of the XML database solutions examined provides a schema catalog that is fully compliant to MPEG-7 DDL. In fact, the majority of the investigated solutions – namely Infonyte-DB, PDOM, TEXTML, Xindice, dbXML, eXist, Lore, Natix, TIMBER, Microsoft SQLXML, ozone/XML, Monet XML, Shimura et al., and XML Cartridge – do not maintain schema catalogs at all. They do not make use of available schema information for the storage of XML documents.

Other database solutions, such as X-Hive/DB and IBM DB2 XML Extender, maintain DTD-based schema catalogs. However, as pointed out before in Chapter 3 DTD-based schema catalogs in general cannot be considered appropriate for managing MPEG-7 media description schemes.

As XML Schema constitutes a large subset of MPEG-7 DDL, database solutions that implement XML-Schema-compliant schema catalogs promise at least partial support for the management of MPEG-7 media description schemes. eXcelon XIS, Oracle XML DB, and Oracle XML DB/Structured Mapping provide schema catalogs that more or less support the XML Schema standard and therefore can be expected to cope with many MPEG-7 media description schemes. In contrast, the schema catalogs of Tamino and GoXML DB implement very restricted subsets of XML Schema not going far beyond DTDs, significantly limiting their ability to

manage typical MPEG-7 media description schemes.

Validation of media descriptions All examined database solutions that maintain a schema catalog for the XML documents contained in a database employ the schema information available with the catalog for document validation. Up to the degree of MPEG-7 DDL / XML Schema supported, the database solutions eXcelon XIS, GoXML DB, Tamino, Oracle XML DB, and Oracle XML DB/Structured Mapping are thus able to validate MPEG-7 media descriptions.

These solutions usually perform the validation of XML documents against their schema definition automatically during document import. Only MPEG-7 media descriptions that are consistent with their media description schemes can thus be inserted into a database. As Tamino and Oracle XML DB support coarse-grained updates of XML documents only and, therefore, the update of a document is equivalent to the import of a new document into the database with these systems, both solutions also validate documents automatically after an update. This ensures that no update of an MPEG-7 media description can violate its media description scheme.

eXcelon XIS, GoXML DB, and Oracle XML DB/Structured Mapping, in contrast, do not validate documents after updates automatically. Thus, update operations can result in MPEG-7 media descriptions inconsistent with their description scheme. To avoid this, applications are required explicitly initiate the validation of media descriptions after updates with these systems.

Inference of typed representations Out of all investigated database solutions, only Oracle XML DB/Structured Mapping to some extent utilizes type information available in the schema catalog to create typed representations of the basic contents of MPEG-7 media descriptions. As wxplained before, the mapping scheme of Oracle XML DB/Structured Mapping translates a schema definition written in XML Schema to a relational database schema consisting of a series of SQL object types which have appropriately typed fields to store simple element content and the content of attribute values. When mapping an XML document to a relational schema created in such a manner, the document's basic contents are automatically brought from textual to typed representation when assigned the typed field specifically prepared for the respective content. Nevertheless, the ability of Oracle XML DB/Structured Mapping to produce typed representations for MPEG-7 media descriptions is limited: recall, for instance, that lists are generally assigned to textual fields and that elements with mixed and arbitrary content as well as elements using complex type polymorphism are kept in textual overflow stores.

The other XML database solutions that to some extent support the typed representation of simple element content and the content of attribute values, namely, Lore and XML Cartridge, do not maintain schema catalogs. As they therefore do not have necessary type information at their disposal, it remains unclear how they are supposed to infer typed representations basic document contents.

Access optimization The analyzed XML database solutions, native database solutions as well as database extensions, are generally weak when it comes to exploit schema information available within media description schemes for the optimization of an application's accesses to MPEG-7 media descriptions contained in a database. Even though many database solutions come with index structures that can considerably accelerate access, applications are usually forced to explicitly use existing indexes. Just to name a representative example, the native XML database eXcelon

XIS relies on applications to formulate XPath expressions in such a way that it is more than obvious to apply an existing index for query evaluation. Rarely, query processors are able to equivalently rewrite queries based on an XML document's schema definition such that relevant indexes are found and applied automatically.

Moreover, most database solutions do not employ schema definitions to reduce the complexity of queries by identifying and cutting off subexpressions that are redundant or that can never yield a result. Some systems like Infonyte-DB, PDOM, and TIMBER rewrite queries applying universally valid transformation rules on the basis of statistical information and heuristics in the hope that the resulting queries are simpler to evaluate and the amount of intermediary results is reduced.

Out of the examined solutions, only Tamino and Oracle XML DB/Structured Mapping offer more sophisticated techniques for access optimization based on the schema definition of an XML document. Tamino employs schema information to bring an X-Query query into a canonical form. Among others, path expressions (which may include wildcards and traversals to indirect child nodes) occurring in a query are replaced in the canonical form by the disjunction of all those paths that qualify for these expressions according to the schema definition. Based on this canonical form, it is straightforward for the query processor to decide which value or text indexes are available for query evaluation and to apply the structure index to filter out documents which cannot contribute to the query result.

Oracle XML DB/Structured Mapping employs the schema definition to rewrite an XPath expression that is to be evaluated on an XML document to an equivalent SQL query that operates on the tables and SQL object types of the relational schema specifically created for the schema definition. The SQL query is then passed on to the query processor of the underlying Oracle DBMS which can then apply all the sophisticated relational query optimization techniques available with the system.

5.4 Extensibility

Extensibility with functionality The examined categories of XML database solutions, native solutions and database extensions, differ considerably with regard to their extensibility with application-specific functionality. Many native database solutions, i.e., GoXML DB, Infonyte-DB, PDOM, X-Hive/DB, eXist, Lore, Natix, and TIMBER do not provide means with which new, custom functions and procedures can be integrated into these systems.

Native solutions that can be extended with application-specific functions are eXcelon XIS, Xindice, dbXML, and Tamino. Such a functional extension is called DXE Servlet with eXcelon XIS, XMLObject with Xindice and dbXML, and query function with Tamino. DXE Servlets, XMLObjects, and query functions are realized as Java classes; optionally, a Tamino query function can also be implemented as a COM class. Though very similar, the concepts of DXE Servlets, XML Objects, and query functions differ in the way how seamless they can be applied for the processing of XML documents contained in a database: query functions can be used in Tamino's X-Query language just like any of the built-in functions whereas both DXE Servlets as well as XMLObjects are not integrated with the respective system's query language and have to be explicitly invoked via dedicated APIs.

The investigated XML database extensions, in contrast, can all be augmented with custom functions and procedures using the mechanisms for functional extension available with the underlying DBMS. The object-oriented DBMS ozone underlying the database extension ozone/XML can be enhanced with arbitrary classes real-

izing application-specific functionality; the relational DBMSs underlying the other examined XML database extensions all support the definition of custom stored procedures and functions. These custom procedures and functions perfectly fit together with the XML database extensions on the level of SQL where they can be used in queries for XML documents just like built-in functions. However, they typically cannot be used within XML query languages often additionally supported by relational database extensions. For example, application-specific stored functions cannot be called within XPaths expression that are to be executed with the XPath evaluation function on XML document stored in CLOBs using Oracle XML DB.

Extensibility with index structures Most of the examined XML database solutions do not offer dedicated interfaces with which new index structures can be integrated if desired by an application. Instead, database solutions usually present themselves to application developers as monolithic systems. Out of the investigated native XML database solutions, only X-Hive/DB offers an open interface for the integration of new index structures into the system. However, this interface is limited to the integration of full-text index structures only.

For XML database extensions, the question of integrating a new index structure boils down to the question whether the underlying DBMS is extensible with new index structures. Among the DBMSs IBM DB2, Microsoft SQL Server, Oracle, ozone, Monet, and PostgreSQL which form the basis of the examined XML database solutions, only Oracle and Monet can be extended with index structures without the need to modify the system's kernel. Oracle can integrate new index structures into the system via the Extensible Indexing API; Monet features a microkernel that has been specifically designed to be extensible with so-called extension modules that can, among others, contain application-specific index structures. As a result, the investigated XML database solutions that base on Oracle and Monet, i.e., Oracle XML DB, Monet XML, XML Cartridge, and Oracle XML DB/Structured Mapping, can be extended with new index structures as well.

5.5 Classic DBMS Functionality

Many of the examined native XML database solutions have remarkable deficiencies with regard to classic DBMS functionality. Though transactions are a traditional concept of DBMSs, a surprising number of native solutions does not offer transaction support: Infonyte-DB and PDOM avoid the need for transactions by allowing a database to be accessed by a single process only thereby realizing an isolated view on a database's contents in the truest sense of the word; TEXTML implements a simple check-in/check-out mechanism for XML documents instead of transactions; Xindice, dbXML, and eXist even do not provide any means to shield concurrent applications from each other.

Those native XML database solutions that implement transaction support, i.e., eXcelon XIS, GoXML DB, Tamino, X-Hive/DB, Lore, Natix, and TIMBER, do not necessarly exert fine-grained concurrency control between different transactions concurrently accessing stored MPEG-7 media descriptions. The concurrency control mechanism of Tamino is based on coarse-grained document locks whereas X-Hive/DB locks the entire database during updates.

With regard to fine-grained access control, the situation is even worse. Infonyte-DB, PDOM, Xindice, dbXML, eXist, Lore, Natix, and TIMBER do not implement any kind of access control. As a consequence, any user can access any MPEG-7

media description stored with one of these systems. eXcelon XIS, TEXTML, and X-Hive/DB at least perform user authentication when a database connection is being established. Once user authentication has succeeded, however, full access to the database is granted. GoXML DB goes a small step further and allows to define access rights for individual XML documents.

The only covered native XML database solution that offers a fine-grained access control mechanism permitting the regulation of access to the individual elements of MPEG-7 media descriptions is Tamino. For each element type maintained within Tamino's schema catalog, access rights can be defined controlling access to the elements of this type occurring within XML documents.

The analyzed native XML database solution also lack maturity with regard to backup and recovery. Only eXcelon XIS, Tamino, X-Hive/DB, Natix, TIMBER, and eXist support backups and maintain write-ahead logs to recover a crashed database to a recent consistent state. But out of these solutions, just X-Hive/DB supports incremental backups. GoXML DB, Infonyte-DB, PDOM, and Lore offer means to perform backups as well but do not maintain write-ahead logs for recovery to a recent consistent state. In an emergency, one has to fall back to the state of the latest backup. Xindice and dbXML do not provide any backup and recovery support at all but rely on a database administrator to perform manual offline backups of hopefully consistent database snapshots via file copy.

In contrast to the native XML database solutions, the investigated XML database extensions are built on top of traditional DBMSs which all offer mature implementations of classic DBMS functionality. As they extend these traditional DBMSs, the XML database extensions can directly benefit from this functionality, especially with regard to transaction support and backup and recovery. Whether fine-grained concurrency control and fine-grained access control are available for access to MPEG-7 media descriptions stored with a database extension, however, depends on the particular storage scheme for XML documents. As IBM DB2 XML Extender, Microsoft SQLXML, and Oracle XML DB store XML documents unstructuredly in CLOBs, they can neither exert fine-grained concurrency control nor fine-grained access control: the relational DBMSs underlying these extensions cannot lock or define access rights for parts of CLOBs.

In comparison, the ozone XML database extension that follows the structured storage approach permits both fine-grained concurrency control as well as fine-grained access control: it represents the elements and attribute values of an XML document as individual objects and the concurrency control and access control mechanisms of the underlying ozone DBMS operate on the object level.

The storage schemes for XML documents employed by the database extensions Shimura et al. and XML Cartridge that also follow the structured storage approach represent the elements and attribute values of an XML document as individual rows in central edge tables shared among all XML documents in a database. As the relational DBMSs underlying both extensions support row-level locking but allow the definition of access rights for entire tables only, fine-grained concurrency control is possible for access to MPEG-7 media descriptions stored with Shimura et al. and XML Cartridge but no fine-grained access control.

The remaining database extension Monet XML following the structured storage approach cannot control access to MPEG-7 media descriptions it stores. Also, concurrency control performed by Monet XML is rather coarse-grained. This is due to the fact that the underlying Monet DBMS does not provide any means for access control and is only capable of locking entire database tables. Given the storage scheme applied by Monet XML, a lock on a table corresponds to a lock

on all elements or attribute values contained in any XML document stored in the database which can be reached by the access path the table represents.

Finally, the storage scheme applied by Oracle XML DB/Structured Mapping following the mapping approach maps the contents of XML documents to instances of SQL object types which in detail remodel the schema definitions to which the documents comply. As the underlying Oracle DBMS is able to fine-grainedly lock and define access rights on SQL objects, fine-grained concurrency control and access control can be realized for access to XML documents stored with this extension. Regarding the storage of typical MPEG-7 media descriptions, however, the granularity of both concurrency control and access control will be considerably reduced in practice because significant portions of these descriptions can be expected to be kept unstructuredly in CLOBs acting as overflow stores.

5.6 Summary

Summarizing one can say that none of the investigated XML database solutions suffices all requirements for the management of MPEG-7 media descriptions. Even though, if taking a look back at Figure 5.1, there are database extensions like Oracle XML DB/Structured Mapping and native database solutions like eXcelon XIS, Tamino, and X-Hive/DB which cover quite a lot of these requirements, this should not belie the substantial limitations that are seriously compromising their eligibility for the management of MPEG-7 media descriptions.

The main weakness of the examined solutions is that they store and treat simple element content and the content of attribute values of MPEG-7 media descriptions largely as text, regardless of the particular content type. This is unsatisfactory because of the large fractions of media descriptions consisting of non-textual data like numbers, vectors, and matrices making up technical metadata of media such as melody contours; applications will definitely want to access and process these data according to their type and not as text.

The source of this weakness is the fact that the inspected solutions do not sufficiently make use of schema and type information available with media description schemes for the management of MPEG-7 media descriptions. As a result, the systems often lack valuable data crucial not only for ensuring the consistency of database content by validating media descriptions and for reasonable query optimization but also for inferring the types of the basic contents of a media description. The majority of inspected database solutions – like Infonyte-DB, TEXTML, Xindice, Microsoft SQLXML, Monet XML, and Shimura et al. to cite some representative examples – totally ignore schema definitions for the storage of XML documents. And even those database solutions that maintain schema catalogs – such as eXcelon XIS, Tamino, GoXML DB, X-Hive/DB, IBM DB2 XML Extender, and Oracle XML DB – primarily use the information contained therein for the validation of XML documents only. Moreover, the schema catalogs just support DTDs or more or less XML Schema; none of the solutions fully supports MPEG-7 DDL.

Among all XML database solutions covered here, solely the relational database extension Oracle XML DB/Structured Mapping in principle constitutes a step into the right direction and makes use of schema definitions written in XML Schema to manage simple element content and the content of attribute values in a typed fashion. However, as it has been set out in detail before, the system nevertheless represents a very high percentage of the constituents of typical MPEG-7 media descriptions as text due to limitations of its mapping scheme, e.g., lists, matrices,

and elements making use of complex type polymorphism.

In addition to the issue of typed representations, there are further aspects that constrain the applicability of the examined XML database solutions for the management of MPEG-7 media descriptions. The value indexing support offered by the database solutions is generally not sufficient. At best, the analyzed solutions offer one-dimensional, B-Tree-based index structures for the indexing of the basic contents of XML documents. However, no solution supports multidimensional index structures such as R-Trees for the indexing of document content. This notably obfuscates the prospects of successfully implementing efficient multimedia retrieval applications on large collections of MPEG-7 media descriptions with these systems.

When applying existing XML database solutions for the management of MPEG-7 media descriptions – native database solutions as well as database extensions – one should also be aware of the fact that rather basic DBMS functionality such as fine-grained concurrency control and fine-grained access control cannot be taken for granted. While several solutions, e.g., Tamino, X-Hive/DB, ozone/XML, and Oracle XML DB/Structured Mapping, address some of these aspects, none of the investigated solutions satisfyingly covers all.

Disregard of these fundamental deficiencies: what are strengths and weaknesses of native XML database solutions on the one side and XML database extensions on the other side concerning the management of MPEG-7 media descriptions?

Native XML database solutions are typically strong with regard to the representation of and access to MPEG-7 media descriptions. Taking a look at Figure 5.1, one can see that native database solutions normally store MPEG-7 media descriptions with a fine granularity. Usually backing these fine-grained storage representations with an array of path, value, and text index structures, they also come with navigational APIs and dedicated XML query languages, in many cases not only allowing fine-grained access to but also fine-grained updates of media descriptions. However, native database solutions are weak when it comes to extensibility with new functionality and index structures and classic DBMS functionality.

XML database extensions offer a more heterogeneous picture compared to native database solutions with regard to the representation of and access to MPEG-7 media descriptions. It highly depends on the particular storage approach followed by an extension with which granularity MPEG-7 media descriptions are represented, whether and by what means fine-grained access and updates of media descriptions are possible, and whether and which index structures of the underlying DBMS can be exploited for the indexing of media descriptions. In this respect, database extensions that follow the unstructured storage approach have clear deficiencies whereas several of the database extensions that follow the structured storage and mapping approaches are comparable to native database solutions.

The main strength of XML database extensions is that they found on long established traditional database technology. Consequently, they can benefit from the mature implementations of classic DBMS functionality of the underlying DBMSs. Moreover, it has been a trend in the recent years to make traditional DBMS more extensible with new functionality and index structures. As XML database extensions can directly exploit these extension mechanisms for their own purposes, they are also strong with respect to extensibility.

Chapter 6

The Typed Document Object Model

The previous chapter has shown that current XML database solutions – commercial systems, open-source projects, as well as research prototypes – do not suffice the requirements for the management of MPEG-7 media descriptions given by Chapter 3. This thesis therefore wants to develop an XML database solution that better takes account of these requirements.

The heart of any XML database solution is a data model for XML documents. Such a data model provides a representation of the structure and contents of XML documents on a logical level and provides the basis for the processing of XML documents stored within a database. For the realization of an XML database solution that is suitable for the management of MPEG-7 media descriptions, the data model at the solution's core must already address basic requirements for the representation of and access to such descriptions that have been outlined earlier in Sections 3.1 and 3.2.

This chapter proposes the Typed Document Object Model (TDOM), an object-oriented model for XML documents created with exactly these requirements in mind. The chapter first takes a brief look onto existing data models for XML documents unveiling their limitations concerning the representation of MPEG-7 media descriptions (6.1). Having thus fortified the need for a new model, the chapter then illustrates and gives a definition of TDOM (6.2).

6.1 Data Models for XML Documents

A variety of data models for XML documents have been proposed in the literature, e.g., [JLS99, GMW99, GSN99, SYU99, SKW+00, KM99]. Concerning their application for the representation of MPEG-7 media descriptions, however, these models suffer from mainly two weaknesses: firstly, they typically constitute variations of rather simple edge-labeled tree and graph data models. Though they provide fine-grained representations of MPEG-7 media descriptions in principle, these models often ignore more subtle aspects of the descriptions' structure like the ordering the child nodes of an element, markup different from elements and attribute values such as processing instructions and comments, or the distinction between attribute values and elements. Secondly, they usually do not support typed representations: simple element content and the content of attribute values is typically represented as text

hindering the reasonable processing of non-textual data on the basis of these models. Those few models that support typed representations (e.g., [KM99, GMW99]) only offer limited subsets of the elementary simple types predefined with MPEG-7 DDL; none of the models supports the simple type derivation methods of MPEG-7 DDL.

In addition to the models originating from research, several data models for XML documents have appeared in the context of standardization efforts. Prominent representatives are the XPath Data Model which constitutes the foundation for the XPath language [CD99], the DOM Structure Model which is specified along with the DOM API by the Document Object Model (DOM) standard [LLW+00], and the XML Information Set [CT01]. These models generally offer detailed and accurate representations of XML documents. But their applicability for the processing of MPEG-7 media descriptions is limited, since they neglect the types of the basic contents of a description representing them always as text.

The XQuery 1.0 and XPath 2.0 Data Model is currently being defined as the foundation of the XQuery standardization effort for a common XML query language [BCF+02]. What makes the model interesting with regard to the representation of MPEG-7 media descriptions is that it supports the elementary data types predefined by XML Schema for the typed representation of simple element content and the content of attribute values. Nevertheless, difficulties concerning the use of the XQuery 1.0 and XPath 2.0 Data Model for MPEG-7 still remain: with the exception of lists, the current working draft does not support the far majority of the simple type derivation methods offered by XML Schema and MPEG-7 DDL for typed representations. Furthermore, the model is still in an unstable state. For example, the paradigm followed for the specification of the data model has been changed fundamentally during the standardization process. Instead of an open structural definition, the model is now opaquely defined by means of abstract data types.

6.2 TDOM in Six Points

The deficiencies of existing data models for XML documents call for a new model which may serve as the foundation of an XML database solution that pays attention to the specific requirements for the management of MPEG-7 media descriptions. With the Typed Document Object Model (TDOM), such a model is now proposed.

TDOM is an object-oriented model for XML documents that picks up the traditional DOM [LLW+00] approach and develops it further to allow a more adequate representation of MPEG-7 media descriptions. In particular, the development of TDOM has been geared towards supporting five elementary requirements of Chapter 3, namely the fine-grained and typed representation of MPEG-7 media descriptions, allowing fine-grained and typed access to these descriptions, as well as permitting flexible and fine-grained updates.

The remainder of this chapter provides a detailed and illustrated definition of TDOM which is presented in six points closely oriented along the mentioned requirements. Being an object-oriented model, UML class diagrams [Ana01] are employed for the definition of the various classes of TDOM and their interrelationships.[1]

[1]The TDOM classes defined in the subsequent class diagrams do not show any getter and setter methods with which applications can access and manipulate the classes' attributes and associations. A concrete implementation of TDOM, of course, has to provide such methods. As this is straightforward, however, they are omitted from the definition of the model for the sake of clarity.

Whenever it is necessary to make formal statements about TDOM, the Object Constraint Language (OCL) defined as part of the UML standard is employed.

1. TDOM is fine-grained

Similar to traditional DOM, TDOM faithfully and fine-grainedly reproduces the structure of an XML document with an object-oriented model that permits to access and manipulate the document's constituents at any required granularity. Using an object-oriented model is advantageous because object-oriented concepts are widely supported by potential implementation platforms for TDOM today, such as most programming languages, object-oriented and object-relational DBMSs. This promises a small gap between the model and its implementations.

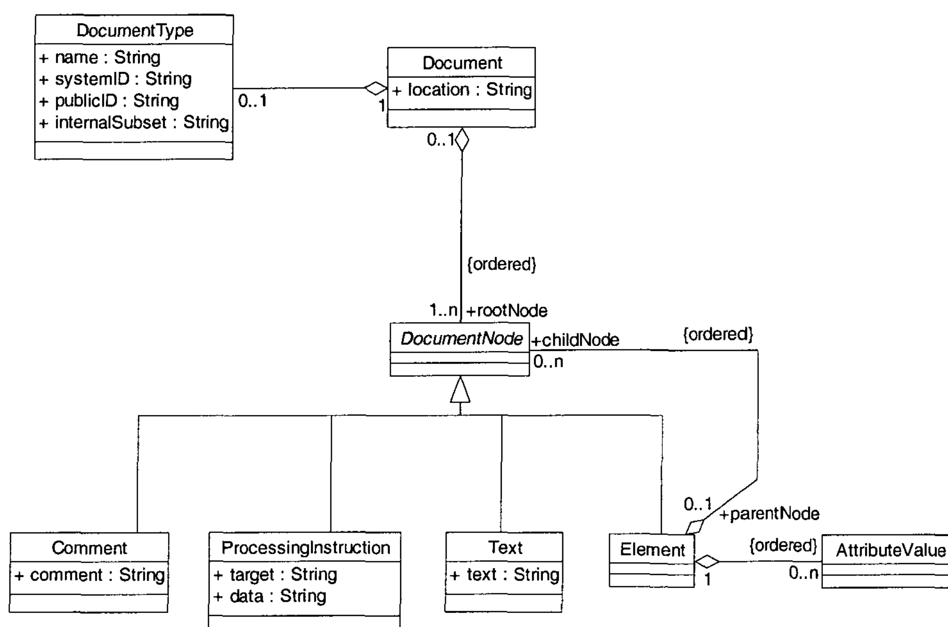


Figure 6.1: TDOM representation of XML document structure (UML class diagram)

The class diagram of Figure 6.1 introduces the classes of TDOM that are responsible for the representation of an XML document and the detailed reproduction of its structure. The class `Document` represents XML documents. In the model, a document is identified by its storage location addressed with an URL. A document can optionally be characterized by document type information (modeled by the class `DocumentType`) that might be conveyed in its `DOCTYPE` section. As an entry point to its contents, each document refers to the sequence of root document nodes constituting the top level of its hierarchical structure, which is expressed by the aggregation between the classes `Document` and `DocumentNode`.

Being an abstract base class, `DocumentNode` subsumes one class each for the representation of the primal kinds of nodes of which an XML document may consist: `Comment` represents comments, `ProcessingInstruction` represents processing instructions together with their associated source and target declarations, `Text` copes with text interspersed with other document nodes in mixed content, and `Element` represents elements. Through elements, the hierarchical structure of a document

is established – elements are the only kind of document nodes that may contain other nodes as their child nodes. This is expressed by the aggregation between `Element` and `DocumentNode`. Since elements can be further described by attribute values, TDOM introduces the class `AttributeValue` for their representation which is aggregated by `Element`.

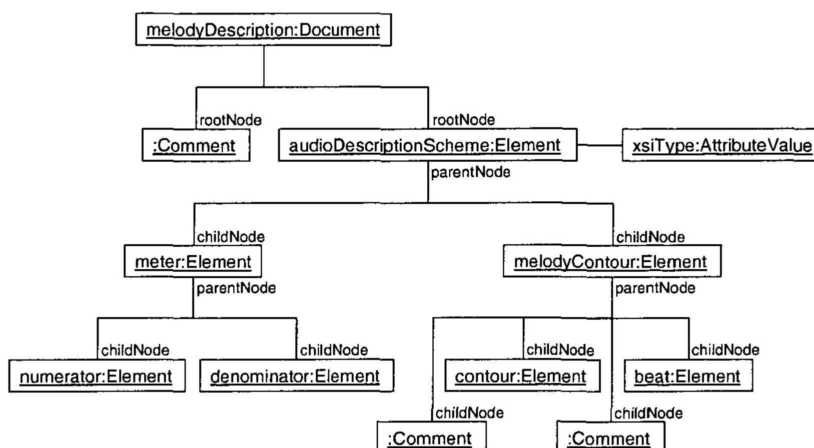


Figure 6.2: Structural representation of an MPEG-7 media description (UML object diagram)

The UML object diagram of Figure 6.2 exemplifies the structural representation of an MPEG-7 media description with TDOM using the example melody description of Figure 2.5. The description itself is represented by the `Document` object depicted at the top of the diagram; the document nodes contained in the description are represented by objects belonging to the TDOM-classes corresponding to the particular kind of node. Via the references of the `Document` object to its root nodes and the references of the `Element` objects to their child nodes and attribute values, TDOM reconstructs the hierarchical structure of the example description. Outgoing from the `Document` object, an application can thus traverse the structure of the example media description and access and manipulate any desired document node at any granularity.

There are some limitations on the allowable structure of XML documents. There is the restriction that the attribute names and attribute namespaces of the attribute values associated with an element must be unique. This is formally expressed in OCL by Contraint 1. The constraint employs the shorthands `attNamespace` and `attName` to refer to the attribute namespace and attribute name of an attribute value. These shorthands will be defined later under Point 3.

Constraint 1 (Unique attribute values)

```
context Element
inv: attributeValue -> forAll(av1, av2 |
      av1.attNamespace = av2.attNamespace and
      av1.attName = av2.attName implies
        av1 = av2)
```

There is the further limitation that there must be exactly one element among the root nodes of a document. Also, there must not be a text node among the root nodes. These restrictions are formally expressed by Constraint 2.

Constraint 2 (Root nodes)

```
context Document
inv: rootNode -> one(e | e.oclIsTypeOf(Element))
inv: not(rootNode -> exists(t | t.oclIsTypeOf(Text)))
```

The single element among the root nodes is called the root element of the document. The term root element is formalized by Definition 1.

Definition 1 (Root element)

```
context Document def:
let rootElement : Element =
    rootNode -> any(e | e.oclIsTypeOf(Element))
```

To ease navigation along the document hierarchy in future OCL expressions, Definition 2 finally introduces the formal shorthands `childElements` and `allChildElements` to refer to an element's sequence of direct child elements and to an element's set of direct and indirect child elements, respectively.

Definition 2 (Child elements)

```
context Element def:
let childElements : Sequence(Element) =
    childNode -> select(d | d.oclIsTypeOf(Element))
let allChildElements : Set(Element) =
    Element.allInstances -> select(e |
      childElements -> includes(e) or
      childElements -> exists(c | c.allChildElements -> includes(e)))
```

2. TDOM is typed

Traditional DOM represents the basic contents of an XML document as text prohibiting appropriate access to non-textual data. In contrast, it is TDOM's primary goal to exploit type information contained in media description schemes to which MPEG-7 media descriptions comply. The idea is to keep simple content of elements and the content of attribute values in a way that is appropriate for the particular content type. For this reason, typed representations have been made a central concept of TDOM.

In typed representation, elements and attribute values are tightly coupled to the element types and attributes declared in the schema definition accompanying an XML document. According to the class diagram of Figure 6.3 which unveils more details regarding the representation of elements and attribute values with TDOM, an element or attribute in typed representation (indicated by the boolean attribute `typed` of the classes `Element` and `AttributeValue`) is explicitly associated with the respective element type or attribute it instantiates, i.e., it is valid to. This is expressed by the associations between the classes `Element` and `ElementType` and `AttributeValue` and `Attribute` respectively. Element types and attributes are characterized by their names and namespaces and an optional scope.

Furnishing the classes `ElementType` and `AttributeValue` with the `scope` attribute is a tribute to the fact that MPEG-7 DDL, just like other schema definition languages for XML documents, not only allows to declare element types and attributes that are globally visible but also those that are only visible within a certain scope, e.g., a complex type. In order to distinguish different element types and

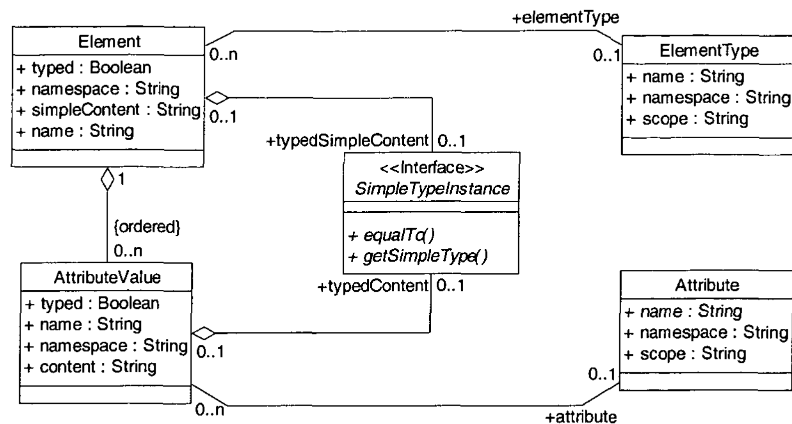


Figure 6.3: TDOM representation of elements and attribute values (UML class diagram)

attributes with identical names and namespaces that might exist within different scopes of one and the same schema definition, the attribute `scope` contains a string uniquely describing the scope in which the element type or attribute is visible.

The explicit association of elements and attribute values in typed representation with their element types and attributes declared in the schema definition not only provides an index allowing to efficiently look up all instances of a certain element type or attribute in a document. It also opens up type information that is used to acquire an adequate representation of the content of elements and attribute values: elements with simple content and attribute values in typed representation do not keep their content as text, but rather encapsulate their content within an object. This is captured in Figure 6.3 by the aggregations between the classes `Element` and `AttributeValue` and the interface `SimpleTypeInstance`, which the objects holding the content have to implement as a minimum (this interface will be described in more detail later under Point 5). Inside these objects, the content is kept in a way adequate to the content type declared for the element type or attribute in the schema definition. The objects offer methods specific to the content type that allow applications to appropriately operate on the content.

Figure 6.4 gives an example for a better understanding of typed representations. At the top of the Figure, the `Contour` element of the example MPEG-7 media description of Figure 2.5 is shown. Below the `Contour` element, the objects used for its representation are depicted in UML object diagram notation. A dashed arrow between an object and the `Contour` element indicates which part of the element is represented by the object. TDOM represents the whole element by an object of the class `Element`. In typed representation, an element is explicitly associated with the element type it instantiates. This is captured in the example by the reference from the `Element` object to the `ElementType` object representing the element type `Contour` that has been declared within the complex type `MelodyContourType` in the media description scheme of Figure 2.4. It is known from this element type declaration that the valid contents for elements of the type `Contour` are lists of integer values. As the example element is kept in typed representation, its content is thus encapsulated within an object of a class that offers an implementation for lists with reasonable methods to work with them – the class `List`. Since the elements

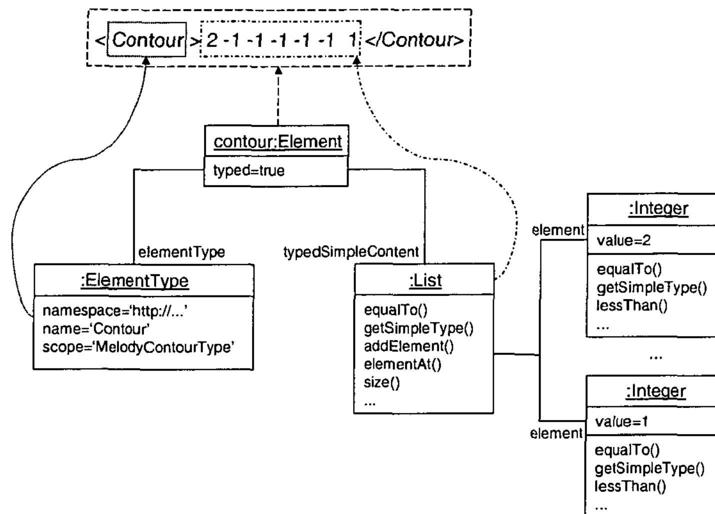


Figure 6.4: Typed representation example (UML object diagram)

of the list are known to be integer values, they are encapsulated in objects of the class `Integer` providing an implementation for integer values.

With this representation of the `Contour` element at hand, an application can now reasonably operate on the element. E.g., an application can query the size of the list making up the content of the element and access its single elements, all by invoking the appropriate methods `size()` and `elementAt()` offered by the class `List`.

There are some constraints that have to be obeyed with regard to typed representations though. It must be ensured that the content of an element in typed representation is either simple, i.e., it is represented by an object implementing the interface `SimpleTypeInstance`, or complex, i.e., its content consists of further child nodes via the aggregation between `Element` and `DocumentNode` given in the class diagram of Figure 6.1, both not both. This is expressed by Constraint 3.

Constraint 3 (Typed element content)

```
context Element
inv: typed implies
     (typedSimpleContent -> notEmpty() implies
      childNode -> isEmpty()) and
     (childNode -> notEmpty() implies
      typedSimpleContent -> isEmpty())
```

Moreover, it must be assured that the element type associated with a root element in typed representation is globally visible, i.e., it may not be scoped. This is expressed by the subsequent Constraint 4.

Constraint 4 (Typed root element)

```
context Document inv:
    rootElement.typed implies
     rootElement.elementType.scope = null
```

3. TDOM needs not to be typed

Even though it is the central goal of TDOM to exploit type information available in schema definitions to infer an adequate, typed representation of elements and attribute values for appropriate access, there are nevertheless situations in which type information is not available. This might be the case, for example, if a media description scheme makes use of constructs that prohibit type inference for parts of an MPEG-7 media description. As an example, the constructs `<any>` and `<anyAttribute>` of MPEG-7 DDL state that an arbitrary element or attribute value is valid as the content of a certain element type respectively. This includes elements and attribute values for which no further schema information is available. Obviously, it will prove difficult to create a typed representation of such elements and attribute values.

As a fallback for such situations, TDOM offers the notion of untyped representations. In untyped representation, elements or attribute values are decoupled from the schema definition, not being explicitly associated with the definition's element types or attributes. They maintain the name and namespace of their respective element type or attribute as well as their content in the corresponding textual attributes of the classes `Element` and `AttributeValue` that are depicted in the class diagram of Figure 6.3 – with all the problems involved related to the appropriate access to the content.

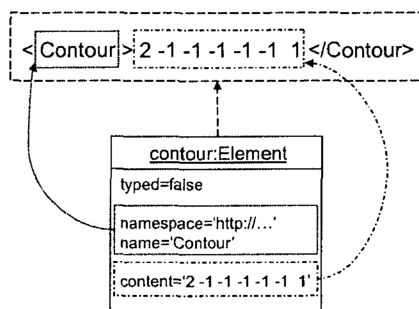


Figure 6.5: Unyped representation example (UML object diagram)

The UML object diagram of Figure 6.5 illustrates the concept of untyped representations. The diagram once more depicts the `Contour` element taken from the example MPEG-7 media description of Figure 2.5 – this time, however, in untyped representation. Again, dashed arrows indicate which parts of the object diagram correspond to which part of the element shown at the top of the figure. The encoding of the list of integer values constituting the content of the element in the textual attribute `content` is especially noteworthy. There is no indication for an application that this string represents a list. Without further knowledge, the content of the element can thus only be processed as a string with doubtable usefulness. Even if the application had that knowledge, it would always have to parse the string and cast it to an appropriate internal representation before adequate access to the list of integer values could take place.

There are some contraints with regard to untyped representations. Just as with elements in typed representation, it must be assured that the content of an element in untyped representation is either simple or complex. This is the purpose of Constraint 5.

Constraint 5 (Untyped element content)

```
context Element
inv: not(typed) implies
      (simpleContent <> null implies
       childNode -> isEmpty()) and
      (childNode -> notEmpty() implies
       simpleContent = null)
```

Moreover, elements and attribute values must be created in a consistent manner: an element or attribute value has to be either in typed or in untyped representation but not in an odd mixture of both. I.e., an element or attribute value in typed representation should not make use of the attributes of the classes `Element` and `AttributeValue` that are intended for untyped representations and vice versa. This is covered by Constraint 6.

Constraint 6 (Consistency of representations)

```
context AttributeValue
inv: typed implies
      attribute -> notEmpty() and
      typedContent -> notEmpty() and
      name = null and namespace = null and content = null
inv: not(typed) implies
      attribute -> isEmpty() and
      typedContent -> isEmpty() and
      name <> null and namespace <> null and content <> null

context Element
inv: typed implies
      elementType -> notEmpty() and
      name = null and namespace = null and
      simpleContent = null
inv: not(typed) implies
      elementType -> isEmpty() and
      typedSimpleContent -> isEmpty() and
      name <> null and namespace <> null
```

Finally, the definition of the formal shorthands `attName` and `attNamespace` that have been used in Constraint 1 to address the name and namespace of the attribute to which an attribute value belongs can now be provided:

Definition 3 (Attribute name and namespace)

```
context AttributeValue def:
let attName : String =
    if typed then
      attribute.name
    else
      name
    endif
let attNamespace : String =
    if typed then
      attribute.namespace
    else
      namespace
    endif
```

The definition of similar formal shorthands `etName` and `etNamespace` to address the name and namespace of the element type of an element will also prove useful later:

Definition 4 (Element type name and namespace)

```
context Element def:
let etName : String =
    if typed then
      elementType.name
    else
      name
    endif
let etNamespace : String =
    if typed then
      elementType.namespace
    else
      namespace
    endif
```

4. TDOM can be typed and untyped at the same time

It has already been mentioned before that MPEG-7 DDL offers constructs, e.g., `<any>` and `<anyAttribute>`, which permit the inclusion of elements and attribute values in an MPEG-7 media description for which no further schema information is available that could be used for the construction of typed representations. As a consequence, TDOM has to keep these elements and attribute values in untyped representation. Considering the advantages of typed representations, however, it is undoubtedly unattractive to keep all the description's other elements and attribute values for which schema information is available in untyped representation as well, just because of the existence of a few untypeable elements and attribute values.

For this reason, TDOM explicitly allows elements and attribute values in typed and untyped representation to coexist in a single document. It is very well possible that an element in typed representation has attribute values and child elements in untyped representation among its constituents: the declaration of the element type to which the element refers in typed representation might allow arbitrary child elements and attribute values including those for which typed representations cannot be inferred due to the lack of type information.

On the contrary, an element in untyped representation is not permitted to contain child elements and attribute values in typed representation. In untyped representation, the exact element type of an element is not known (only its name and namespace) and with it the type's declaration. Without the declaration, the exact element types and attributes of the child elements and attribute values of the element are not known as well and therefore the child elements and attribute values cannot be in typed representation. This restriction is captured by Constraint 7.

Constraint 7 (Untyped representation of elements)

```
context Element
inv: not(typed) implies
     not(childNode -> exists(e : Element | e.typed)) and
     not(attributeValue -> exists(av | av.typed))
```

5. TDOM supports arbitrary simple types

From the perspective of MPEG-7 DDL, an object representing the simple content of an element or the content of an attribute value in typed representation constitutes an instance of a simple type. MPEG-7 DDL predefines a comprehensive set of elementary simple types whose instances may occur as the content of elements and attribute values in MPEG-7 media descriptions, as well as a variety of derivation methods for the definition of new simple types.

For the handling of simple types and their instances, TDOM provides a generic simple type framework. Using that framework, support for arbitrary simple types and their instances can be smoothly integrated with TDOM which keeps the model simple and extensible and relieves us from the need to anticipate and to hardwire all supported simple types into the model.

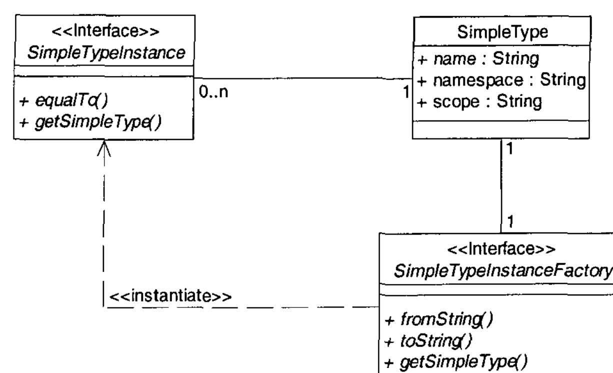


Figure 6.6: TDOM simple type framework (UML class diagram)

The simple type framework of TDOM is presented in the class diagram of Figure 6.6. As shown in the diagram, the framework represents simple types by the class `SimpleType`. A `SimpleType` object serves to represent either an elementary simple type predefined by MPEG-7 DDL or a simple type specific to a certain schema definition that has been derived from a predefined simple type using the constructs for type derivation available with MPEG-7 DDL. TDOM attributes a simple type with its name, namespace and an optional scope in which it is visible in a schema definition.

TDOM represents the instances of a simple type as objects of a class offering a meaningful implementation for the instances of that type. Each of these objects encapsulates a suitable representation of the simple type instance and offers type-specific functionality that can be used by applications to appropriately operate on the instance. The simple type framework, however, abstracts from the concrete classes implementing a certain simple type. Instead, it demands a minimal functionality that they have to provide which is specified by the interface `SimpleTypeInstance`. The interface `SimpleTypeInstance` consists of the methods `equalTo()`, which provides basic lookup functionality for simple type instances as it can be used to compare two simple type instances for equality, and `getSimpleType()`, which delivers simple type of the instance. Each simple type keeps track of its instances which is expressed by the association between `SimpleType` and `SimpleTypeInstance`.

Having provided a way to represent simple types and their instances, it must

be possible to construct simple type instances from the textual representation in which they are conveyed in XML documents as well as to reconstruct that textual representation from a given simple type instance. For that purpose, each simple type references a factory for the production of its instances. TDOM demands a minimum functionality for each of these factories which is collected by the interface `SimpleTypeInstanceFactory`. The interface provides the methods `fromString()`, which produces an instance of the simple type to which the factory is related from the textual representation in which the instance is conveyed in an XML document, `toString()`, which returns a textual representation of a simple type instance, and `getSimpleType()`, which delivers the simple type whose instances are produced by the factory.

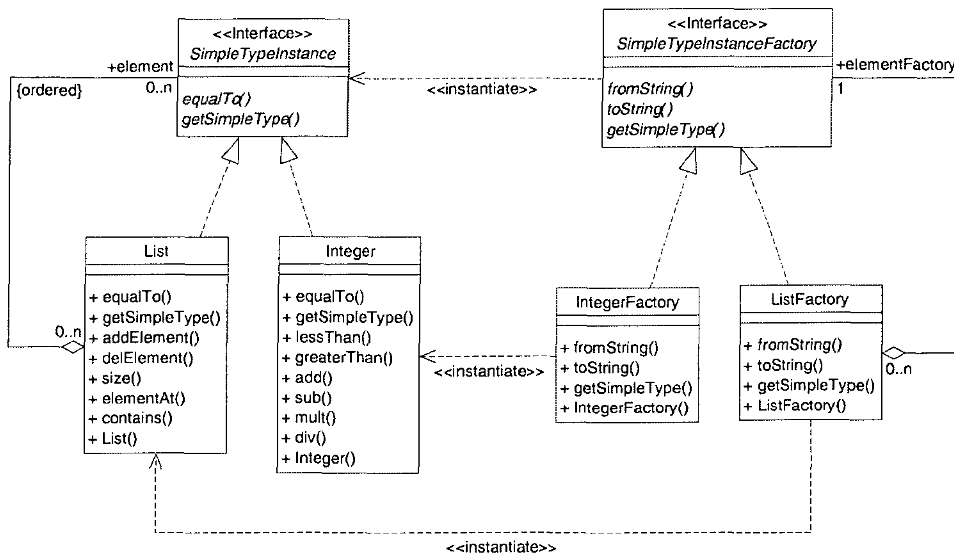


Figure 6.7: Example simple type support (UML class diagram)

Figure 6.7 gives an impression of how the simple type framework can be utilized to support a set of simple types. In the example, support for the simple type integer and its instances is provided. This is achieved by defining the class `Integer` which, beyond type-specific methods, e.g., for adding and substracting, implements the interface `SimpleTypeInstance` so that its objects are usable as the content of elements and attribute values in typed representation. For the construction of `Integer` objects from the textual representations in which integer values are encoded in XML documents, the class `IntegerFactory` is supplied implementing the Interface `SimpleTypeInstanceFactory`.

Likewise, TDOM can accommodate derivation methods for simple types. Figure 6.7 exemplifies the integration of a list type with TDOM. Similar to other simple types, the classes `List` and `ListFactory` provide support for the instances of the list type and for their construction by implementing the interfaces `SimpleTypeInstance` and `ListFactory`, respectively. In contrast to elementary simple types such as integer, however, the construction of instances of a derived simple type typically includes the construction of instances of the base type. In our example, the construction of a list includes the construction of instances of the simple type of its elements. Therefore, the factory for the list type must refer to the factory of its base type, modeled by the aggregation between `ListFactory` and `SimpleTypeInstanceFactory`.

With these classes, lists of integer values can be adequately represented in TDOM and constructed from the textual representation in which they are conveyed in XML documents; one can thus already build the typed representation of the example Contour element of Figure 6.4. The approach outlined for the implementation of simple types can be systematically followed to the extent where all the elementary simple types and simple type derivation methods coming with MPEG-7 DDL are supported.

In the following, the semantics of the methods of interfaces SimpleTypeInstance and SimpleTypeInstanceFactory introduced by the simple type framework are formally specified. Constraint 8 starts with the interface SimpleTypeInstance.

Constraint 8 (Simple type instance)

```
context SimpleTypeInstance::equalTo(SimpleTypeInstance sti) :
    Boolean
post: sti = self implies
      result = true
post: result = true implies
      self.getSimpleType() = sti.getSimpleType()

context SimpleType inv:
    simpleTypeInstance -> forAll(sti |
      sti.getSimpleType() = self)
```

The first postcondition of the method equalTo() ensures that a simple type instance is always equal to itself. The second postcondition states that, in order to be equal, two simple type instance must be of the same simple type. The invariant for the class SimpleType defines that the result of the method getSimpleType() on a simple type instance is the simple type associated with the simple type instance.

Constraint 9 describes the interface SimpleTypeInstanceFactory in more detail.

Constraint 9 (Simple type instance factory)

```
context SimpleTypeInstanceFactory::fromString(String s) :
      SimpleTypeInstance
post: result <> null implies
      result.getSimpleType() = self.getSimpleType()
post: result <> null implies
      self.simpleType.simpleTypeInstance -> forAll(sti |
        self.toString(sti) = s implies
          sti.equalTo(result))

context SimpleType
inv: simpleTypeInstance -> forAll(sti1, sti2 |
      simpleTypeInstanceFactory.toString(sti1) =
        simpleTypeInstanceFactory.toString(sti2) implies
          sti1.equalTo(sti2))
inv: simpleTypeInstanceFactory.getSimpleType() = self
```

The first postcondition of the method fromString() assures that an instance of a simple type successfully constructed from a textual representation refers to the simple type associated with the factory. The second postcondition states that if an instance of a simple type is successfully constructed from a textual representation that is the result of the call of the method toString() on another instance

of the same type, then both instances are equal. In other words, `fromString()` constitutes the inverse method to `toString()`.[2] In general, one can say that if calling `toString()` on two instances of the same simple type yields the same textual representation, then both instances are also equal to each other. This is formally described by the first invariant of the class `SimpleType` in the constraint above. Finally, the second invariant of `SimpleType` defines that the result of the call of the method `getSimpleType()` on a simple type instance factory is always the simple type to which the factory belongs.

6. TDOM facilitates flexible, fine-grained updates

The basic characteristics of TDOM pave the way to sophisticated updates on MPEG-7 media descriptions. The model's fine-grained representation of an XML document's structure allows applications to access any part of the document and to perform modifications at any granularity. Moreover, the combination of the concepts of typed and untyped representation of elements and attribute values offer great flexibility with respect to updates.

To illustrate the benefit of having both typed and untyped representation available, Figure 2.5 depicts an update on the example melody media description. An application might want to replace the `Beat` element by a new one. A natural way to perform this task would be the deletion the `Beat` element followed by the insertion of the new `Beat` element as a child of the element `MelodyContour`.

Did TDOM only support typed representations, it would have to be ensured after every single update operation that every element and attribute value affected by the update is valid with respect to the declaration of the particular element type or attribute it is associated with in typed representation. This is very rigid. In the example, the deletion the `Beat` element already violates the validity of the `MelodyContour` element, since, according to the schema definition of Figure 2.4, an element of type `MelodyContour` must contain exactly one element of type `Beat`. Thus, the deletion and thereby the whole sequence of update operations would have to be refused – even though the subsequent insertion of the new `Beat` element would restore schema consistency.

But having the additional means of untyped representations at hand (see Figure 6.8), applications can transform elements and attribute values in typed representation (1) that are affected by an update to a corresponding untyped representation (2). Thereby, they are decoupled from the element types and attributes of the schema definition. Any desired sequence of update operations can then be performed without being concerned with schema validity (3). After all update operations have been completed, the updated elements and attribute values can be brought back to corresponding typed representations (4) as long as the document is still valid with respect to the schema definition.

What is meant exactly by the terms corresponding untyped representation and corresponding typed representation? A corresponding untyped representation should reproduce an element or attribute value that is kept in typed representation as faithful as possible with the means of untyped representation. Likewise, a corresponding typed representation should faithfully reproduce an element or attribute value in untyped representation by the means of typed representation.

[2] The opposite need not to be true. For example, one and the same float value might be constructed from different textual representations (e.g., `123e-2` and `12.3e-1` represent the same float value `1.23`). However, calling `toString()` on the float value always yields just one of the possible textual representations which does not need to be the one from which the value has been constructed.

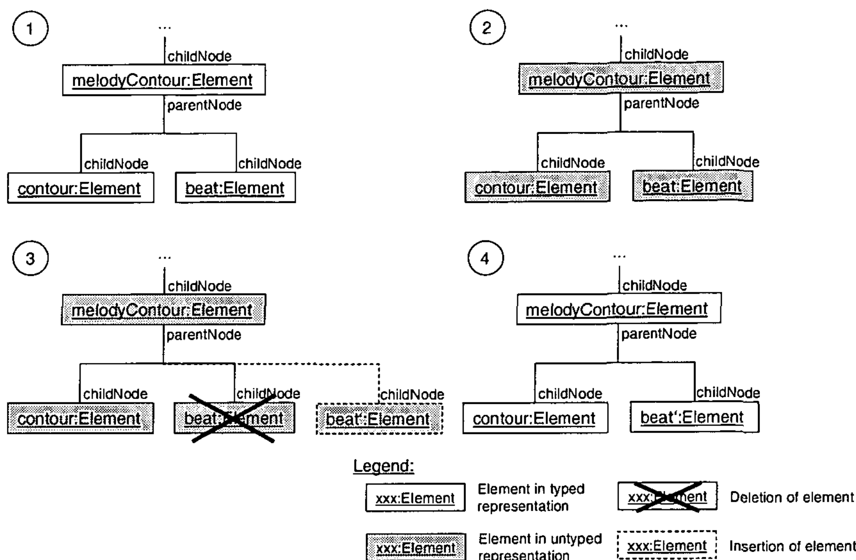


Figure 6.8: Switching between corresponding representations

Definition 5 formalizes a natural notion of correspondence for attribute values. An attribute value `av'` in untyped representation constitutes a corresponding untyped representation of an attribute value `av` in typed representation (formally: `av.CUR(av')`), if `av'` refers to the name and namespace of the attribute associated with `av` and if the textual content of `av'` is a textual representation of the simple type instance forming the content of `av`. Conversely, one can also say that `av` constitutes a corresponding typed representation of `av'` (formally: `av'.CTR(av)`).

Definition 5 (Corresponding representations of attribute values)

```
context AttributeValue def:
let CUR(AttributeValue av) : Boolean =
    typed and not(av.typed) and
    attribute.namespace = av.namespace and
    attribute.name = av.name and
    typedContent.simpleType.simpleTypeInstanceFactory.
      fromString(av.content) <> null and
    typedContent.simpleType.simpleTypeInstanceFactory.
      fromString(av.content).equalTo(typedContent)
let CTR(AttributeValue av) : Boolean =
    av.CUR(self)
```

Definition 6 formally introduces a notion of correspondence for elements.[3] Following that definition, an element `e'` in untyped representation constitutes a corresponding untyped representation of an element `e` in typed representation (formally: `e.CUR(e')`), if `e'` refers to the name and namespace of the element type associated with `e`. If `e` has simple content, it is furthermore demanded that `e'` has simple content as well and the simple content of `e'` is a textual representation of the simple type instance forming the simple content of `e`. If `e` has complex content, however, it

[3]In the definition, the existence of the method `deepEqualTo()` to compare two objects for deep equality is assumed.

is demanded that `e'` also has complex content and the child nodes of `e'` are equal to the child nodes of `e` – with the exception of elements: the child elements of `e'` are expected to be corresponding untyped representations of the respective child elements of `e`. Finally, every attribute value of `e'` must appear among the attribute values of `e` or be a corresponding untyped representation of an attribute value of `e`. With all these conditions fulfilled, one can conversely say that `e` constitutes a corresponding typed representation of `e'` (formally: `e'.CTR(e)`).

Definition 6 (Corresponding representations of elements)

```
context Element def:
let CUR(Element e) : Boolean =
    typed and not(e.typed) and
    elementType.namespace = e.namespace and
    elementType.name = e.name and
    (typedSimpleContent -> notEmpty() implies
      e.simpleContent <> null and
      typedSimpleContent.simpleType.simpleTypeInstanceFactory.
        fromString(e.simpleContent) <> null and
      typedSimpleContent.simpleType.simpleTypeInstanceFactory.
        fromString(e.simpleContent).equalTo(typedSimpleContent)
    ) and
    (childNode -> notEmpty() implies
      childNode -> size() = e.childNode -> size() and
      Sequence{1..childNode -> size()} -> forAll(i : Integer |
        childNode -> at(i).deepEqualTo(e.childNode -> at(i)) or
        (childNode -> at(i).oclIsTypeOf(Element) and
        e.childNode -> at(i).oclIsTypeOf(Element) and
        childNode -> at(i).CUR(e.childNode -> at(i)))
    ) and
    (attributeValue -> notEmpty() implies
      attributeValue -> size() = e.attributeValue -> size() and
      attributeValue -> forAll(av1 |
        e.attributeValue -> exists(av2 |
          av1.deepEqualTo(av2) or av1.CUR(av2))))
let CTR(Element e) : Boolean =
    e.CUR(self)
```

The construction of a corresponding untyped representation of an element or attribute value in typed representation is straightforward as the typed representation generally contains all the information that must be included with the corresponding untyped representation. An attribute value in typed representation keeps the name and namespace of the attribute with the attribute referred to by the attribute value in typed representation. Moreover, a textual representation of the content of the attribute value can be obtained from the simple type instance by employing the method `toString()` of the associated simple type instance factory. Definition 7 furnishes the class `AttributeValue` with the method `untype` which transforms an attribute value in typed representation to a corresponding untyped representation in this manner. The definition also outlines a straightforward implementation of this method as pseudocode.

Definition 7 (Untyping attribute values)

```
context AttributeValue::untype()
pre: self.typed
```

post: self@pre.CUR(self)
pseudocode:
```
-- change attribute value to untyped representation
self.typed := false
-- get attribute name and namespace from
-- attribute definition
Attribute att := self.attribute -> any(true)
self.name := att.name
self.namespace := att.namespace
-- remove reference to attribute definition
self.attribute := self.attribute -> excluding(att)
-- construct textual representation of content
-- from simple type instance
SimpleTypeInstance sti := self.typedContent -> any(true)
self.content := sti.simpleType.simpleTypeInstanceFactory.
  toString(sti)
-- remove reference to simple type instance
self.typedContent := self.typedContent -> excluding(sti)
```

Likewise, an element in typed representation keeps the name and namespace of the element type with the element type referenced. A textual representation of a potentially existing simple content can be derived from the simple type instance representing that simple content in typed representation via the associated simple type instance factory. Corresponding untyped representations of any child elements and attribute values of the element can be obtained recursively. Definition 8 augments the class `Element` with the method **untype** which implements this approach to bring an element in typed representation to a corresponding untyped representation.

Definition 8 (Untyping elements)

context Element::untype()
pre: self.typed
post: self@pre.CUR(self)
pseudocode:
```
-- change all child elements to untyped representation
foreach e1 in self.childNode ->
  select(e2 : Element | e2.typed) do
    e1.untype()
endforeach
-- change all attribute values to untyped representation
foreach av1 in self.attributeValue ->
  select (av2 | av2.typed) do
    av1.untype()
endforeach
-- change element to untyped representation
self.typed := false
-- get name and namespace of element type from
-- element type definition
ElementType et := self.elementType -> any(true)
self.name := et.name
self.namespace := et.namespace
-- remove reference to element type definition
self.elementType := self.elementType -> excluding(et)
-- construct textual representation from simple type
-- instance representing potentially existing simple
-- content
```

```
if self.typedSimpleContent -> notEmpty() then
  SimpleTypeInstance sti := self.typedSimpleContent ->
    any(true)
  self.simpleContent := sti.simpleType.
    simpleTypeInstanceFactory.toString(sti)
  -- remove reference to simple type instance
  self.typedSimpleContent := self.typedSimpleContent ->
    excluding(sti)
endif
```

In contrast to the construction of a corresponding untyped representation, the construction of a corresponding typed representation of an element or attribute value in untyped representation is more complicated. This is due to the fact that elements or attribute values in untyped representation do not, apart from the name and namespace of their respective element type or attribute, convey type information that would allow the construction of a valid corresponding typed representations solely on the basis of the untyped representation. Additional information in form of a schema definition is needed. With the element types and attributes and the associated type information contained in a schema definition, the respective element type or attribute can be inferred to which an element or attribute value in untyped representation is valid. Based on the inferred element type or attribute and the associated type information, a corresponding typed representation can then be constructed straightforwardly.

To this end, this thesis has developed typing automata. A typing automaton constitutes a well-defined, executable representation of the schema and type information carried in a schema definition that is capable of traversing an XML document, inferring the element types and attributes to which the elements and attribute values of the document comply, and obtaining corresponding typed representations of these elements accordingly. We will treat typing automata in detail in the next chapter.

Chapter 7

Typing

The previous chapter has introduced the TDOM data model for XML documents as a basis for the development of an XML database solution that is suitable for the management of MPEG-7 media descriptions. The design of TDOM already addresses several of the fundamental requirements regarding the management of MPEG-7 media descriptions. The model's main virtue is that it offers the concept of typed representation for elements and attribute values in XML documents. With typed representations, TDOM exploits available type information contained in schema definitions such as MPEG-7 media description schemes to represent simple element content and the content of attribute values appropriate to the particular content type thereby allowing applications to reasonably access and process such contents. For cases that type information is not available, TDOM still offers untyped representations where simple element content and the content of attribute values is kept as text.

A central characteristic of TDOM is that representations can be switched depending on the needs of a particular task. For instance, it may be useful to transform elements and attribute values that are affected by an update operation to untyped representation prior to the update. In that manner, they are decoupled from the schema definition permitting updates that temporarily violate the schema. Similarly, it is reasonable during the import of XML documents to TDOM, i.e., when bringing XML documents from their textual format into TDOM representation, to first produce a TDOM representation of the document that makes use of untyped representation only: untyped representations can be constructed without having to consider schema information. As a second step, the elements and attribute values can then be brought to corresponding typed representations by exploiting schema information for a more reasonable representation of document contents.

While the straightforward construction of corresponding untyped representations from elements and attribute values in typed representation has already been covered, this chapter discusses in detail how, given a media description scheme written in MPEG-7 DDL, corresponding typed representations from elements and attribute values in untyped representation can be obtained.

The discussion starts with some basic considerations on the problem (7.1). Then, the concept of typing automata as an executable and language-neutral formalism for representing MPEG-7 media description schemes is proposed (7.2). Typing automata are capable of inferring and creating typed representations of elements and attribute values. The computational complexity of the behavior of typing automata is examined (7.3) and, in order to reduce the effort necessary for creating typed rep-

resentations in many practical situations, optimizations are suggested (7.4). This chapter concludes showing how the basic typing automaton mechanism can be extended, so that even the more complex constructs of MPEG-7 DDL are supported and the expressiveness of that language is reached (7.5).

7.1 Basic Considerations

The construction of a corresponding typed representation of an element or attribute value in untyped representation can be regarded as a process consisting essentially of two steps: firstly, it has to be inferred to which element types or attributes declared in a schema definition the element or attribute value is valid (if any). Secondly, a typed representation of the element or attribute value has to be constructed based on the type information carried by one of the inferred declarations. While the second step is pretty straightforward, implementing the first step on the basis of schema definitions expressed in a schema definition language like MPEG-7 DDL quickly shows considerable complexity.

Figure 7.1 intends to get across a presentiment of this. It shows the sample MPEG-7 media description known from Figure 2.5 in a TDOM representation consisting solely of untyped representations along with the `Melody` media description scheme of Figure 2.4 to which the description complies. The element type declarations spread all over the description scheme are highlighted. For the first step in constructing typed representations, a TDOM implementation must find out which element types the elements of the media description validly instantiate – i.e., the implementation somehow has to infer exactly those relationships between elements and element types which have been marked by dashed arrows in the figure.

But this inference is difficult: MPEG-7 DDL is a declarative schema definition language. It defines no directly executable algorithm for inferring those declarations in a schema definition that are validly instantiated by a particular element or attribute value. As it can be seen at hand of the example `Melody` media description scheme, MPEG-7 DDL furthermore supports highly complex constructs for the structuring of schema definitions such as complex types and complex type derivation that further complicate validation directly on the basis of DDL syntax.

Faced with these difficulties, the adoption of an artifice common to the discipline of compiler construction lies close at hand. In compiler construction, declarative grammars are typically translated to various kinds of formal automata that serve as simpler, executable intermediary representations of grammars for the purpose of parsing. In a similar manner, MPEG-7 media description schemes could be translated into a simpler intermediary and executable representation for the purpose of inferring valid element types and attributes.

In the literature, several executable intermediary representations of schema definitions have been proposed for XML document validation. Proposals include rather exotic approaches that translate schema definitions to XSLT stylesheets [Cla99] which transform XML documents to HTML pages highlighting those places inside these documents that are not valid [Jel99]. Another approach is to transform schema definitions to LL(1) grammars [KV00] which are then fed into standard parser generators used for compiler construction to generate code for specialized parsers specifically tailored to these schema definitions. Further approaches use various kinds of formal automata for the intermediary representation of schema definitions, such as finite state automata [SV02] (which can cover a restricted subset of non-recursive schema definitions only due to their limited expressiveness), pushdown automata

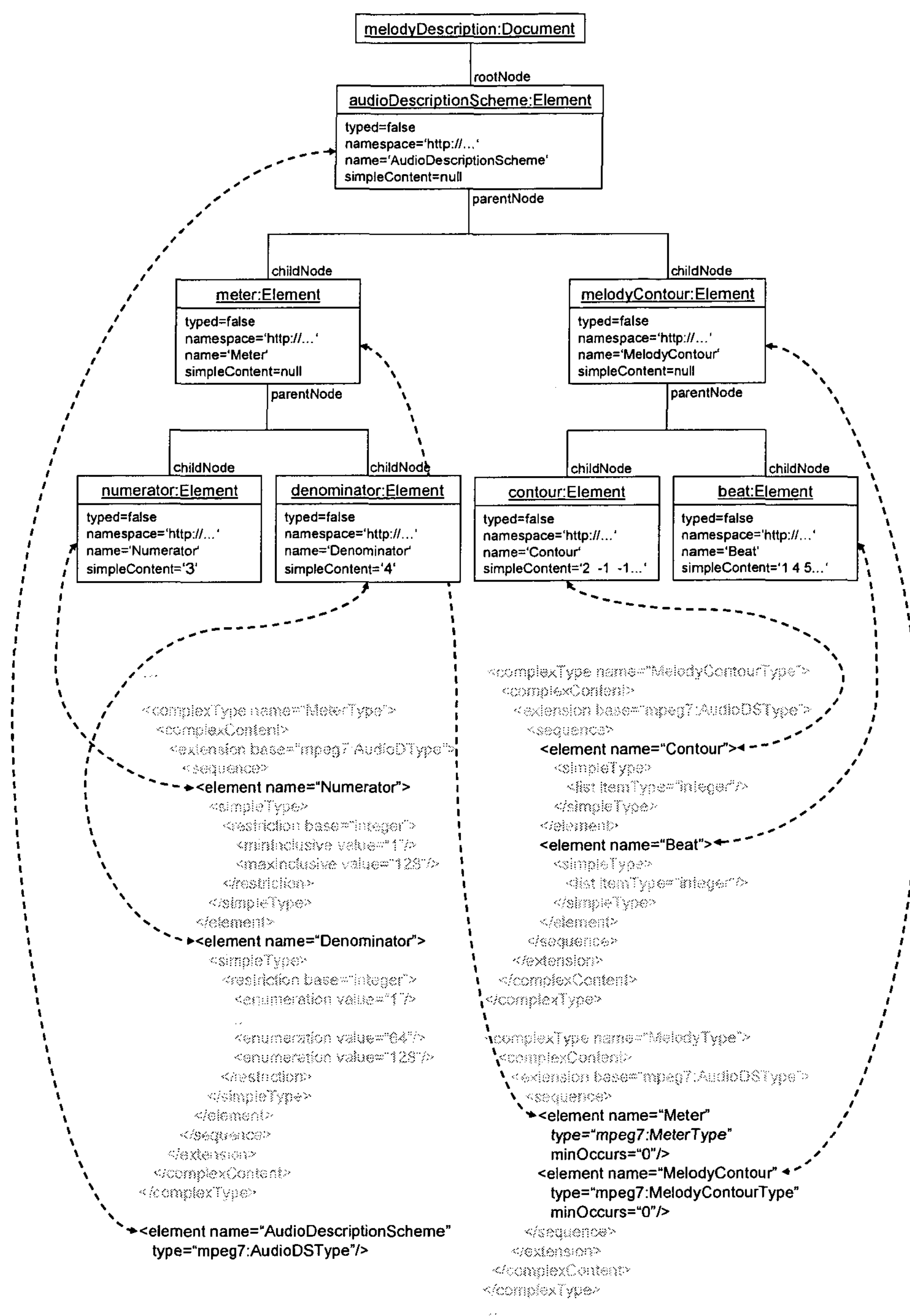


Figure 7.1: Typing problem

[SV02], and regular tree automata [Chi00, ML02, Mur99, Pre98, Hos00]. The latter have heavily inspired the design of several schema definition languages such as TREX [Cla01] and RELAX-NG [CM01].

With regard to the typing problem, the adoption of regular tree automata as

an intermediary representation of MPEG-7 media description schemes is especially attractive: regular tree automata essentially reduce the problem of validating an XML document to the problem of successively evaluating string regular expressions. The evaluation of string regular expressions is well-understood and there exists a broad variety of highly efficient software libraries for this purpose. Apart from the fact that these libraries not only simplify the implementation of regular tree automata in practice, most of these libraries – for instance, libraries that support Perl 5 regular expressions – additionally offer powerful extensions to traditional regular expressions that prove useful to cope with more complex constructs of MPEG-7 DDL. Regular tree automata also have manageable computational complexity: it is known that a deterministic regular tree automaton consumes a tree with a running time linear to the number of tree nodes [CDG+02]. Last but not least, regular tree automata permit a natural and intuitive representation of MPEG-7 media description schemes as we will see.

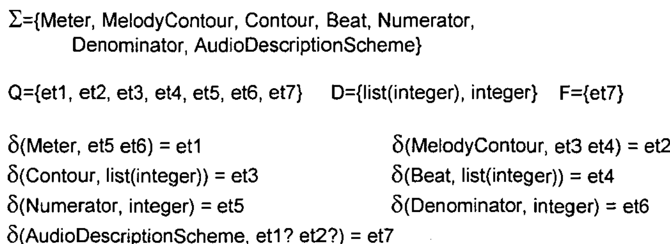


Figure 7.2: Example tree automaton

To give an impression of regular tree automata and how they can serve as a means for the intermediary representation of media description schemes, Figure 7.2 depicts the example `Melody` media description scheme (depicted at the top of the figure) represented as a bottom-up regular tree automaton[1] (depicted at the

[1] Note that literature also knows of top-down regular tree automata [CDG+02]. Since these

bottom). The figure employs the notation for regular tree automata introduced in [Chi00].

As it can be seen to the left of the figure, a bottom-up regular tree automaton is basically a 5-tuple consisting of the sets Σ defining the alphabet used for the naming of tree nodes, Q defining the set of states that may be applied to the individual tree nodes during the consumption of a tree by the automaton, D defining the set of datatypes to which leaf node contents have to comply, $F \subseteq Q$ defining the set of final states indicating a successful consumption of a tree, and the function δ defining the transition rules according to which states are applied to tree nodes.

A transition rule always takes a name $n \in \Sigma$ and yields a state $q \in Q$. Two different variants of transition rules are distinguished. The first variant is applicable to leaf nodes only and takes a datatype $d \in D$ as an argument in addition to n, e.g., $\delta(\mathit{Contour}, \mathit{list}(\mathit{integer})) = \mathit{et3}$. Whenever a leaf node l bears the name n and has a content that complies to d, the transition rule fires and q is applied to l. The second variant of transition rules is applicable to inner nodes only and takes a string regular expression over Q as an additional argument, e.g., $\delta(\mathit{Meter}, \mathit{et5}\ \mathit{et6}) = \mathit{et1}$. Whenever an inner node i bears the name n and there exists a concatenation of states applicable to the child nodes of i that complies to the regular expression, the transition rules fires and q is applied to i.

A bottom-up regular tree automaton starts consuming a tree at the leaf nodes making its way up to the root node constantly trying to apply the transition rules to the nodes traversed. If a state $f \in F$ can be applied to the root node, then the tree has been successfully consumed by the automaton.

As Figure 7.2 exemplifies, the formalism of regular tree automata can be utilized for the representation of the `Melody` media description scheme in a straightforward manner. Every element type declared in the media description scheme is given a textual label (*et1*, ..., *et7* in this case) which is indicated in the figure by grey circles next to the declarations. These labels make up the set of states Q. The idea is that, while consuming an XML document from the bottom up, the tree automaton applies to an element exactly the labels of those element types that are validly instantiated by the element. The labels of all unscoped element types in the description scheme make up the set of final states F. Whenever the tree automaton attaches one of these states to the root element, the document is considered valid because the root element correctly instantiates a globally visible element type declaration.

Furthermore, the names of the element types declared in the media description scheme (namespaces have been neglected for the sake of simplicity) make up the alphabet of allowed tree node names Σ. The simple types used in the description scheme for the declaration of element types with simple content constitute the set of datatypes D (again, some details of the simple type declarations, such as enumerations and the like, have been omitted in this example for simplicity reasons).

Finally, every element type declaration in the description scheme is translated to a corresponding transition rule of the tree automaton. Each of these transition rules takes the name of the declared element type as the first argument and yields the element type's label as its result. Depending on whether the content of the element type is declared as simple or complex, a transition rule of the first or second variant is created. For element types with simple content, the second argument of the transition rule is the simple type used for the content declaration. For element types with complex content, the content model is translated to an equivalent regular expression based on the labels of those element types that occur in the content

classes are generally equivalent to each other, the limitation to bottom-up regular tree automata implies no loss of generality.

model. For instance, the content model of the element type `MelodyContour` (which is labeled *et2*) consisting of a sequence of elements of types `Contour` and `Beat` (labeled *et3* and *et4*, respectively) is translated to the regular expression *et3 et4*. The regular expression created in this manner consitutes the second argument of the transition rule.

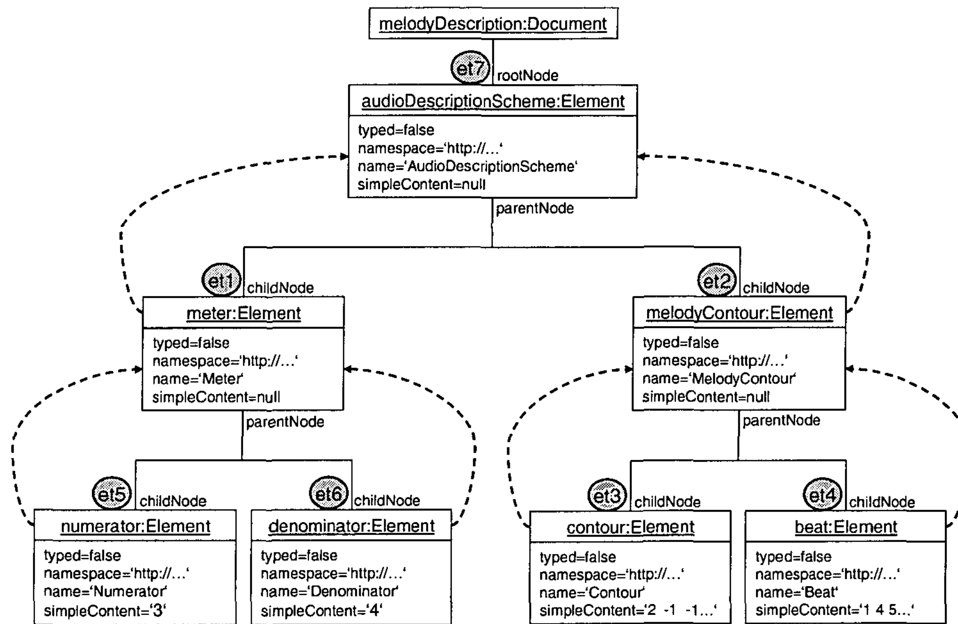


Figure 7.3: Tree automaton application (UML object diagram)

Figure 7.3 illustrates the consumption of the example MPEG-7 media description in TDOM representation by the constructed bottom-up regular tree automaton. Beginning at the leaf elements of the description, the automaton ascends through the tree structure as indicated by the dashed arrows. For every element, the automaton fires as much transition rules as possible. The figure shows the states yielded for the different elements of the media description as grey circles. As the applicable states are labels representing the different element type declared in the media description scheme, the automaton thus infers exactly those types that are instantiated by the respective elements. Since the label *et7* attached to the root element refers to the globally visible element type `AudioDescriptionScheme`, the automaton has also found that the media description as a whole is valid with regard to the `Melody` media description scheme.

7.2 Typing Automata

As seen in the previous section, bottom-up regular tree automata provide an intuitive formal foundation for the intermediary, language-neutral representation of MPEG-7 media description schemes. Their execution behaviour permits the inference of the element types and attributes that are instantiated by the elements and attribute values in an MPEG-7 media description. For the aim of constructing corresponding typed representations of elements and attribute values in untyped

representation within TDOM, that mechanism is therefore picked up and to developed further to what this thesis calls typing automata.

At their core, typing automata still constitute bottom-up regular tree automata. However, they have been remodeled in an object-oriented fashion in order to be compatible and seamlessly applicable to XML documents represented with TDOM. During remodeling, special care has been taken to keep typing automata extensible so that they can reach the expressiveness of MPEG-7 DDL and are thus suitable for the representation of arbitrary MPEG-7 media description schemes. Further exceeding the functionality of regular tree automata, typing automata are not only capable of validating XML documents and finding the element types and attribute values instantiated by the elements and attribute values of an XML document; they are additionally able to produce corresponding typed representations of the elements and attribute values on the basis of these inferred element types and attributes in a second processing phase.

In the following, typing automata are introduced and formally specified by means of UML and OCL. First, some basic definitions and the specification of the overall structure of typing automata is provided (7.2.1). For simplicity, the existence of attributes and attribute values is neglected in the ensuing definitions (we will come back to attributes and attribute values and how they can be incorporated into typing automata later in Section 7.5). Then, the behavior of typing automata when they are applied to TDOM-represented XML documents is specified. This behavioral specification is broken down into two phases: the validation phase (7.2.2), in which the element types instantiated by the elements of an XML document are inferred, and the typing phase (7.2.3), in which elements in untyped representation are brought to corresponding typed representations accordingly.

7.2.1 Structure

Before starting with the structural definition of typing automata, some preliminaries have to be addressed. In order to give a typing automaton the ability to address element types within string regular expressions just like a regular tree automaton, the class `ElementType` that represents element types within TDOM must be able to provide a textual label uniquely identifying a given element type.

Definition 9 serves exactly that purpose. It introduces the formal shorthand `etID` which delivers a textual identifier for an element type consisting of four parts separated by the delimiter `"::"`: the first part is always the string `"et"` indicating that the ID refers to an element type. The second part consists of the scope of the element type, followed by the namespace and the name of the element type as the third and fourth part. The inclusion of the scope, namespace, and name of an element type into its ID has the advantage that these data can be accessed within string regular expressions. As it will be seen later, this facilitates the implementation of more complex constructs supported by MPEG-7 DDL on the basis of regular expressions.

Figure 7.4 illustrates element type IDs by showing the element types occurring in the example `Melody` media description scheme represented as instances of `ElementType` together with their IDs.

Definition 9 (Element type ID)

```
context ElementType def:
let etID : String =
    "et::".concat(scope.concat("::".concat(
      namespace.concat("::".concat(name)))))
```

Figure 7.4: Example element type IDs (UML object diagram)

There may be situations in which no element type is declared in a schema definition that suits a particular element in an XML document. Nevertheless, the document does not necessarily have to be invalid: the element of unknown type might validly occur, for example, in an element whose content is defined via the <any> construct of MPEG-7 DDL. In order to be able to proceed with the consumption of the document, a typing automaton needs a textual label for the unknown type of the element. Definition 10 introduces the shorthand `uetID` for the class `Element` that provides such an identifyer for an unknown element type. The IDs delivered by `uetID` are very similar to those delivered by `etID`. The only differences are that they always start with the prefix `"uet"` to distinguish them from known element types declared in a schema definition and that the scope fraction of the ID is always set to `"null"`.

Definition 10 (Unknown Element type ID)

```
context Element def:
let uetID : String =
    "uet::null::".concat(etNamespace.concat("::".concat(etName)))
```

After these preliminaries, the specification of typing automata can now be provided. Figure 7.5 defines the structure of a typing automaton by means of an UML class diagram. As it can be seen from that diagram, a typing automaton, which is modeled by the class `TypingAutomaton`, consists of a set of states, which are element types represented by the TDOM class `ElementType`, and a set of transition

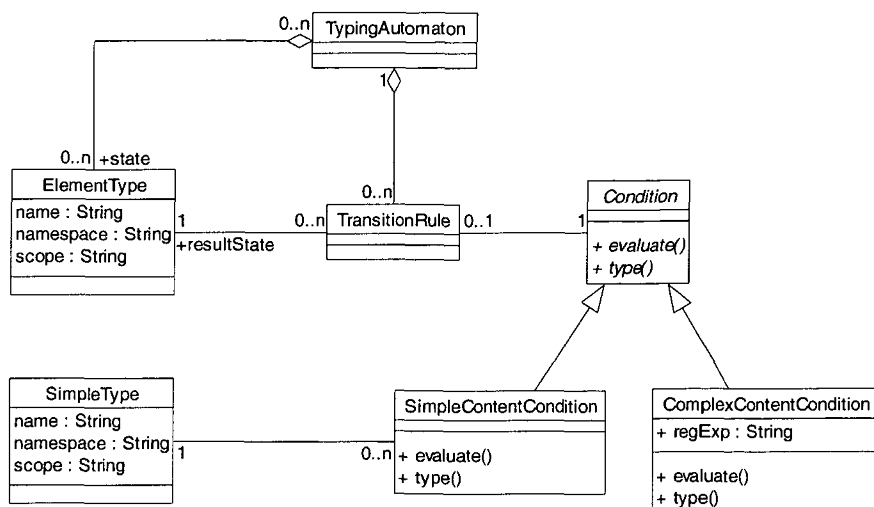


Figure 7.5: Typing automaton structure (UML class diagram)

rules modeled by the class `TransitionRule`. Transition rules define how the states, i.e., element types, are to be applied to the elements of a TDOM-represented XML document when the document is consumed by the automaton. A transition rule consists of two parts: the result state which is applied to an element when the transition rule is applicable and a condition that decides applicability.

Conditions are represented by the abstract base class `Condition` which offers two abstract methods `evaluate()` and `type()`. The method `evaluate()` takes an element and the transition rule to which the condition belongs as its arguments and returns whether the condition represented by a `Condition` object is satisfied by the element or not. The method `type()` takes an element and the transition rule to which the condition belongs as its arguments and transforms the element to a corresponding typed representation in a way that depends on the particular kind of condition.

Subsuming the conditions of transition rules under an abstract base class makes typing automata extensible with regard to expressiveness. Different kinds of conditions can be integrated with the basic typing automaton mechanism by subclassing `Condition` and providing the methods `evaluate()` and `type()` until all constructs offered by a schema definition language like MPEG-7 DDL are supported by typing automata as well.

For the beginning, the expressiveness of typing automata is restricted to bottom-up regular tree automata. Two kinds of conditions are introduced, namely simple content conditions and complex content conditions represented by the classes `SimpleContentCondition` and `ComplexContentCondition`, respectively. Essentially, a simple content condition refers to a simple type which is represented by the class `SimpleType` of TDOM's simple type framework. The condition is fulfilled if an element has simple content and if this simple content is a valid instance of the simple type referenced. A complex content condition consists of a Perl 5 string regular expression (kept in the attribute `regExp`) and is fulfilled if an element has complex content and the concatenation of the IDs of the element types applicable to the element's child elements satisfy the regular expression. Hence, both variants

of transition rules that are offered by traditional regular tree automata can be expressed using simple content conditions and complex content conditions within a typing automaton as well.

Constraint 10 imposes several structural restrictions on typing automata. Firstly, it is ensured that there exists at least one transition rule for every state of a typing automaton that features exactly that state as its result state. Otherwise a typing automaton would have states that would never be applied to an element. Secondly, it is ensured that every result state of a transition rule also occurs among the states of the typing automaton in which the transition rule is contained.

Constraint 10 (Typing automaton)

```
context TypingAutomaton
inv: state -> forAll(et | transitionRule -> exists(r |
      r.resultState = et))

context TransitionRule
inv: typingAutomaton.state -> includes(resultState)
```

The structural definition of typing automata is concluded with an example. The UML object diagram of Figure 7.6 depicts all transition rules of a typing automaton capturing the example `Melody` media description scheme. Just as with the regular tree automaton of Figure 7.2, a corresponding transition rule has been constructed for every element type declaration contained in the scheme. Each transition rule refers to the corresponding element type declared as its result state. Depending on whether the element type declaration defines a simple or complex content model, simple content conditions or complex content conditions have been created appropriately. Due to limitations of space, the figure refrains from using the full element type IDs as given by Figure 7.4 within the regular expressions of complex content conditions. Instead, `'et.etID'` is used as a placeholder for the ID of element type `et`.

7.2.2 Validation phase

Having specified the structure of typing automata, the specification can continue with the treatment of their behavior. As already mentioned, the consumption of an XML document in TDOM representation by a typing automaton proceeds in two phases. During the first of these phases, the validation phase, the element types which the elements of the document validly instantiate are inferred. A typing automaton does this in a way similar to bottom-up regular tree automata: the automaton attempts to apply all transition rules to each of the document's elements. *The element types serving as the result states of all those transition rules that are* applicable to a given element are called the element's applicable element types.

This notion is formally concretized by Definition 11. According to the definition, an element type `et` is applicable to an element `e` if and only if the name and namespace of `et` match the element type name and namespace of `e` and if the typing automaton has a transition rule which bears `et` as its result state and for which its condition evaluates to `true` for `e`.

Definition 11 (Applicable element types)

```
context TypingAutomaton def:
let applicableElementTypes(Element e) : Set(ElementType) =
```

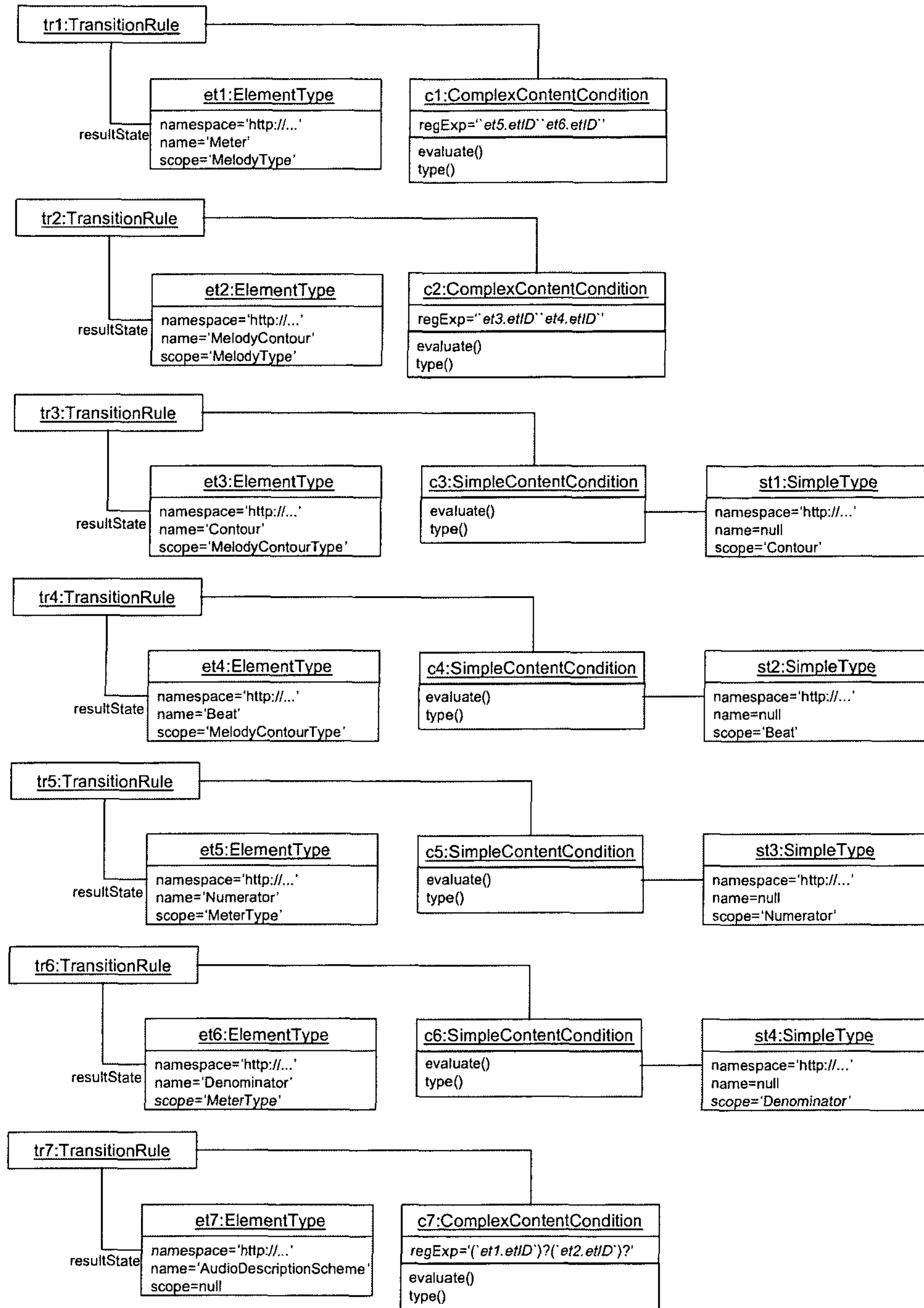


Figure 7.6: Example transition rules (UML object diagram)

```
states -> select(et | transitionRule -> exists(tr |
  et = tr.resultState and
  e.etName = et.name and
  e.etNamespace = et.namespace and
  tr.condition.evaluate(e, tr)))
```

An XML document is defined to be valid with regard to a typing automaton, if there exists an applicable element type for the document's root element that is not scoped, i.e., that is globally visible. This is formally expressed by Definition 12.

Definition 12 (Valid document)

```
context TypingAutomaton def:
let valid(Document d) : Boolean =
    applicableElementTypes(d.rootElement) -> exists (et |
     et.scope = null)
```

In Definition 11, much of the complexity of validating an XML document with regard to a typing automaton lies hidden within the method `evaluate()` of the abstract class `Condition`. For a complete specification, the respective implementation of this method for both kinds of conditions considered so far, simple content conditions and complex content conditions, has to be detailed.

Definition 13 specifies the behavior of the method `evaluate()` for the class `SimpleContentCondition`. The method checks whether an element has simple content and whether that simple content complies to the simple type referenced by the simple content condition. Taking a closer look at the postconditions contained in the definition, `evaluate()` returns `false` if the element passed as the method's argument does not have simple content, i.e., the element has either complex content or empty content. In case that the element is in typed representation and has simple content, `evaluate()` returns `true`, if and only if an instance of the simple type referenced by the simple content condition can be successfully constructed from the textual representation of the element's content using the simple type instance factory of TDOM's simple type framework that is associated with the simple type. In case that the element is in untyped representation and has simple content, `evaluate()` returns `true`, if and only if an instance of the simple type referenced by the simple content condition can be successfully constructed from the element's content.

Definition 13 (Evaluation of simple content condition)

```
context SimpleContentCondition::evaluate(Element e,
      TransitionRule tr) : Boolean
post: e.childNode -> notEmpty() or
      (e.typedSimpleContent -> isEmpty() and
      e.simpleContent = null) implies
       result = false
post: e.typedSimpleContent -> notEmpty() implies
       result = self.simpleType.simpleTypeInstanceFactory.
         fromString(e.typedSimpleContent.getSimpleType().
           simpleTypeInstanceFactory.
             toString(e.typedSimpleContent)) <> null
post: e.simpleContent <> null implies
       result = self.simpleType.simpleTypeInstanceFactory.
         fromString(e.simpleContent) <> null
```

Definition 14 specifies the behavior of the method `evaluate()` for the class `ComplexContentCondition`. The method checks whether an element has complex content and whether the IDs of the element types applicable to the element's child elements satisfy the string regular expression of the complex content condition. Closer inspecting the postconditions of the definition, `evaluate()` always returns

`false` if the element passed as the method's argument has simple content. If the element does not have simple content, `evaluate()` returns `true` if and only if there exists a sequence of element types applicable to the element's child elements for which holds that the sequence's signature, i.e., the concatenation of the element type IDs in the sequence, matches the condition's regular expression.[2]

Definition 14 (Evaluation of complex content condition)

```
context ComplexContentCondition::evaluate(Element e,
      TransitionRule tr) :  Boolean
post: e.typedSimpleContent -> notEmpty() or
      e.simpleContent <> null implies
        result = false
post: e.typedSimpleContent -> isEmpty() and
      e.simpleContent = null implies
        result = tr.typingAutomaton.
          applicableChildElementTypes(e) -> exists(acet |
            tr.typingAutomaton.signature(acet).matches(regExp))
```

Definition 15 serves to clarify the meaning of the construct `applicableChildElementTypes(e)` used in the previous definition to denote sequences of element types applicable to the child elements of a given element `e`. More precisely, `applicableChildElementTypes(e)` refers to the set of all possible sequences that have the same size as the sequence of child elements of `e` and whose members satisfy the following conditions: if the set of applicable element types for a given child element of `e` is not empty, then the member of the sequence at the position corresponding to the position of the child element below `e` must be one of these applicable element types. If the set of applicable element types for a given child element of `e` is empty, then the member of the sequence at the position corresponding to the position of the child element must be the child element itself.

Definition 15 (Applicable child element types)

```
context TypingAutomaton def:
let applicableChildElementTypes(Element e) :  Set(Sequence(OclAny)) =
    Sequence(OclAny).allInstances -> select(seq |
      seq -> size() = e.childElements -> size() and
      Sequence{1..seq -> size()} -> forAll(i |
        (applicableElementTypes(e.childElements -> at(i)) ->
          notEmpty() implies
          applicableElementTypes(e.childElements -> at(i)) ->
            contains(seq -> at(i)) and
        (applicableElementTypes(e.childElements -> at(i)) ->
          isEmpty() implies
          e.childElements -> at(i) = seq -> at(i))))
```

For the sake of completeness, Definition 16 finally provides us with the specification of the signature of a sequence of applicable child element types as employed within Definition 14. This signature is simply the concatenation of all element type IDs and unknown element type IDs of all element types and elements contained in that sequence, respectively.

[2]For the definition, it is assumed that the type `String` predefined by OCL supplies the operation `matches` which evaluates a given string against a Perl 5 regular expression and returns `true` if and only if the string constitutes a valid word of the language defined by that regular expression.

Definition 16 (Signature)

```
context TypingAutomaton def:
let signature(Sequence(OclAny) seq) : String =
    seq -> iterate(
     obj : OclAny;
     res : String = "";
     if obj.oclIsTypeOf(ElementType) then
       res.concat(obj.etID)
     elseif obj.oclIsTypeOf(Element) then
       res.concat(obj.uetID)
     endif
    )
```

One might get the impression that due to the mutually recursive definition of `applicableElementTypes()` and `applicableChildElementTypes()` via the indirection of the method `evaluate()` of the class `ComplexContentCondition`, the behavior of a typing automaton during the validation phase consitutes a form of top-down processing. However, one should consider that, according to these definitions, the recursion immediately descends down to the leaf elements of a document without performing any calculations; the applicable element types are not inferred until the recursion ascends back up the document on its way from the leaves. Thus, not denying its origin from bottom-up regular tree automata, a typing automaton's behaviour during the validation phase rather has to be considered as bottom-up processing.

7.2.3 Typing phase

If a *typing automaton has succeeded in validating* an XML *document in* TDOM representation and inferring the element types applicable to the document's element during the validation phase, it enters its second phase of processing, the so-called typing phase. Starting out from the root element in a top-down manner, the automaton uses the applicable element types inferred during the validation phase to transform the individual elements of the document to corresponding typed representations.

Figure 7.7 illustrates the typing phase of a typing automaton using the example MPEG-7 `Melody` media description. Beginning at the root element, the typing automaton selects an applicable unscoped element type for the root element – `et7` in this case as it is the only one available – and uses this element type to bring the root element into a corresponding typed representation (1). Having transformed the root element to typed representation, the automaton proceeds with the root's child elements and selects one of their applicable element types to produce corresponding typed representations as well (2). In that fashion, the automaton continues on descending down the document (3) until the leaf elements of the document have been reached and transformed to corresponding typed representations (4).

The core of a typing automaton's behavior during the typing phase is given by Definition 17. This definition formally introduces the method `typeElement()` of the `TypingAutomaton` class. The method is passed an element `e` and an element type `et` as its parameters. As specified by the pre- and postconditions in the definition, the method transforms `e` to a corresponding typed representations on the basis of `et` provided that `e` is in untyped representation and that `et` is applicable to `e`. The transformation recursively brings as much of the child elements of `e` as possible to corresponding typed representations. More precisely, there are only two

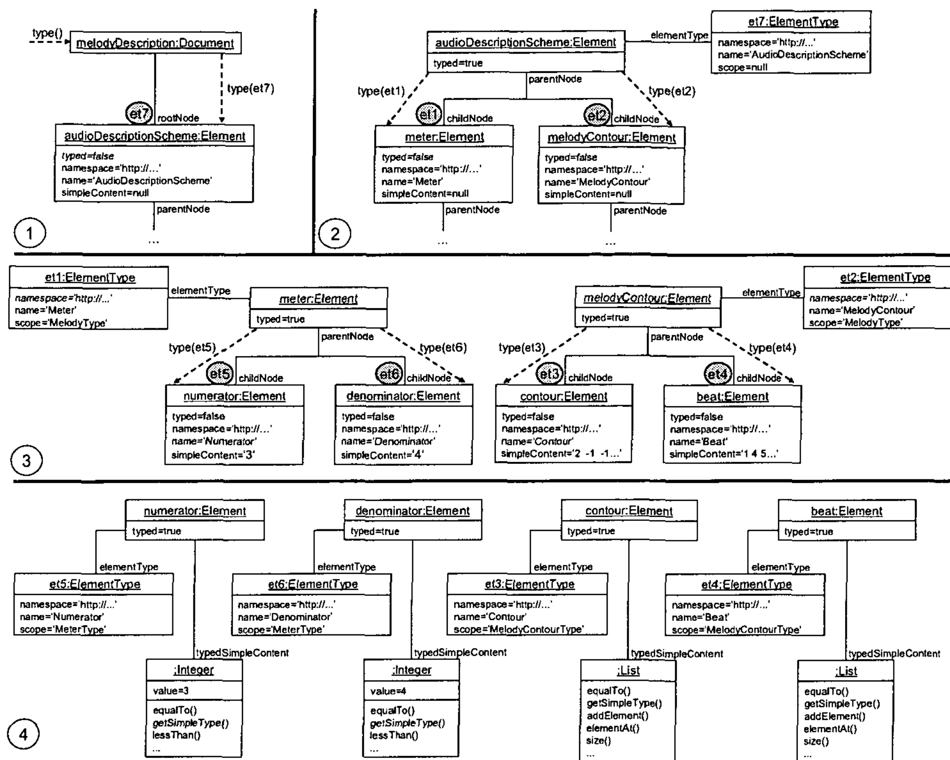


Figure 7.7: Typing phase (UML object diagrams)

cases in which one of the direct or indirect child elements of `e` is not transformed to a corresponding typed representation using one of its applicable element types: in case that the child element has no such applicable element types or in case that its parent element has not been transformed to typed representation either. Consideration of the latter case is necessary because it can happen that an element has applicable element types while its parent element has not. As a corresponding typed representation of the parent element thus cannot be constructed and TDOM does not allow an element in typed representation to appear among the child nodes of an element in untyped representation, a corresponding typed representation of the element itself also cannot be created.

The pseudocode given in the definition proposes a simple algorithm for the implementation of this method that consists of two major steps: in the first step, a transition rule of the typing automaton is selected that has decided in the validation phase that `et` is applicable to `e`. I.e., the chosen transition rule must return `et` as its result state and its condition must evaluate to `true` for `e`. In the second step, the element is passed on to the `type()` method of the transition rule's condition which brings it into a corresponding typed representation in a way that depends on the particular kind of condition.

Definition 17 (Typing elements)

```
context TypingAutomaton::typeElement(Element e, ElementType et)
pre:  not(e.typed)
pre:  self.applicableElementTypes(e) -> includes(et)
post: e.CUR(e@pre)
```

```
post: e.elementType = et
post: e.allChildElements -> forAll(c |
        not(c.CUR(c@pre) and
        self.applicableElementTypes(c) -> includes(c.elementType)) implies
          self.applicableElementTypes(c) -> isEmpty() or
          not(c.parentNode.typed))
pseudocode:
    -- Find a transition rule which decides that the element
    -- type is applicable
    tr := self.transitionRule -> any(tr1 |
      tr1.resultState = et and
      tr1.condition.evaluate(e, tr1))
    -- Type element according to the transition rule's condition
    tr.condition.type(e, tr)
```

Note that the proposed algorithm bears a source of inefficiency if implemented naively. It includes the selection of a transition rule deciding that `et` is applicable to `e`. Simply realizing this step by checking all transition rules of the typing automaton until one is found that delivers `et` as its result state and whose condition evaluates to `true` for `e` is not very efficient: this calculation has already been performed during the validation phase – not to mention the fact, that the repeated evaluation of a complex content condition might involve the inference of the applicable element types of `e`'s child elements which has already been done in the validation phase as well.

Nevertheless, the `typeElement()` method can be realized efficiently at the expense of main memory without needing to change the overall structure of the proposed algorithm. While traversing an XML document from the bottom-up inferring the applicable element types during the validation phase, the typing automaton can cache for each element of the document (a) its applicable element types and (b) the transition rule which decided that an element type is applicable. With these data at hand, the selection in question merely constitutes a simple cache lookup operation.

Definition 18 specifies common characteristics of the `type()` method that have to be fulfilled by every implementation of that method in the subclasses of `Condition`, even though the concrete behaviour of these implementations depends on the particular kind of condition. According to the pre- and postconditions given by the definition, all implementations of `type()` have in common that whenever they are passed an element in untyped representation as their first argument which has the same element type name and namespace as the element type acting as the result state of the transition rule passed as their second argument and for which the condition evaluates to `true`, they bring the element and as much of its direct and indirect child elements as possible into a corresponding typed representation on the basis of the result state.

Definition 18 (Typing functionality of Conditions)

```
context Condition::type(Element e, TransitionRule tr)
pre:  not(e.typed)
pre:  tr.resultState.name = e.etName
pre:  tr.resultState.namespace = e.etNamespace
pre:  self.evaluate(e, tr)
post: e.CUR(e@pre)
post: e.elementType = tr.resultState
post: e.allChildElements -> forAll(c |
        not(c.CUR(c@pre) and tr.typingAutomaton.
```

```
applicableElementTypes(c) -> includes(c.elementType)) implies
tr.typingAutomaton.applicableElementTypes(c) -> isEmpty() or
not(c.parentNode.typed))
```

Definition 19 proposes an algorithm for the implementation of the `type()` method for simple content conditions. The algorithm transforms elements with simple content, that are in untyped representation and foe which a given simple content condition evaluates to `true`, into a suitable corresponding typed representation. Following the pseudocode given by the definition, `type()` first changes the basic structure of the passed element to typed representation, i.e., the element's `type` attribute is set to `true`, the `name` and `namespace` attributes are set to `null`, and the element is associated with the element type acting as the result state of the passed transition rule. Then, employing the factory of the simple type associated with the simple content condition, an appropriate simple type instance for the simple element content is constructed and linked with the element. Note that this is always possible as the preconditions of the `type()` method assure that a call of the `evaluate()` method of the condition yields `true`; `evaluate()` already checks whether a simple type instance can be constructed for the element content.

Definition 19 (Typing elements with simple content)

```
context SimpleContentCondition::type(Element e, TransitionRule tr)
pre:  not(e.typed)
pre:  tr.resultState.name = e.etName
pre:  tr.resultState.namespace = e.etNamespace
pre:  self.evaluate(e, tr)
post: e.CUR(e@pre)
post: e.elementType = tr.resultState
post: e.allChildElements -> forAll(c |
        not(c.CUR(c@pre) and tr.typingAutomaton.
          applicableElementTypes(c) -> includes(c.elementType)) implies
          tr.typingAutomaton.applicableElementTypes(c) -> isEmpty() or
          not(c.parentNode.typed))
pseudocode:
    -- Bring element to appropriate typed representation
    e.typed := true
    e.name := null
    e.namespace := null
    e.elementType := e.elementType -> including(tr.resultState)
    -- Use the simple type instance factory associated
    -- with the simple type of the condition to produce
    -- an appropriate simple type instance for use as
    -- typed element content
    stif := self.simpleType.simpleTypeInstanceFactory
    sti := stif.fromString(e.simpleContent)
    -- Set simple type instance as simple content of element
    e.simpleContent := null
    e.typedSimpleContent := e.typedSimpleContent ->
      including(sti)
```

Definition 20 covers an algorithm for the implementation of the `type()` method for complex content conditions. The algorithm transforms elements with complex content, that are in untyped representation and on which a given complex content condition evaluates to `true`, into a suitable corresponding typed representation. Similar to the `type()` method for simple content conditions, the implementation

first changes the basic structure of the passed element to typed representation. Then, `type()` chooses a sequence of element types applicable to the element's child elements that satisfies the condition, i.e., whose signature matches the Perl 5 string regular expression of the condition. Again, this is always possible. The `evaluate()` method already checks the existence of such a sequence. At last, the child elements of the element passed as the method's parameter are brought to corresponding typed representations in a way that the complex content condition remains satisfied. This is achieved by synchronously iterating over the chosen sequence of applicable child element types and the sequence of child elements. In case that the current member of the sequence of applicable child element types is an element type, the method uses this element type to bring the corresponding member in the sequence of child elements into typed representation by recursively calling the typing automaton's `typeElement()` method.

Definition 20 (Typing elements with complex content)

```
context ComplexContentCondition::type(Element e, TransitionRule tr)
pre:  not(e.typed)
pre:  tr.resultState.name = e.etName
pre:  tr.resultState.namespace = e.etNamespace
pre:  self.evaluate(e, tr)
post: e.CUR(e@pre)
post: e.elementType = tr.resultState
post: e.allChildElements -> forAll(c |
        not(c.CUR(c@pre) and tr.typingAutomaton.
          applicableElementTypes(c) -> includes(c.elementType)) implies
          tr.typingAutomaton.applicableElementTypes(c) -> isEmpty() or
          not(c.parentNode.typed))
pseudocode:
    -- Bring element to appropriate typed representation
    e.typed := true
    e.name := null
    e.namespace := null
    e.elementType := e.elementType -> including(tr.resultState)
    -- Choose a suitable sequence of applicable element
    -- types for the element's child elements.
    -- The sequence's signature must match the regular
    -- expression of the condition
    acet := tr.typingAutomaton.
      applicableChildElementTypes(e) -> any (acet1 |
        tr.typingAutomaton.contentSignature(acet1).
          matches(self.regExp))
    -- Type child elements according to the applicable content
    -- element types
    foreach i in Sequence{1..acet -> size()} do
      if acet -> at(i).oclIsTypeOf(ElementType) then
        tr.typingAutomaton.typeElement(e.childElements -> at(i),
          acet -> at(i))
      endif
    endforeach
```

The proposed implementation of the `type()` method once more contains a possible source of inefficiency. Selecting a suitable sequence of applicable element types for the child elements of the passed element such that the complex content condition evaluates to `true` for that sequence is inefficient if implemented naively: this

calculation has already taken place during the typing phase and repetition of that calculation implies the repeated inference of the applicable element types of the child elements.

But again, the method can be realized efficiently at the expense of memory without the need of changing the overall structure of the proposed algorithm. During the validation phase, the typing automaton can not only cache for each element of the document its applicable element types and the transition rule which decided applicability but also, in case that the transition rule bears a complex content condition, the sequence of child element types for which the complex content condition evaluated to `true`. This way, the selection in question constitutes a cache lookup operation.

So far, the behavior of typing automata has been separated into a validation phase and a typing phase. What is still missing is a central entry point to a typing automaton's behavior that interconnects both phases. Such an entry point is given by Definition 21. The definition provides the specification of the method `type()` of the class `TypingAutomaton`. This method takes an XML document in TDOM representation as its parameter.

As specified by the postconditions in the definition, the method transforms as much elements as possible to corresponding typed representations if the document is valid with regard to the typing automaton represented by the current `TypingAutomaton` object. More specifically, there are only three cases in which an element is allowed not to constitute a corresponding typed representation compared to its representation prior to the call of the method: the first case is that element has already been in typed representation before the call and is still in typed representation on the basis of one of its applicable element types. The second case is that the element is in untyped representation and has no applicable element types. The third case is that not only the element is in untyped representation but also its parent element. Should the document be invalid with regard to the typing automaton, `type()` transforms all elements of the document to untyped representation.

The pseudocode given in the definition shows a straightforward implementation of this method. In order to obtain a clean basis for processing, the method first brings all elements to untyped representation by calling the `untype()` method on the root element. Next, the method initiates the validation phase by checking the validity of the document and, in doing so, inferring the applicable element types for the document's elements. If the document is valid, the method proceeds to the typing phase and selects an unscoped element type applicable to the root element and employs it to create a corresponding typed representation of the root element via `typeElement()`.

Definition 21 (Typing documents)

```
context TypingAutomaton::type(Document d)
post: self.valid(d) implies
        Element.allInstances -> forAll(e |
          e.document = d and not(e.CUR(e@pre) and
          self.applicableElementTypes(e) -> includes(e.elementType)) implies
            (e.typed@pre and e.typed and
            self.applicableElementTypes(e) -> includes(e.elementType)) or
            (not(e.typed) and
            self.applicableElementTypes(e) -> isEmpty()) or
            (not(e.typed) and e.parentNode -> notEmpty() and
            not(e.parentNode.typed)))
post: not(self.valid(d)) implies
```

```
        not(d.rootElement.typed)
pseudocode:
    -- Untype document if root element is typed
    if d.rootElement.typed then
      d.rootElement.untype()
    endif
    -- Select an arbitrary applicable unscoped element type to type
    -- root element with if possible.
    if self.valid(d) then
      rootElementType := self.applicableElementTypes(d.rootElement) ->
        any(et | et.scope = null)
      -- Type root element
      self.typeElement(d.rootElement, rootElementType)
    endif
```

7.3 Computational Complexity

Having provided the core specification of typing automata, it is useful to obtain an indicator for the computational complexity of their behavior. Therefore, this section estimates an upper bound for the running time of the `type()` method of the `TypingAutomaton` class given by Definition 21. This bound will be expressed in terms of the number of elements n contained in the XML document that is to be typed and the number of transition rules t of the typing automaton used for typing.

The estimation makes two assumptions: firstly, it is assumed that a typing automaton caches the element types applicable to the elements of the document during the validation phase along with the transition rules that decided applicability and the sequences of child element types what were applied to these transition rules. This has already been suggested in Section 7.2.3 for the implementation of the `typeElement()` and `type()` methods of the classes `TypingAutomaton` and `ComplexContentCondition`. Such a caching ensures that applicable element types only need to be calculated once for each element and that the complexity of repeated access to the cached results of this calculation is negligible for the estimation, i.e., $O(1)$, if applying a suitable hashing technique.

Secondly, it is assumed that a typing automaton is deterministic, i.e., the number of applicable element types for each element is at most one. This restriction does not imply a loss of generality. It has been proven in literature that the classes of non-deterministic and deterministic bottom-up regular tree automata are equivalent to each other [Chi00, CDG+02]: for each non-deterministic bottom-up regular tree automaton an equivalent deterministic one can be algorithmically constructed. Given the structural similarity between bottom-up regular tree automata and typing automata – both types of automata support the same kinds of transition rules and the mapping between them is straightforward as it has been illustrated by means of Figures 7.2 and 7.6 – this result also applies to typing automata.

Besides, unambiguousness is a natural quality criterion for schema design. It is no surprise that most schema definitions for XML documents occurring in practice are intuitively designed to be unambiguous and thus straightforwardly translate to deterministic typing automata. The example `Melody` media description scheme of Figure 2.4 and its typing automaton representation given by Figure 7.6 perfectly illustrate this point.

Given these assumptions, the running time of the `type()` method of the class `TypingAutomaton` in terms of n and t can be expressed as follows:

$$T(n,t) = \sum_{i=1}^{n} U_{e_i} + \sum_{i=1}^{n} A_{e_i} + \sum_{i=1}^{n} C_{e_i} \tag{7.1}$$

Equation 7.1 becomes clear when taking a look at the algorithm proposed in Definition 21. The algorithm first brings all elements of the XML document that is to be typed to untyped representation, then initiates the validation phase in which the element types applicable to the elements are inferred, and finally starts the typing phase in which typed representations of the elements are produced according to the inferred element types. For each of the document's elements e_i, $i = 1 \ldots n$, a typing automaton thus spends the running times U_{e_i} to bring it to untyped representation, A_{e_i} to infer its applicable element types, and C_{e_i} to create a corresponding typed representation of e_i on the basis of an applicable element type.

Since the production of a corresponding untyped representation of a single element e_i – if at all required because e_i might already be in untyped representation – merely involves changes to the attribute values of the `Element` object representing e_i and the associations it participates in, the required running time U_{e_i} is independent of the number of elements n in a document and the number of transition rules t of a typing automaton. Hence:

$$U_{e_i} = O(1), i = 1 \ldots n \tag{7.2}$$

Similarly, the production of a corresponding typed representation of a single element e_i on the basis of an applicable element type mainly involves changes to the `Element` object representing e_i that are independent of n and t. Moreover, since a caching of applicable element types is assumed, the selection of the particular applicable element type and transition rule that is used for the creation of the corresponding typed representation constitutes an effort that is independent of n and t as well. Therefore:

$$C_{e_i} = O(1), i = 1 \ldots n \tag{7.3}$$

Inserting Equations 7.2 and 7.3 into Equation 7.1 yields:

$$T(n,t) = \sum_{i=1}^{n} O(1) + \sum_{i=1}^{n} A_{e_i} + \sum_{i=1}^{n} O(1) = O(n) + \sum_{i=1}^{n} A_{e_i} \tag{7.4}$$

The estimation of an upper bound for the running time A_{e_i} that has to be spent for the inference of the applicable element types of element e_i during the validation phase is more complicated. Basically, a typing automaton attempts to apply all of its t transition rules to e_i. In case that a transition rule has a simple content condition, the test for the transition rule's applicability mainly involves checking whether e_i has simple content and whether a valid simple type instance can be constructed from the textual representation of e_i's simple content (see Definition 13). This is independent of the number of elements n in the document and the number of transition rules t of the typing automaton and thus can be estimated with $O(1)$.

In case that a transition rule has a complex content condition, the test for the transition rule's applicability mainly involves checking whether e_i has complex content and evaluating a string regular expression on the signature of the sequence of applicable child element types of e_i (see Definition 14). As it is assumed that the typing automaton is deterministic, there exists only one such signature. It is a well-known fact that the evaluation of string regular expressions on a string

takes linear time with regard to the length of the string [ASU86]. Since the length of the signature of the sequence of applicable child element types of e_i is roughly proportional to the number of e_i's child elements c_{e_i}, the running time for checking a complex content condition should not exceed $O(c_{e_i})$ in practice.

Therefore, A_{e_i} can be bounded as follows:

$$A_{e_i} = t \max(O(c_{e_i}), O(1)) = t\, O(c_{e_i}), i = 1 \ldots n \tag{7.5}$$

Inserting Equation 7.5 into Equation 7.4 delivers:

$$T(n,t) = O(n) + t \sum_{i=1}^{n} O(c_{e_i}) \tag{7.6}$$

As in general $\sum_{i=1}^{n} O(f_i(n)) = O(\sum_{i=1}^{n} f_i(n))$ [CLR90], Equation 7.6 becomes:

$$T(n,t) = O(n) + t\, O(\sum_{i=1}^{n} c_{e_i}) \tag{7.7}$$

Since the only element among the n elements contained in an XML document that is not a child element of another and that is thus not covered by the sum $\sum_{i=1}^{n} c_{e_i}$ is the root element, it follows that $\sum_{i=1}^{n} c_{e_i} = n - 1$. Hence, Equation 7.7 can be rewritten as:

$$T(n,t) = O(n) + t\, O(n-1) = O(t\, n) \tag{7.8}$$

Given the bound of Equation 7.8, it can be stated that, subject to the preliminary assumptions concerning caching and determinism, the running time of the `type()` method of the class `TypingAutomaton` never grows more than linearly with the number of transition rules t of the typing automaton on which the method is executed. Its running time also never grows more than linearly with the number of elements n in the XML document that is passed to `type()`. Given this linearity, one can conclude that a typing automaton's behaviour is reasonably efficient to allow the application of even complex typing automata with large numbers of transition rules to large XML documents.

7.4 Optimizations

Having discussed of the computional complexity of the behavior of typing automata, it is now time to spend some thoughts on possible optimizations. The proposed algorithm for the `type()` method of the class `TypingAutomaton` (see Definition 21), which validates a document and transforms as much elements as possible to corresponding typed representations, is suboptimal in many practical cases. The algorithm behaves reasonably well when applied to an XML document whose TDOM representation consists of elements in untyped representation only. This is typically the case during the import of a document to TDOM where usually a TDOM representation based solely on untyped representations is produced as a first step, before applying a typing automaton in order to validate the document and to create typed representations. In such a situation, the algorithm traverses the elements contained in the document from the bottom up in order to calculate their applicable element types while checking the document's validity. Assuming a caching as described above, the algorithm then traverses the elements from the top down for a second time and brings them to corresponding typed representations.

But when applied to a document which does not just contain elements in untyped representation, the behaviour of the algorithm is less reasonable. Such a situation might occur during the update of an XML document where only parts of the document have been temporarily transformed to untyped representation in order to decouple them from the schema definition and now need to be brought back to typed representation. In this case, the algorithm first brings all elements of the document to untyped representation as a first step before continuing on as supplied before. It is rather obvious that this behaviour is far from perfect as it ignores the typing results of previous runs of the typing automaton on the document just because potentially very small fractions of a document have been changed and brought to untyped representation during the update. Especially for large documents, validation and production of typed representations might have consumed considerable processing power that should not be thrown away carelessly.

Therefore, an alternative implementation of the `type()` method of `TypingAutomaton` similar in idea to [PV03] is now proposed that makes use of already existing typed representations of elements. This variant is called local document typing as it aims at limiting the effects of bringing the document's elements from untyped to corresponding typed representations to the immediate vicinity of the elements in untyped representation. It attempts to preserve already existing typed representations of elements thereby significantly reducing the number of elements that need to be traversed for the purpose of document typing in many cases occurring in practice.

Figure 7.8 illustrates the different steps of the local document typing approach at hand of an excerpt of the example `Melody` media description known from Figure 2.5 in TDOM representation. The excerpt covers the `Meter` element and its child elements. It is assumed that all elements of the media description are in typed representation with the exception of the `Numerator` element which is in untyped representation because its simple content has been changed to `5` during an update operation. It is now intended to bring as much elements of the description as possible, especially the `Numerator` element of course, to appropriate typed representations by employing the typing automaton that represents the `Melody` media description scheme with transition rules as given by Figure 7.6.

To achieve that aim, local document typing starts the typing process at the topmost untyped elements of the document. These are all those elements that are in untyped representation but whose parent elements are in typed representation or, in case that the root element is in untyped representation already, the root element itself. The topmost untyped elements subsume all elements of the document that need to be brought to corresponding typed representations as their direct and indirect child elements. They further constitute the boundary to those parts of the document which already are in typed representation and which ideally should be affected by the typing process as little as possible.

In the figure, the single topmost untyped element is the `Numerator` element. Local document typing picks up that element and memorizes the element type of its parent element, namely the element type `Meter` represented by the `ElementType` object `et1` (1). The parent element, and with it all of its direct or indirect child elements, is transformed to a corresponding untyped representation (2). Then, its applicable element types are determined (3). Since the memorized element type `Meter` still occurs among its applicable element types, the parent element, and with it again all of its direct or indirect child elements, is brought back immediately to a corresponding typed representation on the basis of `Meter` (4). There is no need for any further processing: the implicit assumption underlying the typed represen-

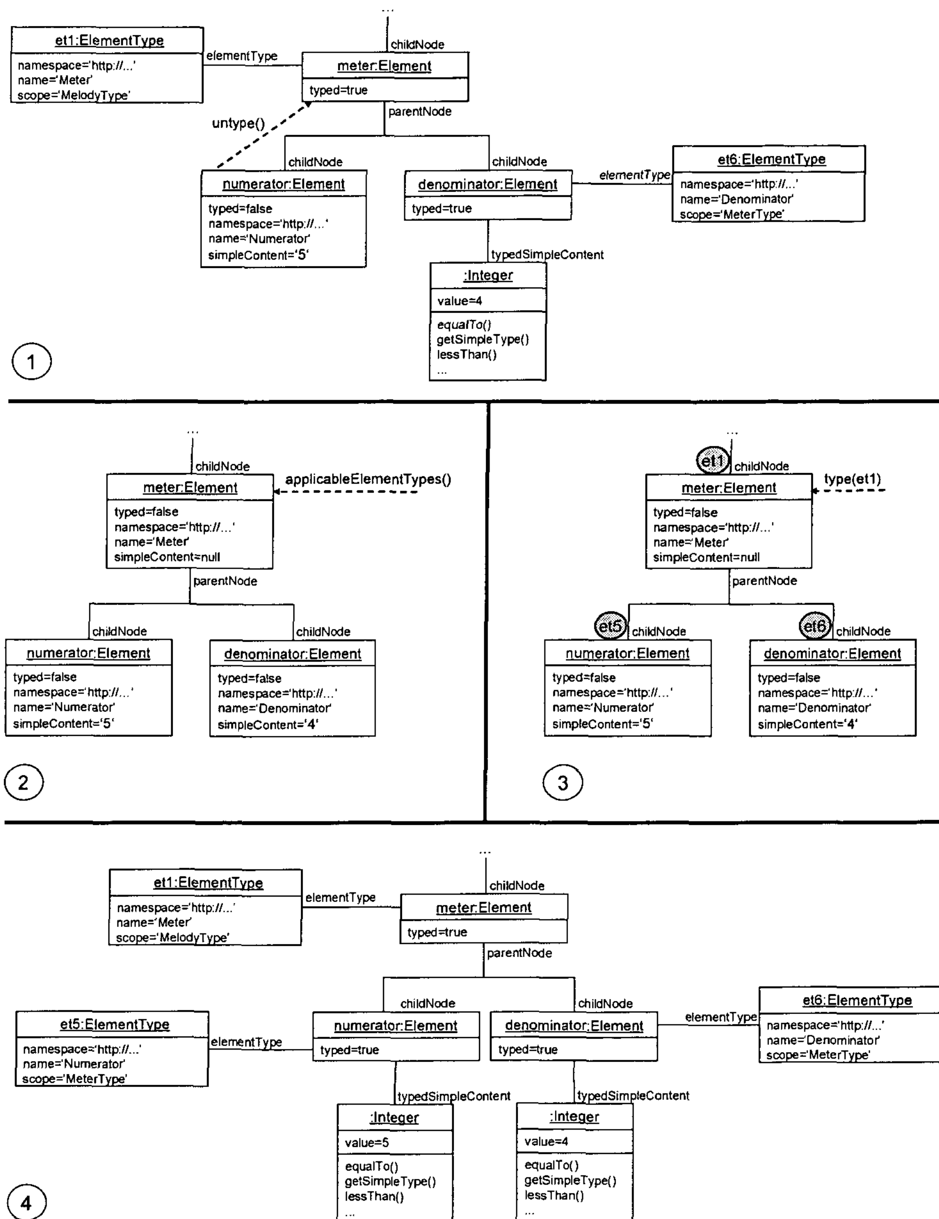


Figure 7.8: Local document typing (UML object diagrams)

tations of the elements located above the parent element in the document hierarchy originating from previous runs of the typing automaton is that the parent element validly instantiates `Meter` – which it still does.

Figure 7.9 illustrates the behaviour of local document typing in case that the memorized element type of a topmost untyped element's parent element is no longer applicable. For this purpose, we assume that the simple content of the `Numerator` element has been set to the nonsense-string `invalid` instead of `5` during the update operation. In the beginning, local document typing proceeds as usual by memorizing the element type of the parent of the `Numerator` element (1) and by transforming

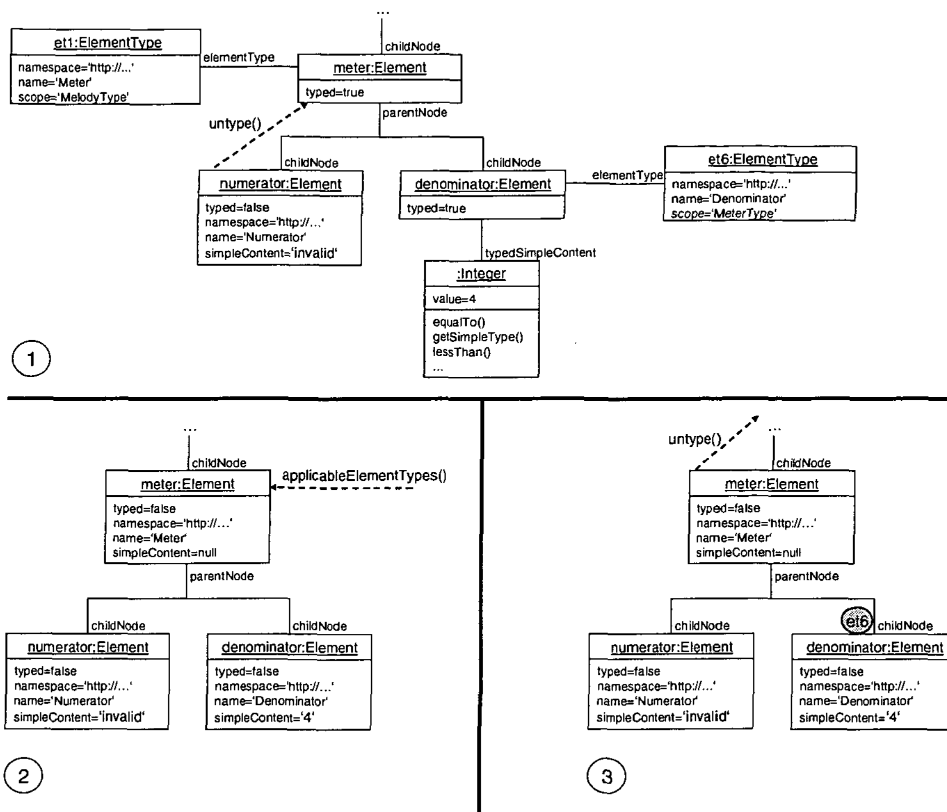


Figure 7.9: Failing local document (UML object diagrams)

the parent element to a corresponding untyped representation (2). When determining the applicable element types of the parent element, however, one finds that the memorized element type is no longer applicable (3). The parent element even does no longer have any applicable element types. This is due to the fact that the `Numerator` element has no applicable element types (the only transition rule of the typing automaton potentially suitable for the `Numerator` element, `tr5`, expects integer content according to Figure 7.6) and thus transition rule `tr1` of the typing automaton is no longer satisfied by the parent element as well. As a consequence, the implicit assumption underlying the typed representation of the element located above the parent element in the document hierarchy that the parent element constitutes a valid instantiation of the element type `Meter` does not hold anymore.

Local document typing responds to this situation by considering the parent element as the new topmost untyped element and relaunches processing as supplied above. In the worst case (which happens to occur in the example), this may result in a cascading untyping of parent elements until the root element of the document is reached and transformed to a corresponding untyped representation (and with it all elements of the document). Local document typing then checks whether there exists an applicable unscoped element type for the root element. If it does, it constructs a corresponding typed representation of the root element on the basis of the unscoped element type. If it does not, the document is invalid and all of its elements remain in untyped representation.

In the following, an alternative implementation of the method `type()` of the class `TypingAutomaton` that realizes the local document typing algorithm is specified. As a prelimary, the notion of the topmost untyped elements of a document is formalized in Definition 22.

Definition 22 (Topmost untyped elements)

```
context Document def:
let topmostUntypedElements : Set(Element) =
    Element.allInstances -> select(e |
      not(e.typed) and e.document = self and
      (e.parentNode -> isEmpty() or
      e.parentNode.typed))
```

Definition 23 provides the pseudocode describing the new implementation of the `type()` method. Throughout the implementation, a set of elements that are to be brought to typed representation is maintained. This set is initialized with the topmost untyped elements of the document. As long as there are still elements in this set and the document has not been found to be invalid, one of these elements is selected successively. For each selected element, the implementation distinguishes whether the element constitutes the root element of the document or not. In case that the selected element is the root element, the implementation behaves exactly like the conventional implementation of the `type()` method of Definition 21: it chooses an applicable unscoped element type and brings the root element to a corresponding typed representation accordingly. If such an element type exists, typing of the document is finished; if not, the document is considered invalid and processing terminates.

In case that the selected element is not the root element of the document, the element type of element's parent is stored in a temporary variable and the parent element is transformed to a corresponding untyped representation. If the stored element type is still applicable to the parent element, it is transformed back to a corresponding typed representation using that element type. Any of the parent element's child elements potentially existing in the set of elements that are to be brought to typed representation are removed from that set: their typing has been already been covered by the construction of the corresponding typed representation of their parent.

If the stored element type is no longer applicable to the parent element, the parent element remains in untyped representation. It is further added to the set of elements to be brought to typed representation. Again, its child elements are removed from this set as well, since their typing will be covered by the typing of the parent element.

Definition 23 (Local variant of document typing)

```
context TypingAutomaton::type(Document d)
post: self.valid(d) implies
        Element.allInstances -> forAll(e |
          e.document = d and not(e.CUR(e@pre) and
          self.applicableElementTypes(e) -> includes(e.elementType)) implies
            (e.typed@pre and e.typed and
            self.applicableElementTypes(e) -> includes(e.elementType)) or
            (not(e.typed) and
            self.applicableElementTypes(e) -> isEmpty()) or
            (not(e.typed) and e.parentNode -> notEmpty() and
```

```
        not(e.parentNode.typed)))
post: not(self.valid(d)) implies
      not(d.rootElement.typed)
pseudocode:
   -- Assume valid document
   invalid := false
   -- Retrieve the elements that need to be brought to typed
   -- representation
   toType := d.topmostUntypedElements
   -- As long as the document is not invalid, iteratively
   -- select one these elements
   while toType -> notEmpty() and not(invalid) do
    -- Select arbitrary element for typing
    element := toType -> any(true)
    if element = d.rootElement then
     -- The root element needs to brought to typed representation.
     -- Use applicable unscoped element type for that purpose
     if not(self.applicableElementTypes(element) ->
       exists(et | et.scope = null)) then
       -- As there is no such element type, the document
       -- is invalid
       invalid := true
     else
       -- Create corresponding typed representation of the
       -- root element using the unscoped element type.
       rootElementType := self.applicableElementTypes(element) ->
         any(et | et.scope = null)
       self.typeElement(element, rootElementType)
       -- Typing of the document is finished
       toType := Set{}
     endif
    else
     -- Memorize the type of the chosen element's parent
     parentType := element.parentNode.elementType
     -- Bring parent element to untyped representation
     element.parentNode.untype()
     if self.applicableElementTypes(element.parentNode) ->
       includes(parentType) then
       -- As the memorized element type is still applicable,
       -- bring the parent element to a corresponding typed
       -- representation on the basis of that type
       self.typeElement(element.parentNode, parentType)
       -- Ignore all of the parent's child elements for the
       -- further creation of typed representations.
       toType := toType -> excludingAll(
         element.parentNode.childNode)
     else
       -- The memorized element type is no longer applicable.
       -- Add the parent element to the set of elements that
       -- are to be brought to typed representation.
       toType := toType -> including(element.parentNode)
       -- Ignore all of the parent's child elements for
       -- the further creation of typed representations
       toType := toType -> excludingAll(
         element.parentNode.childNode)
```

```
        endif
      endif
    endwhile
```

It is noteworthy that, assuming that a typing automaton caches for each element the applicable element types, the respective transition rules that decided applicability, as well as the sequence of child element types for which the transition rules evalutated to true as suggested before, the local variant of document typing never performs worse than traditional document typing given by Definition 21. The worst case for local document typing occurs when the single topmost untyped element of a document is a leaf element for which the existing typed representations of all its direct and indirect parent elements, including the root element, cannot be preserved. In this situation, local document typing essentially performs two major operations at every element while ascending from the leaf element to the document root from the bottom up: the first operation is that every direct or indirect parent element of the leaf element, and with it recursively all of its child elements, is brought to a corresponding untyped representation. This implies that once local document typing has arrived at the root element, all elements of the document have been transformed to corresponding untyped representations. As the implementation of the method `untype()` of the class `Element` as proposed by Definition 8 cancels its recursion whenever hitting an element that already is in untyped representation, it is assured that every element is only brought to untyped representation once. The second operation is that the applicable element types of every parent element traversed and its child elements are inferred. If these are cached by the typing automaton as assumed, inference has also taken place only once for each element when the root element has been reached. Given that there exists an unscoped element type applicable to the root element, local document typing finally performs a third major operation on every element of the document: it uses this element type to bring the root element and recursively the other document's elements to corresponding typed representations.

In this worst case situation for local document typing, traditional document typing basically performs the same three major operations on each element as well, but only in different order: since the root element is in typed representation (the single topmost untyped element is a leaf element), every element with the exception of the leaf element is brought to a corresponding untyped representation. Then, the applicable element types of the root element, and with these recursively the applicable element types for all elements of the document, are inferred. Finally, the root element and the other elements of the document are brought to corresponding typed representations.

In the best case for traditional document typing, i.e., all elements of a document are in untyped representation, local document typing does not exceed the complexity of traditional document typing either. As in this case the topmost untyped element of the document is the root element, local document typing behaves exactly the same as traditional document typing.

In many other cases however – especially after document updates during which only small fractions of a document have been changed to untyped representation – local document typing can be expected to perform substantially more efficient than traditional document typing because existing typed representations are preserved if possible. Thereby, the number of elements for which corresponding typed and untyped representations are created and applicable element types are inferred can often be reduced. As a consequence, local document typing is in any case preferable to traditional document typing.

7.5 Extensions

Up to this point, typing automata have been restricted to the expressiveness of regular tree automata by supporting only two kinds of conditions within transition rules: simple content conditions and complex content conditions. Apart from traditional simple content and complex content declarations, however, MPEG-7 DDL permits the use of additional constructs for declaring the content models of the element types and attributes of a schema definition. As it is the aim to use typing automata as an intermediary representation of MPEG-7 media description schemes, it should be examined how these constructs can be expressed within typing automata.

In the following, several MPEG-7 DDL constructs that face common usage within MPEG-7 media description schemes are examined. For each of these constructs, it is investigated whether they are already expressible by the basic typing automaton mechanism supporting simple and complex content conditions only. If not, appropriate extensions are sketched. Thereby it is shown that typing automata constitute an intermediary representation of schema definitions that is flexible enough to be extended up to the expressiveness of MPEG-7 DDL.

This remainder of this section starts out by examining the representation of any, repeated, and empty content declarations within typing automata (7.5.1). It then investigates the representation of mixed content declarations (7.5.2) as well as complex type declarations (7.5.3). Finally, it is explored how attribute declarations can be covered within a typing automaton (7.5.4).

7.5.1 Any, repeated, and empty content declarations

With complex content conditions, typing automata provide a very flexible means for restricting valid element contents which is already capable of expressing quite a few additional constructs offered by MPEG-7 DDL. This is due to the fact that complex content conditions make use of expressive Perl 5 string regular expressions to determine permissible sequences of applicable element type IDs for an element's child elements.

Employing such Perl 5 string regular expressions, complex content conditions are well-suited, for example, to represent occurrences of the <any> construct within MPEG-7 DDL schema definitions. For a given element type, <any> specifies that arbitrary elements are eligible to appear within the elements of that type. This can be easily expressed by a transition rule that employs a complex content condition with the string regular expression `((et::.*|uet::null)::.*::.*)*`[3] which matches any sequence of known and unknown element type IDs. In order to understand that regular expression, it should be mentioned that within Perl 5 string regular expressions `.` matches any character (except linebreaks) and hence `.*` matches an arbitrary sequence of characters.

Since the namespace of an element type is an integral part of its ID, complex content conditions are also suited to model occurrences of <any> that further restrict the content model of an element type to a certain namespace. For instance, a transition rule with a complex content condition containing the string regular expression `((et::.*|uet::null)::http://www.example.org::.*)*` can be used to limit the contents of the elements of a given type to elements with types that originate from the namespace `http://www.example.org`.

[3]For the sake of clarity, any quoting backslashes are omitted within the Perl 5 string regular expressions to come that would normally be necessary in order to distinguish character data from reserved characters.

Furthermore, Perl 5 string regular expressions enable a painless mapping of repeated content declarations to complex content conditions. Not only optional and arbitrarily repeatable content can be modeled using the standard regular expression operators ? and *. Also, explicitly declared minimum and maximum occurrences of repeatable content can be directly expressed within Perl 5 string regular expressions using curly brackets. For example, a transition rule with a complex content condition containing the string regular expression `(et::MelodyType::http://...::Meter){1,5}` allows the elements of a certain type to consist of one up to five elements of type `Meter` with the scope `MelodyType`.

Finally, curly brackets also permit complex content conditions to enforce empty content. The regular expression `.{0,0}` matches empty strings only. As Definitions 15 and 16 assure, the signature of a sequence of applicable child element types for a given element – on which complex content conditions evaluate their string regular expressions during the validation phase – is an empty string if and only if the element has empty content.

7.5.2 Mixed content declarations

MPEG-7 DDL allows the declaration of mixed content. Permitting mixed content means that it is valid to intersperse arbitrary text fragments between an elements' child nodes. TDOM represents such text fragments by the means of text nodes, i.e., instances of the class `Text`. When examining the support of typing automata for mixed content, one finds that text nodes have not played any role so far for document validation and typing. When determining the applicable element types of an element and when bringing it to a corresponding typed representation, complex content conditions restrict themselves to the element's child elements and their applicable element types completely ignoring any other kinds of document nodes. Given this situation, the question is not whether typing automata accept mixed content whenever it is allowed – they always do because text nodes are simply overlooked – but rather how typing automata can be brought to reject mixed content whenever it is not permitted.

One solution to do this is to introduce a dedicated unmixed content condition. An unmixed content condition is a secondary condition that is not self-contained like a simple or complex content condition. Instead, it augments another condition with additional checks and operations. During the validation phase, an unmixed content condition verifies that no text nodes are interspersed with the child elements of an element in addition to checking the augmented condition. During the typing phase, an unmixed content condition behaves exactly like the augmented condition because the notion of typed representation applies to elements and attribute values only and not to text nodes and is thus independent of mixed content.

The basic idea of secondary conditions is shown in the class diagram of Figure 7.10. The diagram introduces an abstract class `SecondaryCondition` as a subclass of `Condition`. This class subsumes all secondary conditions, among others the class `UnmixedContentCondition` for unmixed content conditions. The association between `SecondaryCondition` and `Condition` ensures that a secondary condition always augments another condition, which may be a simple content condition or complex content condition but just as well another secondary condition. Thus, arbitrarily long chains of secondary conditions can be produced that ultimately refer to a simple content condition or complex content condition. In effect, this paves the way to the representation of very complex content models within the transition rules of a typing automaton.

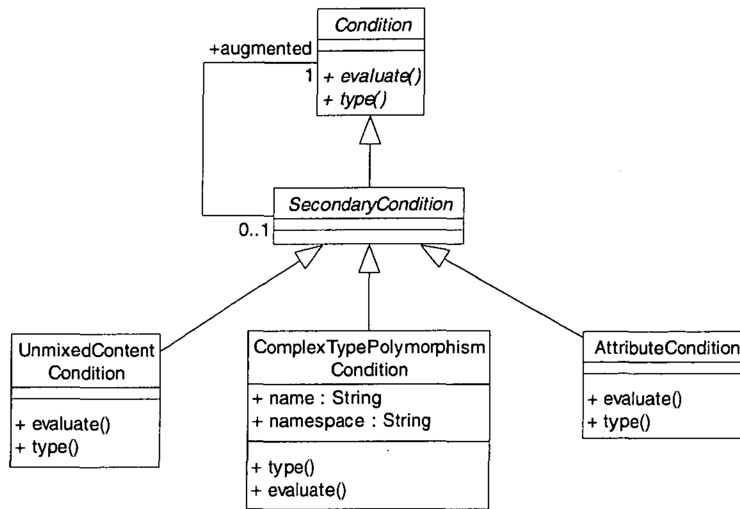


Figure 7.10: Secondary conditions (UML class diagram)

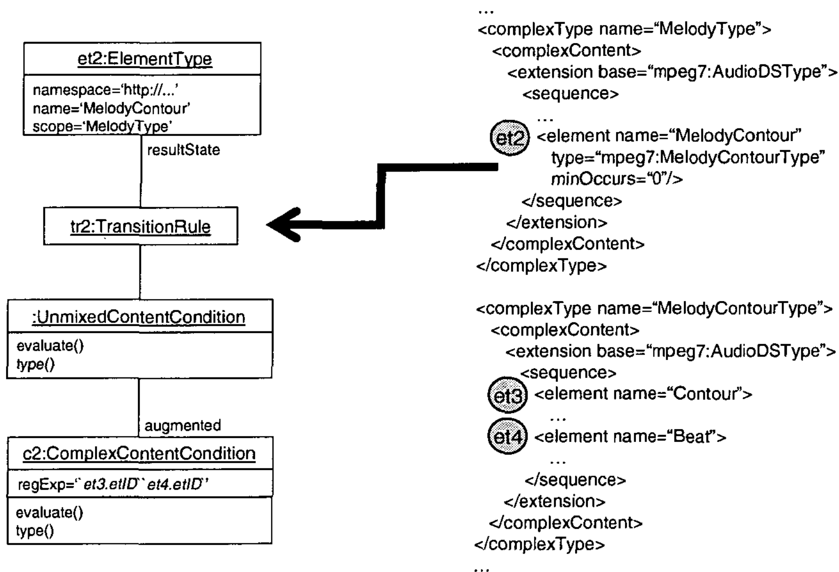


Figure 7.11: Example mapping of unmixed element content declaration (UML object diagram)

Figure 7.11 illustrates the use of unmixed content conditions for the representation of element type declarations contained in MPEG-7 DDL schema definitions whose content models do not permit mixed content. In that figure, the declaration of the element type `MelodyContour` contained in the complex type `MelodyType` within the `Melody` media description scheme is mapped to transition rule `tr2` just as before in Figure 7.6. But as the `MelodyContour` declaration does not permit mixed content, the transition rule's complex content condition `c2` is augmented

with a further unmixed content condition to obtain a more faithful mapping.

Definition 24 outlines a formal specification of the behaviour of `evaluate()` and `type()` methods for the class `UnmixedContentCondition` so that instances of this class can be used like in Figure 7.11 to prevent the occurrence of mixed content. According to the postcondition provided for the specification of `evaluate()`, the method first checks whether the passed element has text nodes among its child nodes. If it has, the unmixed content condition already evaluates to `false`. Otherwise, the result of the `evaluate()` method is identical to the result of the augmented condition's `evaluate()` method. As an unmixed content condition behaves like the augmented condition for the construction of typed representations, the pseudocode given for the `type()` method of `UnmixedContentCondition` simply delegates all its calls to the `type()` method of the augmented condition.

Definition 24 (Unmixed content condition)

```
context UnmixedContentCondition::evaluate(Element e,
      TransitionRule tr) :  Boolean
post: result = not(e.childNode -> exists(d | d.oclIsTypeOf(Text)) and
      self.augmented.evaluate(e, tr)

context UnmixedContentCondition::type(Element e, TransitionRule tr)
pre:  not(e.typed)
pre:  tr.resultState.name = e.etName
pre:  tr.resultState.namespace = e.etNamespace
pre:  self.evaluate(e, tr)
post: e.CUR(e@pre)
post: e.elementType = tr.resultState
post: e.allChildElements -> forAll(c |
        not(c.CUR(c@pre) and tr.typingAutomaton.
          applicableElementTypes(c) -> includes(c.elementType)) implies
          tr.typingAutomaton.applicableElementTypes(c) -> isEmpty() or
          not(c.parentNode.typed))
pseudocode:
    self.augmented.type(e, tr)
```

7.5.3 Complex type declarations

MPEG-7 DDL offers complex type declarations as a powerful and flexible construct for organizing the structure of schema definitions which, as the reader has already been able to observe by means of the sample `Melody` media description scheme, faces extensive use within MPEG-7 media description schemes. A complex type essentially constitutes a named complex content model which can be referenced within an element type declaration in order to define the contents valid for the element type. Given their relevance for MPEG-7, it is clearly of interest to examine how complex type declarations are represented within typing automata.

Inspection of the mapping scheme used for the translation of the `Melody` media description scheme of Figure 2.4 to the set of transition rules depicted by Figure 7.6 reveals that there is no one-to-one correspondence between complex type declarations and transition rules. Instead, a complex type declaration is implicitly covered by the complex content conditions of all those transition rules that represent element type declarations where the complex type is used to define the element type's content model. E.g, the complex type `MeterType` is not translated to a dedicated transition rule. But it is used to create the complex content condition `c1` of

the transition rule `tr1` which represents the declaration of the element type `Meter` because `Meter`'s content model is defined by means of `MeterType`.

This kind of mapping imposes no problems as long as complex types are not interrelated. However, MPEG-7 DDL allows to derive complex types from each other in order to organize them into a kind of specialization hierarchy. As it has been explained in detail in Chapter 2, MPEG-7 makes heavy use of this feature for the organization of its media description schemes. Based on complex type derivation hierarchies, MPEG-7 DDL permits polymorphism via the `xsi:type` attribute.

The problem with the mapping scheme used so far for translating MPEG-7 DDL schema definitions to typing automata is that it completely ignores complex type polymorphism. Because of this – and as an observant reader might have already noticed – we were forced to fall back on some black magic during the translation of the `Melody` media description scheme to the transition rules of Figure 7.6 so that the example media description of Figure 2.5 is valid with regard to the typing automaton. Although the content model of the element type `AudioDescriptionScheme` is defined by the complex type `AudioDSType`, we clairvoyantly knew that the example description would employ complex type polymorphism and fill the content of its root element according to the complex type `MelodyType`. Thus, constructed transition rule `tr7` has been constructed as if the the content model of `AudioDescriptionScheme` had been defined by `MelodyType` right from the beginning.

To obtain support for complex type polymorphism, an extended mapping scheme is suggested. In that scheme, every element type declaration is translated to a corresponding transition rule just as before. But whenever the content model within an element type declaration is defined by means of a complex type from which other complex types are derived, an additional transition is introduced for each of the directly or indirectly derived complex types. Such an additional transition bears the element type of the original element type declaration as its result state; its condition basically constitutes a complex content condition that represents the effective content model which is defined by the derived complex type. Effectively, a transition rule is created for every of the element type's content models that could be potentially instantiated by an element of that type via complex type polymorphism by means of an `xsi:type` attribute value.

It must be observed that this form of mapping might result in a proliferation of transition rules for a typing automaton that represents an MPEG-7 DDL schema definition with a deep complex type derivation hierarchy and element types whose content models are defined by complex types located in the upper parts of this hierarchy. But this is not so much a problem of typing automata and the extended mapping scheme. It is rather a tribute to the high expressiveness and considerable complexity inherent to the concepts of complex type derivation and complex type polymorphism any MPEG-7 DDL schema processor has to deal with. In order to allow reasonable handling of (a potentially large number of) alternative transition rules for one and the same element type declaration introduced by complex type derivation, typing automata should be given a means that helps them to quickly decide for one of these alternative rules during the validation phase when complex type polymorphism occurs inside a document.

Thus, the introduction of complex type polymorphism conditions as a further kind of secondary condition is suggested. Complex type polymorphism conditions are represented by the class `ComplexTypePolymorphismCondition` in the class diagram of Figure 7.10. A complex type polymorphism condition maintains the name and namespace of a given complex type as indicated by the attributes `name` and

`namespace`. During the validation phase, a complex type polymorphism verifies whether the element for which the condition is evaluated features an `xsi:type` attribute value addressing the name and namespace of the complex type maintained by the condition. Only if this relatively simple check has been successfully passed, the condition augmented by the complex content condition is also evaluated. Since complex type polymorphism conditions only serve to decide for one of the alternative transition rules representing a single element type declaration more quickly but do not otherwise influence the creation of corresponding typed representations once an appropriate transition rule has been chosen, they exactly behave like the augmented conditions during the typing phase.

Complex type polymorphism conditions are applied in the extended mapping scheme in that way that the complex content condition of every transition rule which has been additionally introduced for an element type declaration due to complex type derivation is augmented by an appropriate complex type polymorphism condition addressing the name and namespace of the particular derived complex type.

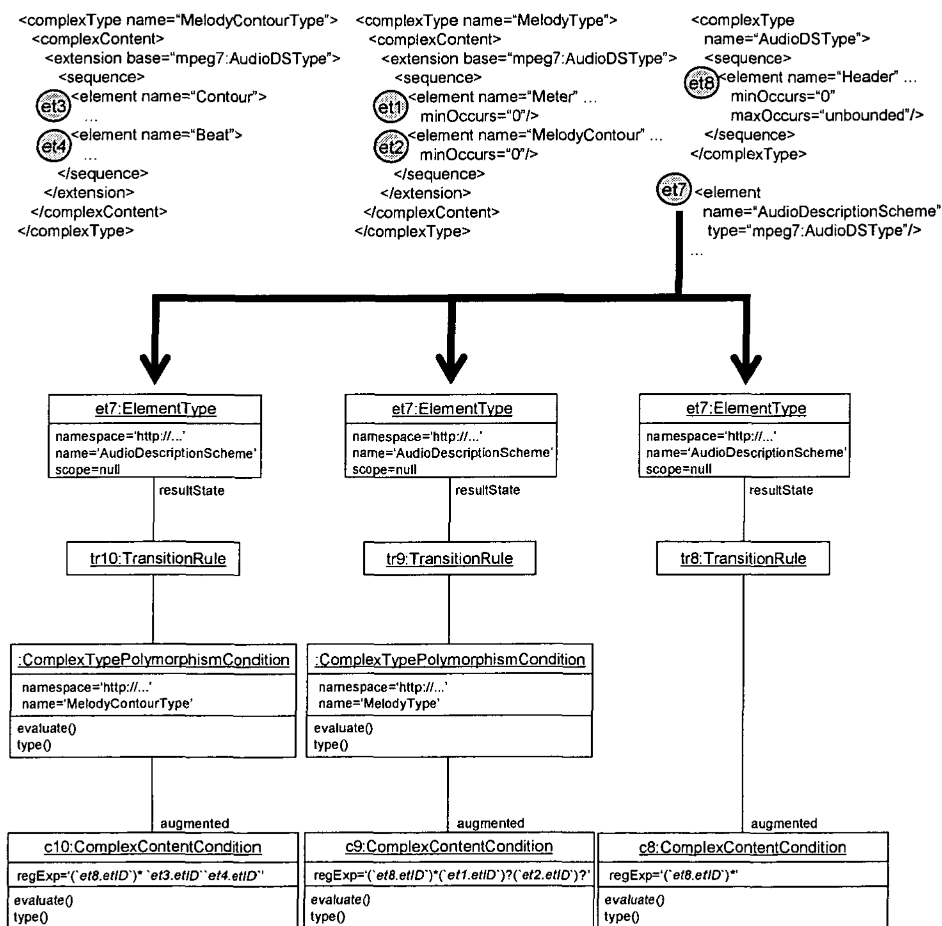


Figure 7.12: Example mapping of complex type derivation hierarchy (UML object diagram)

Figure 7.12 provides an example that illustrates the extended mapping scheme

and the application of complex type polymorphism conditions. The top of the figure shows an excerpt of the `Melody` media description scheme consisting of the complex types `MelodyContourType` and `MelodyType` as well as the element type `AudioDescriptionScheme` as already known from Figure 2.4. In addition, the declaration of the complex type `AudioDSType` is provided that is used to define the content model of `AudioDescriptionScheme`. `AudioDSType` specifies a content model consisting of an arbitrarily repeatable sequence of elements of type `Header`.

The bottom of the figure depicts the mapping of the `AudioDescriptionScheme` element type declaration to the transition rules of a typing automaton according to the suggested extended mapping scheme. First of all, the element type declaration is translated to the transition rule `tr8` as usual. `tr8` employs the complex condition `c8` to restrict the allowable contents of `AudioDescriptionScheme` elements to arbitrarily long sequences of `Header` elements as demanded by `AudioDSType`. In a second step, further transition rules with `AudioDescriptionScheme` as their result state are created for each complex type derived from `AudioDSType`. In this example, these are the complex types `MelodyType` and `MelodyContourType` which lead to the transition rules `tr9` and `tr10`. The conditions of `tr9` and `tr10` are made up of complex type polymorphism conditions which reference the names and namespaces of the respective complex types. Both conditions further augment complex content conditions that model the effective content models defined by the complex types. I.e., condition `c9` restricts the permitted content of `AudioDescriptionScheme` elements to arbitrarily long sequences of `Header` elements followed by optional `Meter` and `MelodyContour` elements as defined by the complex type `MelodyType`; condition `c10` restricts the permitted content to arbitrarily long sequences of `Header` elements followed by `Contour` and `Beat` elements as defined by the complex type `MelodyContourType`.

Given these transition rules, the element type `AudioDescriptionScheme` is only applicable to an element inside an XML document, if (a) the element complies to transition rule `tr8` by satisfying the complex content condition `c8` or if (b) the element complies to transition rule `tr9` or `tr10` by satisfying complex content condition `c9` or `c10`, respectively, and further bears an `xsi:type` attribute value referring to the corresponding complex type `MelodyContourType` or `MelodyContourType`.

We conclude the treatment of the representation of complex types within typing automata by sketching a suitable specification of the `evaluate()` and `type()` methods for the class `ComplexTypePolymorphismCondition` within Definition 25. For the purpose of the definition, it is assumed that there exist the operators `name` and `namespace` for strings that allow to extract the name and the namespace out of a qualified reference to a complex type. The postcondition of `evaluate()` states that the method checks whether the element passed to the method possesses an attribute value with the attribute name `type` and the attribute namespace `http://www.w3.org/2001/XMLSchema-instance` – i.e., an `xsi:type` attribute value – whose content refers to the name and namespace of the complex type maintained by the current complex type polymorphism condition before evaluating the condition it augments. The implementation that is suggested by the pseudocode for the `type()` method simply delegates its call to the condition augmented by the complex type polymorphism condition.

Definition 25 (Complex type polymorphism condition)

```
context ComplexTypePolymorphismCondition::evaluate(Element e,
      TransitionRule tr) : Boolean
post: result = e.attributeValue -> exists(av |
```

```
      av.attName = "type" and
      av.attNamespace = "http://www.w3.org/2001/XMLSchema-instance" and
      av.content.name = self.name and
      av.content.namespace = self.namespace) and
     self.augmented.evaluate(e, tr)
```

```
context ComplexTypePolymorphismCondition::type(Element e,
     TransitionRule tr)
pre:  not(e.typed)
pre:  tr.resultState.name = e.etName
pre:  tr.resultState.namespace = e.etNamespace
pre:  self.evaluate(e, tr)
post: e.CUR(e@pre)
post: e.elementType = tr.resultState
post: e.allChildElements -> forAll(c |
        not(c.CUR(c@pre) and tr.typingAutomaton.
         applicableElementTypes(c) -> includes(c.elementType)) implies
         tr.typingAutomaton.applicableElementTypes(c) -> isEmpty() or
         not(c.parentNode.typed))
pseudocode:
    self.augmented.type(e, tr)
```

7.5.4 Attribute declarations

MPEG-7 DDL, just like any other XML schema definition language, allows to restrict for each element type the attribute values eligible to appear within the elements instantiating that element type in an XML document. For this purpose, content models specified by means of complex type declarations inside MPEG-7 DDL schema definitions can be enhanced with additional attribute declarations. The different kinds of attribute declarations supported by MPEG-7 DDL mainly comprise those already known from classic DTDs. The essential difference between attribute declarations in MPEG-7 DDL and DTDs is DDL's support for strong typing that allows to restrict the permissible domain of the values of an attribute to a simple type.

Regarding this support for strong typing and given the fact that MPEG-7 media description schemes considerably make use of that support to define attributes that carry non-textual data, it is no surprise that TDOM's notion of typed representation not only encompasses elements but also attribute values. In order to be considered an adequate intermediary representation of an MPEG-7 DDL schema definition, typing automata should consequently provide support for the representation of attribute declarations that facilitate the validation of attribute values occurring in a document and the production of corresponding typed representations of these values. So far, however, the specification of typing automata has intentionally ignored attribute values for simplicity.

In the following, an extension of typing automata for the representation of attribute declarations is outlined. Just like the other extensions suggested so far, this extension introduces a dedicated kind of secondary condition, namely attribute conditions which are represented by the class `AttributeCondition` in the diagram of Figure 7.10. An attribute condition basically consists of a collection of attribute declarations. If an attribute condition is used to augment a transition rule's condition, not only the augmented condition is evaluated during the validation phase whenever the transition rule is applied to an element; it is also verified whether the attribute values of the element conform to the attribute declarations collected by

the attribute condition. During the typing phase, the attribute condition behaves like the augmented condition but additionally transforms the attribute values of the element to corresponding typed representations in a way that suits the attribute declarations to which the attibute values comply.

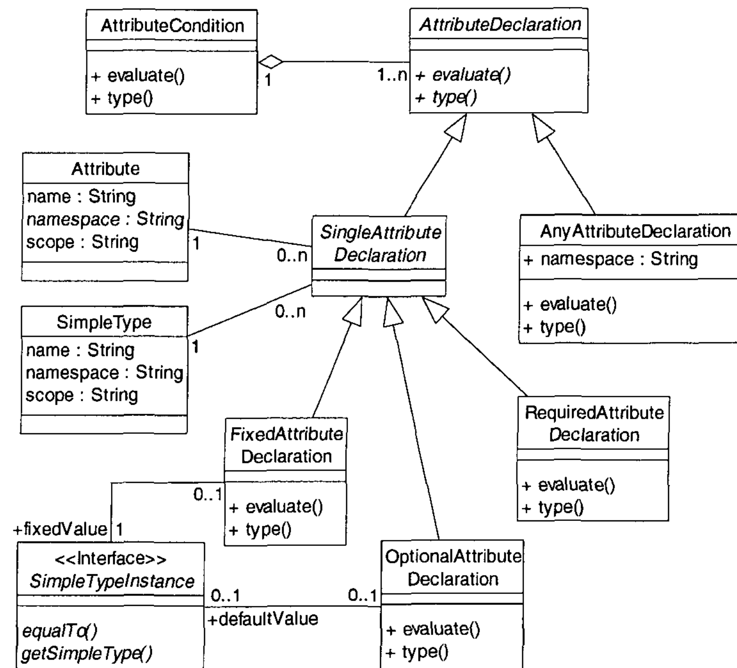


Figure 7.13: Attribute condition structure (UML class diagram)

The class diagram of Figure 7.13 provides further details on the structure suggested for attribute conditions and attribute declarations. According to the diagram, an attribute condition collects at least one attribute declaration all of which are subsumed by the abstract base class `AttributeDeclaration`. Attribute declarations are categorized into single attribute declarations and any attribute declarations as modeled by the subclasses `SingleAttributeDeclaration` and `AnyAttributeDeclaration`. The abstract notion of single attribute declarations subsumes basic attribute declarations that essentially permit a single value of the attribute (addressed by the association to the class `Attribute`) to appear within an element, if the content of the attribute value originates from the domain of the simple type (addressed by the association to the class `SimpleType`). In the diagram, the single attribute declarations considered are fixed, optional, required, and prohibited attribute declarations represented by corresponding subclasses. These represent the usual basic kinds of attribute declarations that are supported by most XML schema definition languages: required attribute declarations enforce the instantiation of a given attribute within an element. Optional attribute declarations allow the instantiation of a given attribute within an element but do not enforce it. A default value in form of a simple type instance can be specified for the case that an optional attribute is not instantiated. Fixed attribute declarations behave like optional attribute declarations but rigidly restrict the allowable contents of attribute

values to the simple type instance provided as the fixed value. Finally, prohibited attribute declarations forbid the instantiation of an attribute within an element. Support for further kinds of attribute declarations could be integrated into this structure by additional subclasses of `SingleAttributeDeclaration` if necessary.

An any attribute declaration, in contrast, allows the occurrence of arbitrary attribute values within an element whose attribute namespaces can be optionally limited to the namespace contained in `AnyAttributeDeclaration`'s `namespace` attribute. This facilitates the representation of the `<anyAttribute>` construct of MPEG-7 DDL within transition rules of a typing automaton.

There are a few restrictions concerning the structure of attribute conditions which are formally expressed by Constraint 11. The first restriction is that the names and namespaces of the attributes referred to by single attribute declarations must be unique in order to avoid conflicting declarations. I.e., there may be no two different single attribute conditions which refer to attributes that bear the same name and namespace. The second restriction is that if an optional attribute declaration refers to a default value, that default value must be a valid instance of the simple type addressed by the attribute declaration. The third restriction is quite similar: the fixed value of a fixed attribute declaration must be an instance of the simple type that is referenced by the declaration.

Constraint 11 (Restrictions on attribute conditions)

```
context AttributeCondition
inv: attributeDeclaration -> forAll(sad1, sad2 :
      SingleAttributeDeclaration |
      sad1.attribute.namespace = sad2.attribute.namespace and
      sad1.attribute.name = sad2.attribute.name implies
        sad1 = sad2)

context OptionalAttributeDeclaration
inv: defaultValue -> notEmpty implies
      defaultValue.getSimpleType() = simpleType

context FixedAttributeDeclaration
inv: fixedValue.getSimpleType() = simpleType
```

Figure 7.14 provides an example how attribute conditions can be used to represent attribute declarations occurring in an MPEG-7 media description scheme. The right part of the figure shows the declaration of the complex type `AudioDSType` which has been enhanced by an attribute declaration permitting the optional use of an attribute `id` of type `ID` and by a declaration on the basis of the `<anyAttribute>` construct further allowing the use of arbitrary attributes as long as they originate from the namespace `http://www.mpeg7.org/`.

The left part of the figure shows the representation of the declaration of the element type `AudioDescriptionScheme`, whose content model is defined via `AudioDSType`, by means of a transition rule. For the figure, complex type polymorphism is neglected for the sake of clarity. As one can observe, the attribute declaration is straightforwardly mapped to an attribute condition within the transition rule. The attribute condition augments a complex content condition that restricts allowable element contents to arbitrarily long sequences of `Header` elements in accordance to `AudioDSType`. The attribute condition models the declaration of the `id` attribute with an optional attribute declaration which references an `Attribute` object representing `id` and a `SimpleType` object modeling the simple

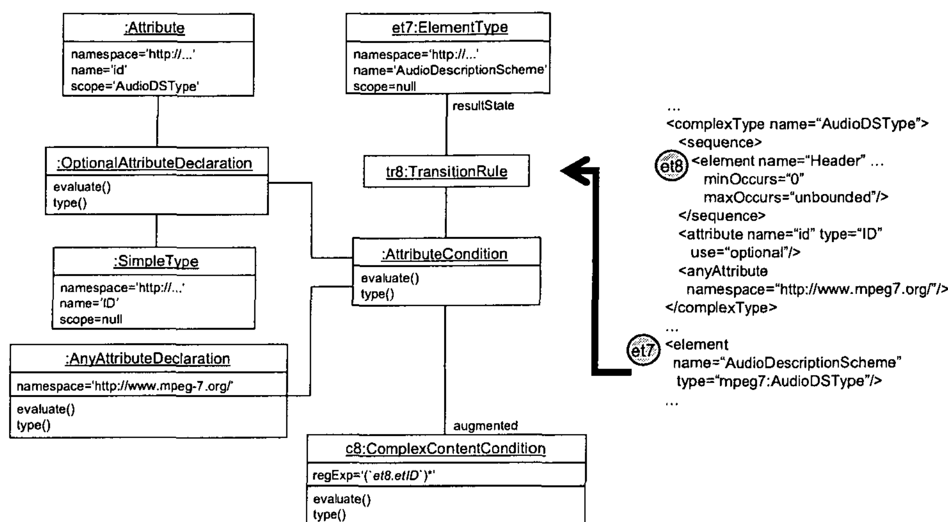


Figure 7.14: Example mapping of attribute declarations (UML object diagram)

type ID. The `<anyAttribute>` construct is mapped to an any attribute declaration whose `namespace` property is set to `http://www.mpeg7.org/`.

Definition 26 details the behaviour of attribute conditions during the validation and typing phases of a typing automaton by outlining a specification of the `evaluate()` and `type()` methods of the class `AttributeCondition`. Since an attribute condition essentially constitutes a container for various kinds of attribute declarations, the specification of both methods relies on a fixed set of functionality that has to be offered by an attribute declaration. As already indicated by the abstract methods of the abstract base class `AttributeDeclaration` in the class diagram of Figure 7.13, any concrete subclass of `AttributeDeclaration` that represents a particular kind of attribute declaration must provide appropriate implementations for the methods `evaluate()` and `type()` similar to a condition's methods of the same name: `evaluate()` expects an attribute value as its parameter and returns `true` if the attribute value constitutes a valid instantiation of the current attribute declaration; `type()` takes an attribute value in untyped representation as its parameter and transforms the attribute value to a corresponding typed representation in a manner that is appropriate for the current attribute declaration.

According to the postconditions that are given in Definition 26 for the `evaluate()` method of the class `AttributeCondition`, an attribute condition performs the following checks on an element during the validation phase of a typing automaton in addition to evaluating the augmented condition: it is first verified for each attribute value of the element whether there exists an attribute declaration within the attribute condition that is validly instantiated by the attribute value. It is further ensured that every required attribute declaration of the condition is satisfied by one of the element's attribute values. Finally, it is ascertained that no attribute value satisfies a prohibited attribute declaration that might be potentially contained within an attribute condition.

Provided that an element in untyped representation is passed to `AttributeCondition`'s `type()` method for which the current attribute condition evaluates to `true`, the pseudocode suggested for the implementation of that method

first brings the element and as much of its direct and indirect child elements as possible to corresponding typed representations by delegating its call to the augmented condition. Then, `type()` tries to transform each of the element's attribute values in untyped representation to a corresponding typed representation using the `type()` method of an attribute declaration which the attribute value validly instantiates. There is always at least one such attribute declaration since this has already been ensured by `evaluate()`. Note that there is one case where the construction of a corresponding typed representation of an attribute value might not be possible which is expressed by the last of `type()`'s postconditions: in case that the attribute value instantiates an any attribute declaration only. Any attribute declarations, however, lack important type information, i.e. the instantiated attribute and the simple type forming the domain of its values, that is necessary for the construction of typed representations.

Definition 26 (Attribute condition)

```
context AttributeCondition::evaluate(Element e,
      TransitionRule tr) :  Boolean
post: result = e.attributeValue -> forAll(av |
        self.attributeDeclaration -> exists(ad | ad.evaluate(av))) and
      self.attributeDeclaration -> forAll(rad :
      RequiredAttributeDeclaration |
        e.attributeValue -> exists(av | rad.evaluate(av))) and
      self.attributeDeclaration -> forAll(pad :
      ProhibitedAttributeDeclaration |
        not(e.attributeValue -> exists(av | pad.evaluate(av)))) and
      self.augmented.evaluate(e, tr)

context AttributeCondition::type(Element e, TransitionRule tr)
pre:  not(e.typed)
pre:  tr.resultState.name = e.etName
pre:  tr.resultState.namespace = e.etNamespace
pre:  self.evaluate(e, tr)
post: e.CUR(e@pre)
post: e.elementType = tr.resultState
post: e.allChildElements -> forAll(c |
        not(c.CUR(c@pre) and tr.typingAutomaton.
          applicableElementTypes(c) -> includes(c.elementType)) implies
          tr.typingAutomaton.applicableElementTypes(c) -> isEmpty() or
          not(c.parentNode.typed))
post: e.attributeValue -> forAll(av |
        not(av.CUR(av@pre)) implies
          self.attributeDeclaration -> exists(sad :
            AnyAttributeDeclaration | sad.evaluate(av)) and
          self.attributeDeclaration -> size() = 1)
pseudocode:
    -- Invoke behaviour of augmented condition
    self.augmented.type(e, tr)
    -- Type all attribute values in untyped representation which
    -- validly instantiate a single attribute declaration
    foreach av in e.attributeValue -> select(untyped) do
      iad := self.attributeDeclaration -> any(ad | ad.evaluate(av))
      iad.type(av)
    endforeach
```

We conclude the treatment of attribute conditions by showing how implementations of the `evaluate()` and `type()` methods of attribute declarations could look like. For brevity, we restrict ourselves to optional attribute declarations and any attribute declarations that are used for the example of Figure 7.14. The implementations of both methods for optional attribute declarations are covered by Definition 27. `evaluate()` delivers `true` for an attribute value if (a) the attribute name and namespace of the attribute value match the name and namespace of the attribute addressed by the current optional attribute declaration and if (b) an instance of the simple type addressed by the optional attribute declaration can be constructed from the textual representation of the attribute value's content. `type()` uses the attribute and simple type associated with the optional attribute declaration to straightforwardly the attribute value that is passed passed to the method to typed representation.

Definition 27 (Optional attribute condition)

```
context OptionalAttributeCondition::evaluate(AttributeValue av) :
      Boolean
post: result = (av.attName = self.attribute.name) and
      (av.attNamespace = self.attribute.namespace) and
      (av.attNamespace = self.attribute.namespace) and
      (av.typedContent -> notEmpty() implies
        self.simpleType.simpleTypeInstanceFactory.
          fromString(av.typedContent.getSimpleType().
            toString(av.typedContent)) <> null) and
      (av.typedContent -> isEmpty() implies
        self.simpleType.simpleTypeInstanceFactory.
          fromString(av.content) <> null)

context OptionalAttributeCondition::type(AttributeValue av)
pre:  not(av.typed)
pre:  self.evaluate(av)
post: av.CUR(av@pre)
pseudocode:
    -- Bring the attribute value to an appropriate typed
    -- representation.
    av.typed := true
    av.name := null
    av.namespace := null
    av.attribute := av.attribute -> including(self.attribute)
    -- Use the simple type instance factory associated
    -- with the simple type of the optional attribute condition
    -- to produce an appropriate simple type instance for use as
    -- typed attribute value content
    stif := self.simpleType.simpleTypeInstanceFactory
    sti := stif.fromString(av.simpleContent)
    -- Set simple type instance as simple content of element
    av.simpleContent := null
    av.typedSimpleContent := av.typedSimpleContent ->
      including(sti)
```

Definition 28 treats the `evaluate()` and `type()` methods of any attribute declarations. `evaluate()` returns `true` if the attribute namespace of the attribute value matches the namespace referred to by the current any attribute declaration. Because an any attribute declaration does not carry sufficient information for the

construction of a corresponding typed representation of the attribute value that is passed to `type()`, the method attempts to delegate its call to a single attribute declaration that is also validly instantiated by the attribute value. Should no such single attribute declaration exists, the attribute value remains in untyped representation.[4]

Definition 28 (Any attribute condition)

```
context AnyAttributeCondition::evaluate(AttributeValue av) : Boolean
post: result = self.namespace <> null implies
        av.attNamespace = self.namespace

context AnyAttributeCondition::type(AttributeValue av)
pre:  not(av.typed)
pre:  self.evaluate(av)
post: self.attributeCondition -> exists(ad :
        not(ad.oclIsTypeOf(AnyAttributeCondition)) and
        ad.evaluate(av)) implies
        av.CUR(av@pre)
pseudocode:
    -- Check whether attribute value also instantiates a
    -- single attribute declaration.
    if self.attributeCondition.attributeDeclaration -> exists(sad :
        SingleAttributeDeclaration | sad.evaluate(av)) then
      -- If so, use single attribute declaration to produce
      -- a corresponding typed representation of the attribute
      -- value, because this can't be done on the basis of an
      -- any attribute declaration.
        isad := self.attributeCondition.attributeDeclaration ->
          any(sad : SingleAttributeDeclaration | sad.evaluate(av))
        isad.type(av)
    endif
```

[4]Note that in this case, the element to which the attribute value belongs still remains in typed representation.

Chapter 8

Implementation

With TDOM and typing automata, we have laid important foundations for an XML database solution that suffices the requirements of Chapter 3 and is thus suitable for the management of MPEG-7 media descriptions. In this chapter, we describe the implementation of a Java-based prototype of such a database solution, the Persistent Typed Document Object Model (PTDOM), whose architecture is shown in Figure 8.1.

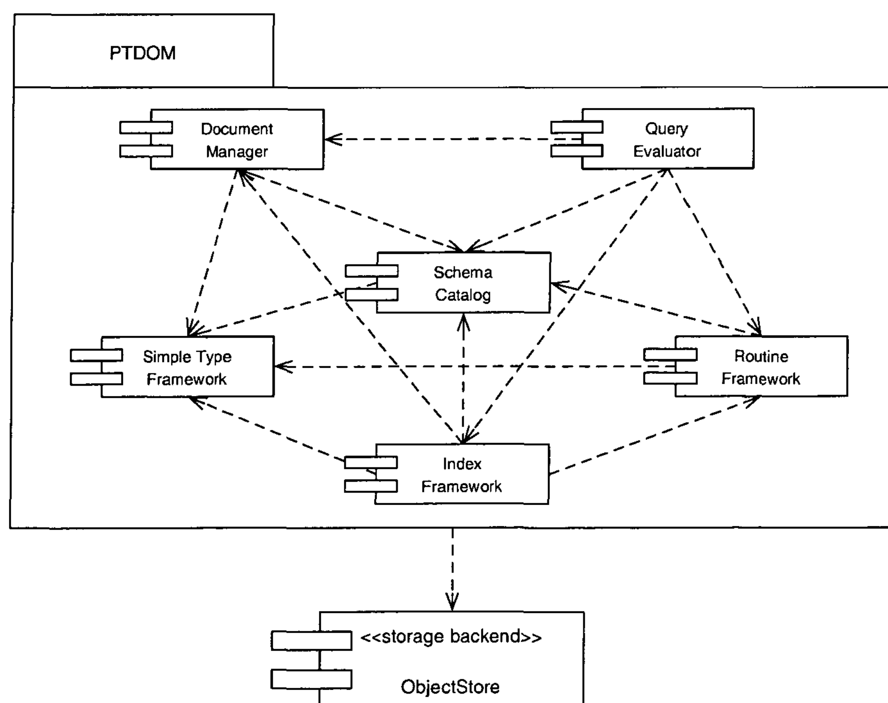


Figure 8.1: PTDOM architecture (UML component diagram)

The components of the depicted architecture address the requirements for the management of MPEG-7 media descriptions in a number of ways:

- *Representation of media descriptions:* The document manager employs

TDOM to *fine-grainedly store and represent* all the XML documents that are managed with PTDOM. In combination with the simple type framework, the document manager also provides a *typed representation* of basic document contents.

- *Access to media descriptions:* By means of TDOM, the document manager already gives applications *fine-grained and typed navigational access* to XML documents. By TDOM's support for both typed and untyped representations, the document manager also permits flexible, *fine-grained updates.* A further way to fine-grained document access is given by the query evaluator component which realizes an XPath query processor.

  Value index structures are brought into PTDOM by the index framework that supplies hash tables, B-Trees, and even multidimensional R-Trees; support for *text index structures* could be integrated with that framework as well.

  Finally, *path indexing* support within PTDOM is realized via TDOM's concept of typed representations: since elements and attribute values in typed representation are tightly coupled to the element types and attributes in the schema definitions they instantiate allowing to immediately obtain all instances of a given element type or attribute, these schema definitions can serve as path indexes comparable to DataGuides [GW97] of the Lore system.

- *Media description schemes:* For the management of the schema definitions to which the documents in the document manager are valid, PTDOM features an *MPEG-7 DDL-compliant schema catalog.* The catalog is able to produce equivalent typing automata for its schema definitions which are employed not only for *validation* but also for the *inference of typed representations.* Although not yet realizing a full-fledged query optimizer, the query evaluator also utilizes the schema definitions within the catalog for an *optimized evaluation* of XPath expressions.

- *Extensibility:* The index framework not only supplies a rich set of ready-to-use value index structures. It also permits to flexibly integrate new unordered, ordered, and multidimensional index structures with PTDOM making the system *extensible with index structures.* The routine framework further allows the integration of arbitrary user-defined routines which may be used for the querying of XML documents, making PTDOM *extensible with functionality.* With the simple type framework, it is even possible to integrate support for new simple types and simple type derivation methods.

- *Classic DBMS functionality:* For classic DBMS functionality, PTDOM relies on the object-oriented DBMS ObjectStore [eXc00] which it employs as its storage backend. Thereby, PTDOM inherits ObjectStore's mature *transaction support* and *backup and recovery* functionality, as well as the system's *fine-grained, page-based concurrency control.* PTDOM even gains flexibility as there exists the PSEPro [eXc01b] small-scale in-process variant of ObjectStore making PTDOM configurable as both a server-based as well as an in-process database solution. In future, we plan to replace ObjectStore with a dedicated XML storage manager, like the ones proposed by [KM99, FHK+02] or [HMF99].[1]

[1]Concerning potential objections with regard to the "nativity" of PTDOM when realized on top of ObjectStore, one should bear in mind that, according to Chapter 4, we regard PTDOM

The following sections present the individual components of PTDOM in more detail (8.1 – 8.6). This chapter concludes with some experimental results which indicate the overall viability of the PTDOM approach and the prospects it offers (8.7).

8.1 Simple Type Framework

The simple type framework component of the PTDOM prototype provides an implementation of TDOM's simple type framework which has already been introduced in Chapter 6. With this component, it is thus possible to enhance PTDOM with support for arbitrary simple types, elementary as well as derived.

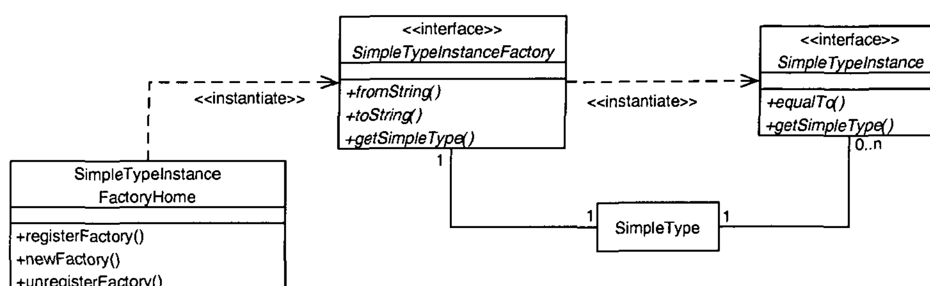


Figure 8.2: Simple type framework component (UML class diagram)

The class diagram of Figure 8.2 presents the overall structure of the simple type framework component. As one can see, the component constitutes a direct implementation of TDOM's simple type framework. The entry point to the component is given by the simple type instance factory home, represented by the class of the same name in the diagram. It serves as a central registry for all simple type instance factories in PTDOM. To make support for a certain simple type available, the class providing the factory for the instances of that type needs to be registered with the simple type instance factory home under the type's name and namespace. Via the method `newFactory()` which is passed the name and namespace of the simple type of which a factory is desired, the simple type instance factory home is capable of dynamically instantiating a registered factory class.

Applying the simple type framework in this manner, we have integrated support for all the simple type derivation methods as well as many of the elementary simple types offered by MPEG-7 DDL into PTDOM.

8.2 Document Manager

The document manager serves as a central registry responsible for managing all XML documents stored with PTDOM. The document manager is capable of importing XML documents into PTDOM as well as exporting them back into an

as a native XML database solution as long as it completely encapsulates the ObjectStore storage backend. In a similar manner, the commercial XML database solution eXcelon XIS that also founds on ObjectStore is commonly regarded as native. One should furthermore consider that ObjectStore is an object-oriented DBMS following a rather low-level page server architecture not very different from storage managers like Shore [CDF+94] which constitutes the storage backend of the native research prototype Timber [JAC+02].

external format. Founding on TDOM and the simple type instance mechanism supplied by the simple type framework, it provides applications with fine-grained and appropriately typed representations of the documents stored with PTDOM giving them fine-grained and type-adequate navigational access to their contents as well as allowing their fine-grained manipulation.

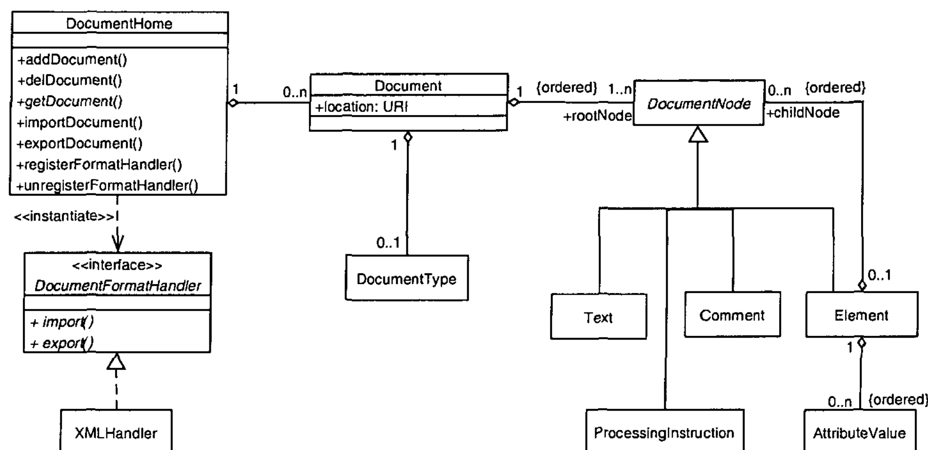


Figure 8.3: Document manager overview (UML class diagram)

The class diagram of Figure 8.3 gives an overview of the document manager component. The entry point to the document manager is formed by the so-called document home which is represented by a corresponding class. The document home acts as a container of all documents stored with PTDOM. As indicated in the right part of the diagram, the document manager makes use of a straightforward implementation of TDOM as introduced in Chapter 6 for the representation of the documents in the document home.

For the import and export of XML documents to and from the document manager, handlers for document formats, i.e., classes that implement the `DocumentFormatHandler` interface, can be registered with the document home, in principle allowing to integrate support for other storage formats than the traditional text format defined by the XML standard [BPSMM00] such as the MPEG-7 binary exchange format BiM [ISO01a] with PTDOM increasing the flexibility of the system. Whenever the methods `importDocument()` or `exportDocument()` of the document home are called in order to import or export an XML document, the schema home looks up and dynamically instantiates the particular handler which has been registered for the desired import or export format and defers the task to the corresponding methods of that handler.

8.3 Schema Catalog

A central component of PTDOM is its schema catalog. It manages the various schema definitions to which the XML documents stored with PTDOM comply. The schema catalog is able to import and export arbitrary schema definitions written in MPEG-7 DDL (support for other schema definition languages can be integrated as well), ensures the integrity of these definitions, keeps a fine-grained and accurate representation of the schema information they carry, and maintains equivalent

typing automata.

The rich information contained in the schema catalog is exploited throughout PTDOM in a number of ways. The typing automata are used for validation of the documents contained in the document manager and for the construction of typed representations of their elements and attribute values. As typed representations of elements and attribute values in TDOM are tightly interlinked with the element types and attribute values they instantiate, the schema catalog also serves as a path index allowing to quickly obtain all elements and attribute values for a given element type or attribute in a schema definition. Finally, the detailed representation of schema information within the catalog can also serve a rich decision base for sophisticated query optimization.

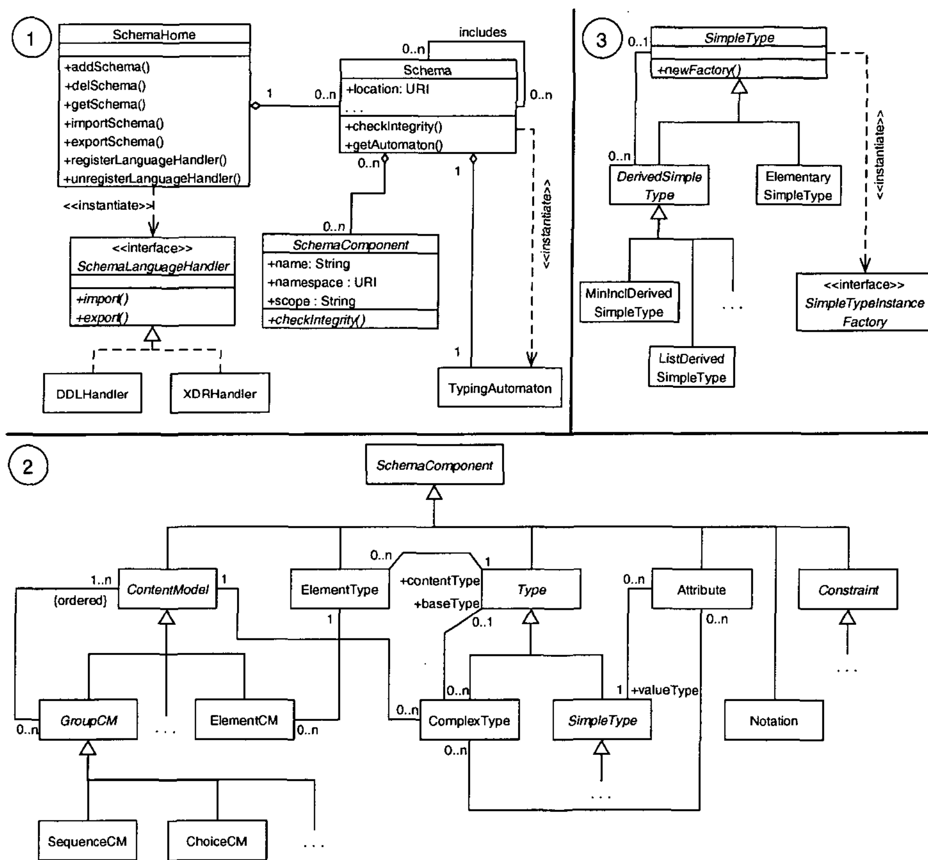


Figure 8.4: Schema catalog overview (UML class diagrams)

The class diagram to the upper left (1) of Figure 8.4 gives an overview of the basic structure of PTDOM's schema catalog which is very similar to the organization the document manager. Just as the document home acting as a container for all XML documents stored with PTDOM forms the entry point to the document manager, a schema home that serves as a container for all the schema definitions stored with PTDOM forms the entry point to the schema catalog; just as document format handlers can be registered with the document home to integrate support for new storage formats for the import and export of XML documents, schema language handlers can be registered with the document home to integrate support for

other schema definition languages than MPEG-7 DDL for the import and export of schema definitions. Within our prototypical implementation, we have also realized an XDR [FT98] handler.

Schema definitions (represented by the class of the same name in the diagram) are identified by the storage location from which they have been imported and are organizable in an inclusion lattice permitting a flexible modularization of schema definitions. Schema definitions are regarded as containers of the schema components which are declared inside these definitions, such as simple types, element types, and attributes. The schema catalog subsumes all the different kinds of schema components under one common abstract base class named `SchemaComponent`. This base class defines that any schema component can bear a name and namespace (potentially scoped) but leaves the representational details of the different kinds of schema components to subclasses provided for them.

In addition to that, `SchemaComponent` demands that its subclasses supply appropriate implementations of the abstract method `checkIntegrity()` which validate the consistency and integrity constraints associated with the respective kinds of schema components represented by these subclasses. For example, the implementation of this method in that particular subclass which represents attributes should ensure that a potentially declared default value matches an attribute's value type. In that manner, PTDOM's schema catalog is enabled to ensure the consistency and integrity of the schema definitions it manages by successively calling `checkIntegrity()` on each of a schema definition's schema components as it is done in the `checkIntegrity()` method of the `Schema` class.

For the purpose of validating and typing the XML documents kept by the document manager, the catalog's schema definitions are furthermore able to compile themselves to equivalent typing automata via the method `getAutomaton()`. The schema catalog provides a full implementation of the typing automaton mechanism as it has been described in Chapter 7 making use of local document typing. To speed up successive calls to `getAutomaton()`, schema definitions actually cache the compiled typing automata and keep them consistent with the schema components they contain.

Up to this point, schema definitions have been regarded as mere collections of abstract schema components. In order to permit the utilization of the schema catalog as a rich source of schema information for further exploitation for the management of XML documents, a detailed model for schema definitions must be provided that concretely defines the different kinds of schema components available as well as their structure and interrelationships. This model should be expressive enough to allow the representation of arbitrary media description schemes written in MPEG-7 DDL.

For this reason, it has been decided to closely orientate the model for schema definitions used within the schema catalog of PTDOM along the XML Schema Component Data Model [TBM+01] which comes with the XML Schema standard and is able to capture the contents of XML Schema definitions in detail. Slightly enhanced to cover the DDL-specific extensions, the model is expressive enough to faithfully reproduce the different constituents of any MPEG-7 media description scheme. Since comparisons of existing schema definition languages for XML documents show that XML Schema (and thus MPEG-7 DDL as well) exceeds the expressiveness of most other XML schema definition languages [LC00], PTDOM and its schema catalog can also be expected to be of use in application domains in which other schema definition languages play dominant roles – as long as only appropriate handlers for these languages are supplied.

The class diagram to the bottom (2) of Figure 8.4 gives a coarse overview of the model summarizing the different kinds of schema components essentially distinguished and highlighting major relationsships between them. The model differentiates – apart from notations and constraints such as uniqueness and key constraints – types, both simple and complex, element types, attributes, and content models. As the associations in the diagram indicate, the model also faithfully reproduces the relationships between these kinds of schema components, for instance, that the content of element types and attributes is defined using complex types and simple types, that complex types can be derived from other simple and complex types, etc.

By the use of typing automata for document validation and typing via typing automata as well as the connection between element types and attributes with the elements and attribute values in typed representation, the schema catalog is closely coupled to the document manager. The schema catalog is further coupled to the simple type framework, as illustrated by the class diagram to the upper right (3) of Figure 8.4. The diagram provides more insight into how simple type declarations are represented within the catalog's model for schema definitions by showing an excerpt of the subclass hierarchy below the base class `SimpleType` which had been previously abbreviated by dots in Figure 8.4 (2). As one can see, the model basically distinguishes elementary simple types coming with a schema definition language – their occurrences in schema definitions are modeled by the class `ElementarySimpleType` – and simple types that are derived from other simple types within a schema definition – subsumed under the abstract base class `DerivedSimpleType`. For every simple type derivation method supported by XML Schema and MPEG-7 DDL, the model provides a subclass of `DerivedSimpleType` to represent simple types that have been derived with that method.

For a comfortable construction of instances of simple types represented in this fashion, `SimpleType` requires its subclasses to implement the abstract method `newFactory()` such that it delivers a suitable simple type instance factory of the simple type framework for a given simple type. The implementation of this method within the class `ElementarySimpleType` simply looks up and returns that factory which has been registered with the simple type instance home for the instances of the given predefined simple type. The implementations of `newFactory()` within the subclasses of `DerivedSimpleType` likewise look up and deliver those factories which have been registered with the simple type instance home for the instances of simple types that have been derived via the methods corresponding to the respective subclasses. They additionally look up, however, the factories for their respective base types and chain them together appropriately: as already explained before in Chapter 6, the construction of instances of derived simple types usually depends on the construction of the instances of their base types.

8.4 Routine Framework

The routine framework allows the integration of user-defined routines with PTDOM which are similar in concept to classic stored procedures and functions of relational DBMSs. Thereby, it is possible to extend the system with arbitrary functionality for the reasonable querying and processing of XML documents stored with PTDOM.

The class diagram of Figure 8.5 gives an overview of the routine framework. The routine home (modeled by a corresponding class in the diagram) keeps track of all routines existing within PTDOM. As expressed by the class `Routine`, each of these routines is characterized by its name and signature: its allowable input parameters

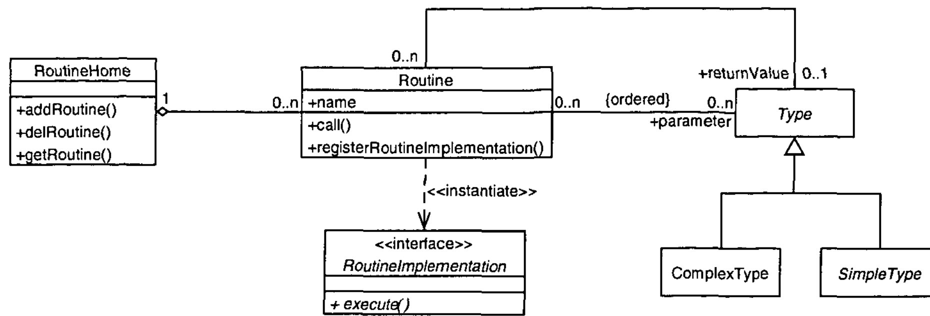


Figure 8.5: Routine framework overview (UML class diagram)

described by a sequence of simple or complex types taken from a schema definition within the schema catalog and its return value described by a further type in case that the routine constitutes a function. If an input parameter or the return value is specified via a simple type, the routine expects to be passed as this parameter or delivers as its return value an instance of that simple type (or one of its derived types); if a parameter or the return value is specified by means of a complex type, the routine expects to be passed as this parameter or delivers as its return value an element in typed representation whose content is filled according to that complex type (or one of its derived types).

The behavior of a routine is provided by a dedicated implementation class which is registered with the routine. The routine framework demands the implementation class to realize the `RoutineImplemenation` interface enforcing the existence of the method `execute()`. This method takes an array of objects as its input parameters, performs the functionality expected from a given routine, and delivers the routine's return value as an object. A routine is invoked via the method `call()` of the class `Routine` which like `execute()` is passed an array of objects as the input parameters to the routine. After typechecking these parameters against the routine's signature, `call()` dynamically instantiates an object of the registered implementation class and delegates processing to this object's `execute()` method. The result delivered by `execute()` is again typechecked and given back as the the routine call's return value.

The separation of routines from the implementations of their behavior has the advantage that similar routines can share implementations. It would be indeed tedious if dozens of very similar implementation classes had to be provided to integrate `equals()` functions into PTDOM, each of which compares two instances of a given simple type for equality. Instead, it can be exploited that the `SimpleTypeInstance` interface of the simple type framework already ensures that every instance of a simple type is able to compare itself against another via the method `equalTo()`. Therefore, a single generic implementation class on the basis of `equalTo()` can be provided that can then be shared among all `equals()` functions. One still has the option to replace this generic implementation class with a more specific one should this be desirable for a certain `equals()` function.

Given this overall organization of the routine framework, the integration of a new routine into PTDOM thus requires the provision of an appropriate implementation class that realizes the routine's functionality, the creation of an appropriate `Routine` object that captures the name as well as the signature of the routine, the registration of the implementation class with that object, and the registration of the

Routine object with the routine home applying the method addRoutine(). In this manner, a comprehensive set of routines, e.g., the XQuery and XPath functions and operators [MMRW02], can be systematically integrated into PTDOM. A subset of these routines has already been realized.

To retrieve routines registered with the routine home, the class RoutineHome offers the method getRoutine() which is passed the name of the routine and the signature desired, i.e., the desired types of the input parameters and the desired type of the return value. If a routine with that name and signature has been registered, this routine is returned. If not, all compatible routines are looked up. Obeying the contravariance rule, a routine is considered compatible if it has the desired name, the types of its input parameters either match the types desired for the input parameters or are base types of these types, and the type of its return value either matches the type desired for the return value or is derived from this type. Out of these compatible routines, getRoutine() similar to CLOS [Ste90] returns the most specific one.

8.5 Index Framework

The index framework makes value index structures available with PTDOM. It not only provides a rich set of such value index structures including hash tables, B-Trees, and R-Trees; it also facilitates the seamless integration of arbitrary further value index structures – unordered, ordered, as well as spatial ones – into PTDOM. The index framework is thus comparable in function to interfaces such as the Oracle Extensible Indexing API [GD02] that allow the integration of new value index structures into object-relational DBMSs.

The class diagram of Figure 8.6 gives more insight into the index framework which closely collaborates with the schema catalog and document manager. Within the framework, value index structures are represented by the interface IndexStructure. A value index structure is attached to either an element type or an attribute within one the schema catalog's schema definitions. The framework gathers these indexable schema components under the common interface IndexableSchemaComponent. The value index structure then indexes all the document nodes along their content which validly instantiate the indexable schema component it is attached to within the documents maintained by the document manager, namely, the attribute values or elements in typed representation based on the schema component. The framework subsumes these indexed document nodes under the interface IndexedDocumentNode.

As indicated by the methods offered by IndexStructure and IndexableSchemaComponent and the bidirectional association between both interfaces, each value index structure not only knows the indexable schema component it is attached to; also, each indexable schema component in the schema catalog vice-versa keeps track of all the value indexes attached to it. This is exploited by PTDOM for the maintenance of index consistency: whenever elements or attribute values in a document are changed in a way that affects value index structures associated with element types or attributes in the schema catalog, the affected index structures are updated by removing and/or inserting the elements or attribute values in question via the removeNode() and insertNode() methods that are provided by every value index structure. For instance, when an element in untyped representation is brought to typed representation during document import, the value index structures attached to the element type on which the newly

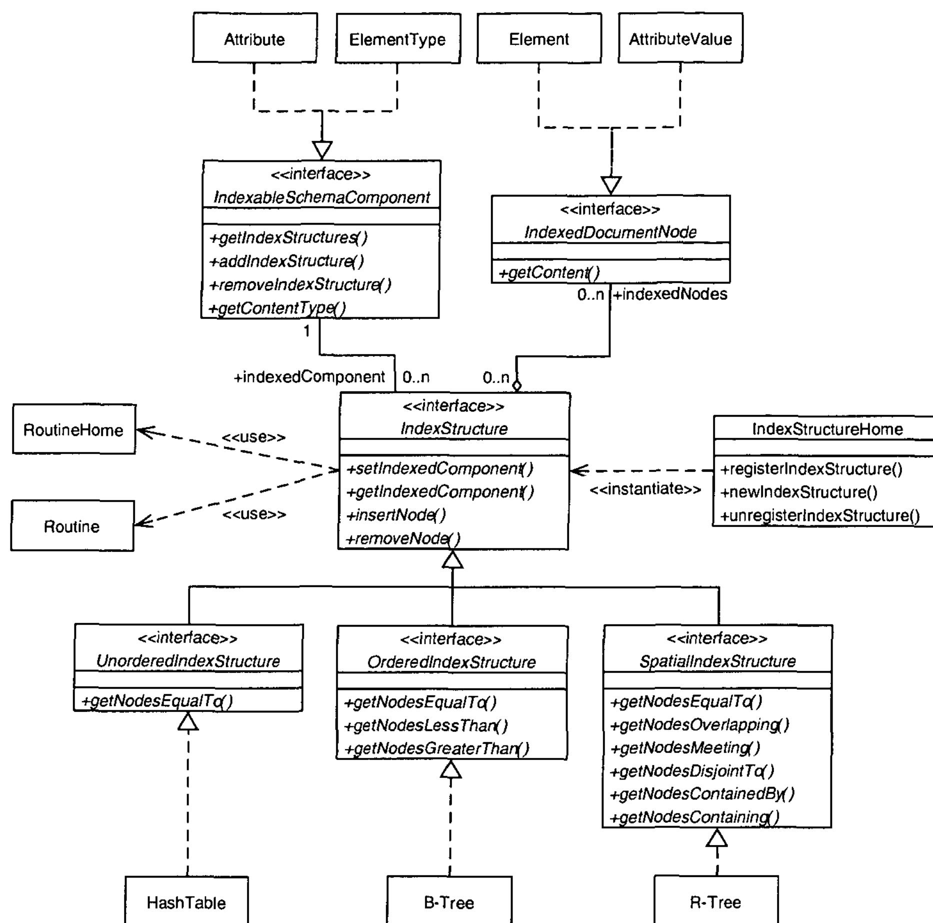


Figure 8.6: Index framework overview (UML class diagram)

constructed typed representation is based are updated by inserting the element via `insertNode()`.

The index framework distinguishes unordered, ordered, and spatial value index structures. These categories are represented by corresponding specializations of the `IndexStructure` interface. The framework can be straightforwardly extended with support for further categories of values index structures such as text index structures by providing further specializations of the `IndexStructure` interface. Each specialized interface defines the methods that are supported by the given category of value index structures for the retrieval of indexed document nodes. For example, spatial value index structures offer methods that allow to, being passed an instance of the content type[2] of the respective indexable schema component they are attached to (which can be obtained via the method `getContentType()` of `IndexableSchemaComponent`), retrieve all the document nodes they index whose content is equal to, overlapping with, meeting with, disjoint to, contained by, or

[2] Just like routine parameters in the routine framework, an instance of the content type of an indexable schema component constitutes an appropriate simple type instance if the content type is given by a simple type. If it is given by a complex type, a suitable instance is an element in typed representation filled according to that complex type.

containing the passed instance of the content type.

To integrate a concrete value index structure into the index framework, one has to supply a class that implements the index structure and supports the respective subinterface of `IndexStructure` for the category to which the value index structure belongs. This class must then be registered under a unique name with the framework's index structure home (represented by the corresponding class in the diagram) which serves as a registry of all value index structures available with PTDOM. Using that name, applications can then dynamically instantiate that index structure via the method `newIndexStructure()` offered by the index structure home and attach the structure to an indexable schema component. The index framework already ships with classes that implement hash tables, B-Trees, and R-Trees as ready-to-use examples of unordered, ordered, and spatial value index structures.

It is noteworthy that the implementations of the value index structures coming with the index framework are tied to the routine framework. The R-Tree implementation, for instance, expects that the functions `intersects()`, `meets()`, `contains()`, `interSectionArea()`, `union()`, and `area()` are registered with the framework's routine home for the content type of the indexable schema component to which an R-Tree is going to be attached. The R-Tree implementation employs these functions to organize the document nodes instantiating the schema component along their content in an R-Tree structure as well as to realize the retrieval functionality of the `SpatialIndexStructure` interface.

Founding the implementation of value index structures on routines of the routine framework is generally a good strategy as it broadens their applicability: applications can provide specialized implementations of these routines that consider the semantics of the application data to be indexed. For example, a song retrieval application based on MPEG-7 melody descriptions could provide implementations of the routines required by PTDOM's R-Tree implementation for the complex type `MelodyContourType` of Figure 2.4 considering the semantics of melody contours. When applying an R-Tree index structure to index the elements of type `MelodyContour` whose content is defined by means of `MelodyContourType`, this index structure could then be exploited to quickly and meaningfully retrieve all melody contours from the document manager which contain the contour of a melody fragment hummed by a user.

8.6 Query Evaluator

The components introduced so far provide a rich foundation for an efficient evaluation of queries on XML documents stored with PTDOM. Query evaluation can take advantage of the fine-grained and typed representation of XML document contents within the TDOM-based document manager for fine-grained and typed access to document contents, of the detailed representation of schema definitions within the schema catalog not just as a basis for query optimization but also – due to the coupling of elements and attribute values in typed representation in the document manager's documents to their respective element types and attibutes – for path indexing, of the routines provided by the routine framework for adequate processing of document contents, and of the rich set of value indexes offered by the index framework for speed-up of query evaluation. PTDOM's query evaluator component constitutes a first step towards a query processor that uses these opportunities.

Essentially, the query evaluator component (an overview of which is given by the class diagram of Figure 8.7) provides an implementation of the PTDOM query

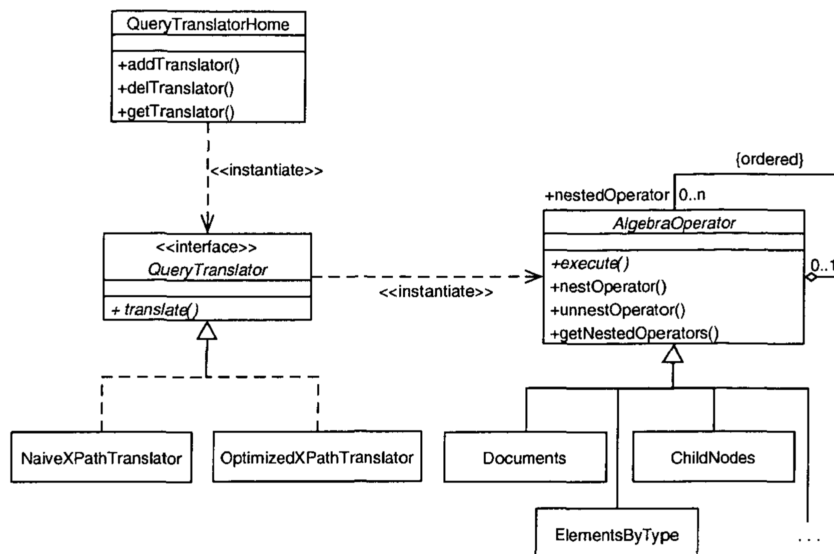


Figure 8.7: Query evaluator component overview (UML class diagram)

algebra [Rei03]. We have specifically developed this algebra to efficiently evaluate XPath expressions on all documents within the document manager that are valid to a given schema definition in the schema catalog exploiting PTDOM's specifics.[3] The query evaluator component implements the different operators of the algebra - each of which takes one or more sets of document nodes[4] and yields a set of document nodes as its result - as individual subclasses of the class `AlgebraOperator`. Within these subclasses, the behavior of the operators is realized by appropriate implementations of the abstract method `execute()` which returns the set of document nodes produced by an operator as an iterator to avoid the full materialization of these potentially large sets in memory. The sets of document nodes to which an operator's behavior is applied are either given by the result sets of other operators - the operators of the PTDOM query algebra can be nested to form operator trees as it is expressed by the ordered reflexive aggregation to the top right of the diagram - or taken from a queue of document node sets that has been passed to `execute()` in case an operator constitutes a leaf of the operator tree.

The query evaluator component further supplies a query translator home (modeled by a corresponding class) with which arbitrary query translators can be registered, classes supporting the `QueryTranslator` interface that are capable of producing equivalent PTDOM query algebra operator trees out of XPath expressions. The query evaluator component already ships with a translator which naively translates XPath expressions to the algebra as well as an optimized translator that exploits the path indexing abilities of schema definitions and value index structures to more

[3]Although not explicitly treated in this thesis, the algebra allows to limit the evaluation of XPath expressions to single documents as well.

[4]For compatibility with the XPath data model for XML documents [FMN02], not just the different kinds of document nodes distinguished by TDOM are considered as eligible document nodes for the operators of the PTDOM query algebra - i.e., elements, comments, processing instructions, and text nodes - but also attribute values and documents themselves. For the purpose of path traversal, a document's root nodes are considered as its child nodes and attribute values are regarded as child nodes of the elements they belong to.

likely generate more efficient operator trees.

PTDOM query algebra			
General operators			
documents(ns, s)	Delivers all documents in the document manager valid to Schema Definition s in the schema catalog ignoring the Node Set ns.	childNodes(ns)	Delivers the direct child nodes of all the nodes contained in Set ns.
descendantNodes (ns)	Delivers all direct and indirect child nodes of the nodes in Set ns.	parentNodes(ns)	Delivers all parent nodes of the nodes in Set ns.
ascendantNodes (ns)	Delivers all direct and indirect parent nodes of the nodes in Set ns.	union(ns1, ns2)	Calculates the union of the two sets of nodes ns1 and ns2.
Intersection (ns1, ns2)	Calculates the intersection of the two sets of nodes ns1 and ns2.	difference(ns1, ns2)	Calculates the difference between the two sets of nodes ns1 and ns2.
filterByParent-Nodes(ns1, ns2)	Selects all nodes from Set ns1 which have a direct parent node in Set ns2.	filterByAscendant-Nodes(ns1, ns2)	Selects all nodes from Set ns1 which have a direct or indirect parent node in Set ns2.
filterByChild-Nodes(ns1, ns2)	Selects all nodes from Set ns1 which have a direct child node in Set ns2.	filterByDescendant-Nodes(ns1, ns2)	Selects all nodes from Set ns1 which have a direct or indirect child node in Set ns2.
filterByName(ns, name, namespace)	Select all nodes from Set ns which have name name and namespace namespace.	filterByKind(ns, k)	Delivers all nodes from Set ns that are of the given class (=document, *element*, *root element*, *attribute* value, processing instruction, comment, text, doctype).
filterByPosition (ns, p)	Select all nodes from Set ns which are the p-th child node (p=1, ..., n, last) of their parent nodes.	filterByExistence (ns, operatortree)	Selects all nodes n from Set ns for which holds that operatortree({n}) returns at least one node in the same document.
filterByConst-Predicate(ns, operatortree, c, p)	Selects all nodes n from Set ns for which holds that the binary boolean predicate function p taken from the routine framework being passed the content of one node n1 from operatortree({n}) (elements and attribute values of the same document than n only) and constant c (simple type instance or element) yields true.	filterByPredicate (ns, operatortree1, operatortree2, p)	Selects all nodes n from Set ns for which holds that the binary boolean predicate function f taken from the routine framework being passed the content of one node n1 (elements and attribute values of the same document than n only) from operatortree1({n}) and one node n2 (elements and attribute values of the same document than n only) from operatortree2({n}) yields true.
Specialized operators			
elementsByType (ns, et)	Delivers all elements within the document manager's documents which instantiate the given Element Type et declared in one of the schema catalog's schema definitions, ignoring the Node Set ns.	valuesByAttribute (ns, at)	Delivers all attribute values within the document manager's documents which instantiate the given Attribute at declared in one of the schema catalog's schema definitions, ignoring the Node Set ns.
elementsByConst-Index(ns, et, rm, c)	Retrieves indexed elements from a value index defined on Element Type et supporting the retrieval method rm by passing constant c (simple type instance or element) to rm, ignoring the Node Set ns.	valuesByConstIndex (ns, at, rm, c)	Retrieves indexed attribute values from a value index defined on Attribute at supporting the retrieval method rm by passing constant c (simple type instance or element) to rm, ignoring the Node Set ns.
elementsByIndex (ns, et, rm)	Retrieves indexed elements from a value index defined on Element Type et supporting the retrieval method rm by successively passing the content of each node in Set ns (elements and attribute values only) to rm.	valuesByIndex(ns, at, rm)	Retrieves indexed attribute values from a value index defined on Attribute at supporting the retrieval method rm by successively passing the content of each node in Set ns (elements and attribute values only) to rm.

Figure 8.8: PTDOM query algebra overview

Figure 8.8 gives an overview of the major operators of the PTDOM query algebra. Roughly, these can be divided into two groups. The first group consists of general operators that constitute more or less direct mappings of the various location steps supported by the XPath language. These operators – although oper-

ators like `filterByConstPredicate` benefit from the fact that (simple) content of elements and attribute values in typed representation is kept in type-adequate and efficient simple type instances and not just as text and that arbitrary user-defined functions registered with the routine framework can be employed as predicates for comparison – do not rely on any specific characteristics of PTDOM. Employing only these operators, most XPath expressions can be straightforwardly expressed in the PTDOM query algebra, as it is actually done by the naive query translator.

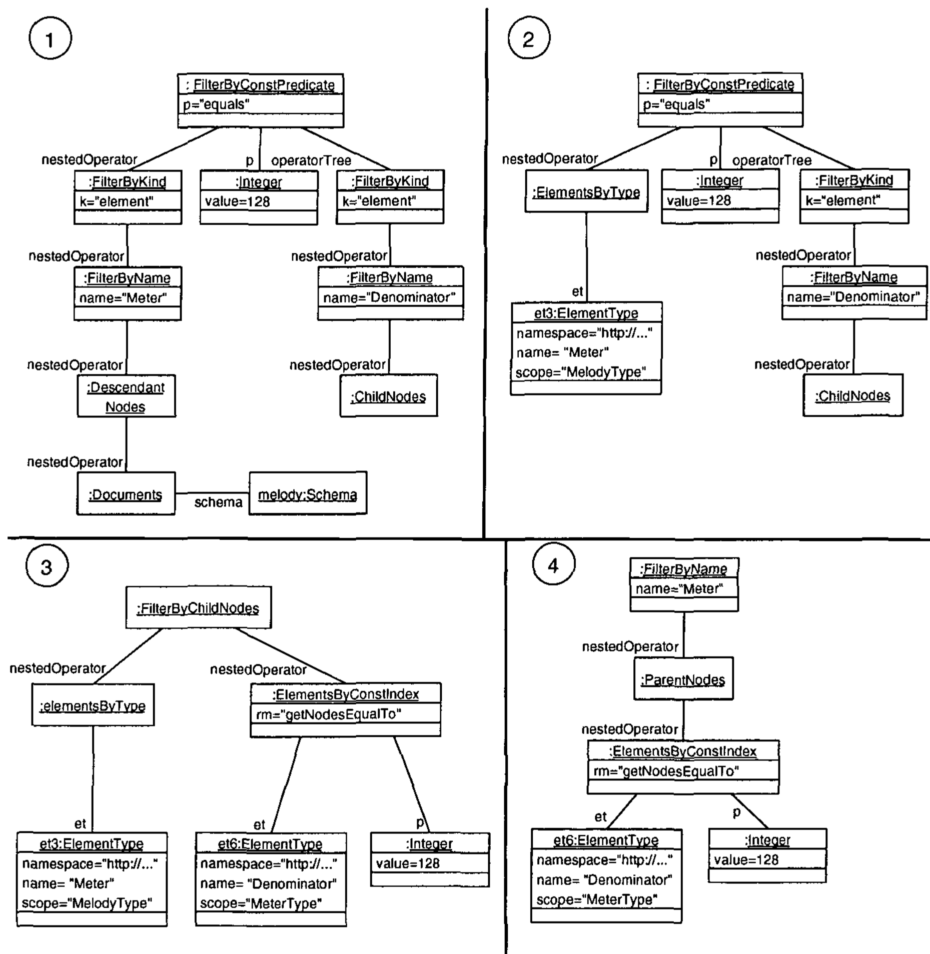


Figure 8.9: Query algebra example (UML object diagrams)

The object diagram to the upper left (1) of Figure 8.9 illustrates how the XPath expression `//Meter[Denominator/data() = 128]` can be evaluated on the XML documents within PTDOM's document manager that are valid to the `Melody` description scheme of Figure 2.4 using the query algebra's general operators. Outgoing from the document nodes of those documents that comply to the schema definition selected via the `documents` operator at the bottom, the depicted operator tree expensively traverses all direct and indirect child nodes of these document nodes via the `descendantNodes` operator on the search for `Meter` elements. Out of all these `Meter` elements, the `filterByConstPredicate` operator forming the root of the operator tree selects those elements employing an appropriate `equals()` function

registered with the routine framework's routine home that have a `Denominator` element among their child nodes with an integer simple content of 128.

The second group of operators of the PTDOM query algebra consists of specialized operators designed to utilize the path indexing capabilities of schema definitions within PTDOM as well as the value index structures offered by PTDOM's index framework. The operators `elementsByType` and `valuesByAttribute` exploit that elements and attribute values in typed representation are tightly coupled to the element types and attributes they instantiate to immediately obtain all elements and attribute values inside the document manager's documents that instantiate a given element type or attribute of one in the schema catalog's schema definitions. The upper right (2) object diagram of Figure 8.9 shows an alternative operator tree for the expression `//Meter[Denominator/data() = 128]` which, assuming that all `Meter` elements are kept in typed representation, employs an `elementsByType` operator to avoid the costly search for `Meter` elements over all nodes of the documents valid to the `Melody` description scheme by directly obtaining all elements instantiating the element type `Meter`.

Furthermore, the second group features operators that allow to obtain elements and attribute values from a value index that might be attached to an element type or attribute within a schema definition. Instead of explicitly navigating from every `Meter` element to its child elements in order to perform the (possibly fruitless) check whether there is an `Denominator` element with a simple content of 128, the object diagram to the bottom left (3) of Figure 8.9 employs an `elementsByConstIndex` operator to obtain all `Denominator` elements with a simple content of 128 from a value index that has been attached to the element type `Denominator`. The root of the operator tree consists of an `filterByChildNodes` operator that lets pass only those `Meter` elements which are parent elements of one of the `Denominator` elements retrieved from the index. As shown by the object diagram to the bottom right (4), this operator tree can even be further simplified. To avoid the filtering of potentially many `Meter` elements which do not have child nodes among the `Denominator` elements retrieved from the index, one can directly navigate from the retrieved `Denominator` elements to their parent elements and select all those with the element type `Meter` (if it is known from the schema definition, that `Denominator` elements always appear as children of `Meter` elements, the latter step is even unnecessary).

The optimized query translator shipping with the query evaluator component – albeit not performing query optimization in the traditional sense – tries to produce more efficient operator trees compared to the naive translator by applying an heuristic translation function that makes intensive use of the second group of operators of the PTDOM query algebra. An excerpt of this translation function is given by Figure 8.10. The function avoids, if possible, expensive access to all documents valid to the schema definition on which the XPath expression is evaluated via `documents` operators as well as full document traversals via `descendantNodes` operators. Essentially, this is achieved by picking named element types and attributes that form the beginning of XPath expressions or the beginning of subexpressions starting with `//`, translating them to corresponding `elementsByType` and `valuesByAttribute` operators, and using these operators as anchor points tree relative to which the rest of the expression is being translated. Furthermore, the translation function makes use of value index structures for the translation of conditions via `elementsByConstIndex`, `elementsByIndex`, `valuesByConstIndex`, and `valuesByIndex` operators if possible.

As an illustration of how the heuristic translation function works, the

Heuristic translation function h_s() for an optimized transformation of XPath expressions to PTDOM query algebra operator trees on the basis of Schema Definition s.

Navigational expressions
Note: et_s is an element type declared in Schema Definition s.

1. h_s(/)= documents(ns,s)	2. h_s(/*)= filterByKind(childNodes(documents(ns,s)), "element")	3. h_s(//*)= filterByKind(descendantNodes(documents(ns,s)), "element")	4. h_s(/et_s)= filterByKind(elementsByType(ns,et_s), "root element")
5. h_s(//et_s)= elementsByType(ns,et_s)	6. h_s(E/*)= filterByKind(childNodes(h_s(E)), "element")	7. h_s(E//*)= filterByKind(descendantNodes(h_s(E)), "element")	8. h_s(E/et_s)= filterByKind(filterByName(childNodes(h_s(E)), et_s), "element")
9. h_s(E//et_s)= filterByAscendant- Nodes(elementsByType(ns, et_s), h_s(E))	. . .	. . .	. . .

Conditions (normal translation)
Note: $ serves as a marker to indicate that a condition relative to the condition'context node in the XPath expression is being translated.

10. h_s(E1[E2])= filterByExistence(h_s(E1),h_s($E2))	11. h_s(E1[E2/data() p c])= filterByConst- Predicate(h_s(E1), h_s($E2),c,p)	12. h_s(E1[E2/data() p E3/data()])= filterByPredicate(h_s(E1),h_s($E2), h_s($E3),p)	13. h_s($/E)= h_s(/E)
14. h_s($//E)= h_s(//E)	15. h_s($*)= filterByKind(childNodes(ns), "element")	16. h_s($et_s)= filterByKind(filterByName(childNodes(ns),et_s), "element")	. . .

Conditions (exploitation of value index structures)
Note: it is assumed that a value index structure offering retrieval method rm_p equivalent to predicate function p is attached to element type et_s.

17. h_s(E[et_s/data() p c])= filterByChildNodes(h_s(E), elementsByConst- Index(ns,et_s,rm_p,c))	18. h_s(E1[et_s/data() p E3/data()])= filterByChildNodes(h_s(E1), elementsByIndex(h_s(E1/E3),et_s,rm_p))	19. h_s(E1[et_s/data() p /E3/data()])= filterByChildNodes(h_s(E1),elementsBy- Index(h_s(/E3),et_s, rm_p))	. . .

Figure 8.10: Heuristic XPath translation overview

optimized query translator will transform the example XPath expression `//Meter[Denominator/data() = 128]` to operator tree (2) of Figure 8.9 by applying rules 11, 16, and 5 of Figure 8.10 if no value index structures are attached to the element type `Denominator`. If they are, the optimized query translator will produce operator tree (3) via rules 17 and 5.

From the fact that in case a value index structure is attached to element type `Denominator`, operator tree (3) will be produced instead of operator tree (4) which is likely to be more efficient, it can be easily seen that the optimized query translator can only be regarded as a first step towards an optimized query evaluation and definitely constitutes no subsitute for a dedicated query optimizer which exploits the rich schema information available with PTDOM's schema catalog to transform operator trees to more efficient representations. Future work should therefore be directed at integrating a sophisticated query optimizer into the query evaluator component.

8.7 Experimental Results

So far, we have been emphasizing the conceptual benefits of PTDOM's schema-aware approach to XML document management with regard to database consistency, treatment of non-textual document contents, path indexing, and opportunities for query optimization. Compared to schema-ignoring approaches like Xindice, TEXTML, eXist, etc., the obvious drawback of PTDOM's approach is – apart from its inherent dependency on the availability of schema definitions – its higher complexity resulting from the efforts required for document validation, the typing of basic document contents, and the linking of elements and attribute values to their element types and attributes in schema definitions for path indexing.

In this section, we present initial experimental results providing evidence that this complexity should be acceptable for many XML database applications in practice and that PTDOM offers not just purely conceptual but also directly noticable benefits in terms of query performance. The experiments are based on a working Java prototype that almost fully implements the PTDOM architecture as described in this chapter and that employs PSEPro as its storage backend. An Athlon XP 3000+ 2.12 GHz PC with 512 MB DDR RAM and a Western Digital 180GB hard disk running Windows XP served as the computional platform for the experiments. All experiments were conducted five times, dropping the highest and lowest numbers and reporting the average of the middle three.

Schema definition import	
Size	342,292 bytes
Parsing and integrity check	1,182 msec
Compilation to typing automaton	4,463 msec
Persistent storage	1,261 msec
Total	6,906 msec

Figure 8.11: Results of schema definition import

The first experiment measures the effort required to import a complex MPEG-7 DDL schema definition into the schema catalog of an empty PTDOM database. As a veritable stress test for our prototype, we have compiled all media description schemes predefined by the MPEG-7 standard [ISO01c, ISO01d, ISO01e] into a single schema definition consisting of more than 370 complex type declarations organized in a deeply nested derivation hierarchy and exceeding 300KB in size.

The table of Figure 8.11 presents the outcome of this experiment. The table breaks down how much of the total duration required for the import of the schema definition was spent on parsing the schema definition into the catalog's internal model and checking the integrity of its schema components, on compiling the schema definition into an equivalent typing automaton, and on storing both schema definition and typing automaton via the PSEPro storage backend.

The effort spent on compiling the typing automaton is about four times larger than the effort spent on parsing and checking the integrity of the schema definition, both consuming about 80% of the total import time of 6.9 sec. Assuming that in typical application scenarios the import of schema definitions can be expected to happen rather infrequently compared to the import of documents and considering

the high complexity of the imported definition, the measured total import time is an acceptable result for an expectedly one-time effort.

	Document import								
	DB 1			DB 2			DB 3		
Number of documents	50			100			200		
	Smallest document	Document average	Largest document	Smallest document	Document average	Largest document	Smallest document	Document average	Largest document
Size	794 bytes	22,323 bytes	104,157 bytes	1,910 Bytes	21,195 bytes	102,146 bytes	495 bytes	22,328 bytes	106,037 bytes
Parsing and validation	15 msec	254 msec	1,578 msec	16 msec	218 msec	1,687 msec	16 msec	239 msec	1,641 msec
Typing	15 msec	201 msec	781 msec	16 msec	221 msec	1,328 msec	16 msec	278 msec	1,109 msec
Persistent storage	15 msec	367 msec	1,953 msec	47 msec	369 msec	2,000 msec	15 msec	509 msec	1,594 msec
Total	45 msec	822 msec	4,312 msec	79 msec	808 msec	5,015 msec	47 msec	1,026 msec	4,344 msec

Figure 8.12: Results of document import

The second experiment measures the effort required for importing XML documents following the schema definition of the first experiment into PTDOM's document manager. For this purpose, we prepared three test sets of 50, 100, and 200 randomly generated MPEG-7 media descriptions each consisting of possibly up to 500 melody descriptions complying to the description scheme of Figure 2.4. For each of these test sets, we prepared a dedicated database with the schema definition of the first experiment already imported into the schema catalog. Into these databases, we then successively imported the documents of the corresponding test sets.

The table of Figure 8.12 shows the outcome of the second experiment. For each database, the table summarizes how much of the total duration required for the import of a document was spent on parsing and validating the document against the schema definition of the first experiment using the typing automaton compiled from that definition, on typing the document's contents – i.e., constructing corresponding typed representations of the document's elements and attribute values linking them to the element types and attributes they instantiate in the schema catalog and creating appropriate simple type instances for their content – again using the typing automaton, and on making the document persistent via PSEPro. Numbers are given for the smallest and largest document in each test set as well as for the document average.

Not going into the exact numbers (considering the complexity of the schema definition, we think that the the efforts for document validation and typing are reasonable for the given document sizes), the interesting observation that can be made on this experiment is that the effort required for document typing is fairly equal to the effort required for document validation and parsing (for larger documents even less), on the average consuming about 25% of the total import time. Thus, if an application is already willing to take the effort of validating XML documents when importing them into a database as a means of ensuring database consistency, it should mostly be able to afford the additional effort of document typing as well. In such scenarios, PTDOM's schema-aware approach proves viable. If, however, an application has so harsh time constraints not allowing to spend the additional typing effort or even to validate a document before importing it into a database (with all incurring problems such as delegating the responsibility for a consistent

database state to applications), PTDOM certainly cannot be the XML database solution of choice.

	Document querying					
	DB 1		DB 2		DB 3	
	Naive translator	Optimized translator	Naive translator	Optimized translator	Naive translator	Optimized translator
1. //Meter	5,359 msec (5,363 hits)	844 msec (5,363 hits)	9,812 msec (9,905 hits)	1,344 msec (9,905 hits)	22,609 msec (21,100 hits)	2,719 msec (21,100 hits)
2. //Meter[Denominator /data()=4] (Hashtable attached to Denominator)	8,906 msec (674 hits)	2,812 msec (674 hits)	16,375 msec (1,270 hits)	6,984 msec (1,270 hits)	37,468 msec (2,659 hits)	15,301 msec (2,659 hits)
3. //Meter[Denominator /data()=4] (No indexing)	8,906 msec (674 hits)	5,422 msec (674 hits)	16,375 msec (1,270 hits)	10,110 msec (1,270 hits)	37,468 msec (2,659 hits)	21,859 msec (2,659 hits)
4. //Meter/Denominator	5,265 msec (5,363 hits)	2,390 msec (5,363 hits)	9,953 msec (9,905 hits)	4,234 msec (9,905 hits)	23,266 msec (21,100 hits)	8,171 msec (21,100 hits)
5. //AudioDescription-Scheme/*//Beat	5,922 msec (6,144 hits)	1,672 msec (6,144 hits)	11,422 msec (11,833 hits)	2,953 msec (11,833 hits)	23,890 msec (24,786 hits)	6,438 msec (24,786 hits)

Figure 8.13: Results of document querying

So far, we have been mainly concerned with the costs of choosing PTDOM for XML document management. We now want to highlight the benefits to be earned. The third experiment measures the time required by the query evaluator component to evaluate five XPath expressions on the three databases created in the second experiment. The experiment opposes optimized query translation, which makes intensive use of the specialized operators of the PTDOM query algebra tailored to exploit the path indexing abilities of the schema catalog and the value index structures of the index framework, and naive query translation, which utilizes the algebra's general operators only.

The table of Figure 8.13 presents the outcome of this experiment. As one would expect, optimized query translation outruns naive query translation considerably. For Queries 1 to 4, the `elementsByType` operator which exploits the path indexing abilities of PTDOM's schema catalog can be brought into play to avoid the evaluation of costly // document traversals. Queries 2 and 3 further show that respectable performance gains can be achieved from available value index structures when using optimized query translation, even in spite of the suboptimal utilization of these structures that has already been explained before in Section 8.6.

Significant increase in performance is also achieved for Query 5. Despite the fact that there is at least one `Beat` element below an `AudioDescriptionScheme` element in every document of our test sets and even though the optimized translator produces a quite complicated operator tree (which selects all `AudioDescriptionScheme` and `Beat` elements via `elementsByType` operators and filters all those `Beat` elements which have one of the `AudioDescriptionScheme` elements in the same document among their ancestors) naive query translation (which simply traverses down all elements in every document on the search for `Beat` elements below `AudioDescriptionScheme` elements) still cannot compete.

The third experiment shows once more the desirability of a dedicated query optimizer. With knowledge of the schema definition in the catalog, an optimizer could, as already illustrated in Section 8.6, not just get even more out of the value index structure for the evaluation of Query 2. It could further simplify Query 5 to `//Beat` which optimized translation could execute in a time comparable to Query 1. While not yet possessing such a query optimizer, the PTDOM architecture with the

rich schema information available in its schema catalog at least provides an ideal ground for its realization.

Chapter 9

Conclusion

In the present thesis, the application of XML database solutions for the management of MPEG-7 media descriptions has been explored. After an introduction to the area of metadata, metadata standards for digital media and MPEG-7, an extensive set of requirements has been developed that should be met by an XML database solution in order to be suitable for the management of MPEG-7 media descriptions. The set comprises requirements addressing the representation of and the access to MPEG-7 media descriptions, requirements treating the management of media description schemes to which MPEG-7 media descriptions comply, as well as requirements concerning extensibility and classic DBMS functionality.

Against these requirements, an exhaustive number of representative XML database solutions, native XML database solutions as well as XML extensions of traditional DBMS, have been analyzed. The analysis has revealed considerable deficiencies of the examined systems seriously threatening their qualification for the management of MPEG-7 media descriptions. The major problem with current XML database solutions, native solutions as well as database extensions, is that they largely neglect schema and type information available with media description schemes for the storage of and the access to media descriptions. As a consequence, large amounts of non-textual technical metadata typically contained in MPEG-7 media descriptions, such as frequency spectrums, color distributions, and object motion vectors, is simply treated as text, significantly hindering access and meaningful processing of these data.

This weakness has its origin in the fact that the schema catalogs offered by the examined solutions are not able to adequately handle MPEG-7 DDL schema definitions. Actually, quite a few of the examined solutions do not provide schema catalogs at all; and those who do usually employ the schema information contained therein for the validation of XML documents only but not for the typed storage of XML document contents.

A further weakness of the examined solutions is the lack of multidimensional value index structures obstructing the implementation of efficient multimedia retrieval applications on top of them. Also, the analysis has shown that profound extensibility with functionality and index structures as well as classic DBMS functionality like transaction support, fine-grained concurrency and access control, and mature backup support cannot be taken for granted.

Facing the deficiencies of current XML database solutions, this thesis has made three substantial contributions towards an XML database solution that better suits the needs of the management of MPEG-7 media descriptions.

Firstly, TDOM has been proposed as an object-oriented model for XML documents that is well-suited for the representation of MPEG-7 media descriptions. The central characteristics of TDOM can be summarized as follows:

- TDOM provides a detailed representation of an XML document permitting applications to access and manipulate its contents at any desired level of granularity.
- TDOM supplies the concept of typed representations with which the basic contents of an XML document are represented in a manner appropriate to the particular content type that has been declared within the schema definition to which the document complies. On the basis of typed representations, applications can reasonably access and process even complex non-textual contents such as vectors, lists, and matrices.
- TDOM provides a simple type framework allowing to integrate support for arbitrary simple data types and simple type derivation methods for use with typed representations.
- For cases where schema and type information is not available or where it is desired to represent XML document contents decoupled from any schema definition, TDOM offers untyped representations. These simply keep basic XML document contents as text just as traditional DOM.
- TDOM permits to pragmatically switch between typed and untyped representations depending on the needs of a particular task. This gives a lot of flexibility in working with the model, for instance when importing XML documents to TDOM or during document updates.

Secondly, typing automata have been proposed as an intermediary representation of schema definitions for XML documents which essentially can be characterized along these lines:

- Typing automata are an executable formalism for the intermediary representation of schema definitions for XML documents that is in principle independent of any particular XML schema definition language.
- Typing automata are able to validate XML documents in TDOM representation against the schema definition they represent and to infer and create appropriate typed representations of their contents.
- Typing automata are reasonably efficient. They can validate and type XML documents in a running time linear to the number of elements contained in the document and the number of transition rules of which they consist.
- Typing automata can be extended up to the expressiveness of MPEG-7 DDL. Therefore, they constitute an adequate means for the representation of MPEG-7 media description schemes.

Thirdly, equipped with TDOM and typing automata, the implementation of the PTDOM database prototype on the basis of the object-oriented DBMS ObjectStore has been described. Summarizing, the prototype has the following properties:

- PTDOM applies typing automata for the realization of a schema catalog that manages schema definitions for XML documents; it applies TDOM for the realization of a document manager component that manages XML documents instantiating these schema definitions.
- PTDOM's schema catalog and the document manager are tightly coupled: not only are the typing automata from the schema catalog used to create typed representations of the contents of the stored XML documents thereby giving applications typed access. Also, the schema catalog interconnects the individual attributes and element types in the schema definitions it manages with those attribute values and elements that instantiate them within the stored XML documents.
- The tight interconnection of schema definitions and instantiating XML documents constitutes a path index that opens up new ways of accessing XML document contents. This has been illustrated with a sketch of an XPath evaluator component that exploits these opportunities.
- Provided that an application scenario already allows to take the effort to validate documents while importing them into a database as a means of ensuring database consistency, the additional effort to produce type basic document contents as well as interlinking elements and attribute values with the element types and attributes they instantiate is likely to be acceptable as well.
- PTDOM supports a broad array of simple types for basic XML document contents, a variety of value index structures like hashtables, B-Trees, and even multidimensional R-Trees, as well as user-defined routines.
- PTDOM is profoundly extensible with new simple types, value index structures, and user-defined routines.
- PTDOM offers mature support for classic DBMS functionality which is inherited from the underlying DBMS ObjectStore serving as the storage backend.

As a consequence of these characteristics, the PTDOM prototype constitutes an XML database solution that satisfies most of the requirements for the management of MPEG-7 media descriptions and that is thus highly qualified for this purpose.

The impact of the present thesis is not just limited to the management of MPEG-7 media descriptions. TDOM, typing automata, and PTDOM constitute generic results that are applicable to the management of arbitrary XML documents. Hence, the contributions this thesis lay important foundations for a new generation of general XML database solutions that thoroughly exploit available schema and type information for the adequate storage of XML documents. The existence of such XML database solutions is highly desirable in any application domain in which data-centric XML documents with large amounts of non-textual data have to be managed. Examples of such domains are electronic data interchange and the exchange of scientific data.

The results of this thesis pave the way to future research work:

- As it has already been illustrated by means of an XPath evaluator, the tight coupling of schema definitions with the XML documents instantiating these definitions that is performed by PTDOM opens up new access paths to XML document contents. It would be interesting to investigate how these access paths can be exploited for an efficient implementation of a full-fledged XML

query or transformation language, such as XQuery and XSLT. It would also be interesting to examine how this coupling can be exploited for the realization of sophisticated query optimizers.

- PTDOM could be integrated with a generic tool suite for the management of MPEG-7 media descriptions and description schemes. Such a suite could include tools for editing media description schemes, tools for editing media descriptions following these description schemes, as well as search tools for the retrieval of media descriptions. As such tools are intended for use by end users, the challenge is to provide tools that on the one hand are generically applicable but on the other hand hide the technical peculiarities of MPEG-7 DDL and the XML representation of MPEG-7 media descriptions behind intuitive user interfaces.

- With TDOM, typing automata, and PTDOM, this thesis has supplied the means necessary to set up a generically applicable database for MPEG-7 media descriptions. It would be interesting to examine how such an MPEG-7 database can be integrated into a larger multimedia database which not only manages media descriptions but also media data and multimedia compositions and which supports the full process of content production: from raw media production, media postprocessing, authoring of multimedia content, and annotation with descriptional metadata to media dissemination and presentation.

Bibliography

[AIT00] AITF. Categories for the Description of Works of Art. AITF Standard Version 2.0, Art Information Task Force (AITF), September 2000.

[Ana01] Analysis & Design Platform Task Force. Unified Modeling Language (UML). OMG Available Specification Version 1.4, Object Management Group (OMG), September 2001.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, et al. The Lorel Query Language for Semistructured Data. *Journal of Digital Libraries*, 1(1), 1997.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullmann. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachussetts, 1986.

[BAOG98] K. Böhm, K. Aberer, M.T. Öszu, and K. Gayer. Query Optimization for Structured Documents Based on Knowledge on the Document Type Definition. In *Proc. of the IEEE Forum on Research and Technology Advances in Digital Libraries (ADL '98)*, Santa Barbara, California, April 1998.

[BCF+02] S. Boag, D. Chamberlin, M.F. Fernandez, et al. XQuery 1.0: An XML Query Language. W3C Working Draft, World Wide Web Consortium (W3C), November 2002.

[BK89] E. Bertino and W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), 1989.

[BK95] P.A. Boncz and M.L. Kersten. Monet – An Impressionist Sketch of an Advanced Database System. In *Proc. of the Basque International Workshop on Information Technology (BIWITT'95)*, San Sebastian, Spain, July 1995.

[BKS98] S. Boll, W. Klas, and A. Sheth. Overview on Using Metadata to Manage Multimedia Data. In A. Sheth and W. Klas, editors, *Multimedia Data Management: Using Metadata To Integrate and Apply Digital Media.* McGraw-Hill, New York, 1998.

[BM01] P.V. Biron and A. Mahotra. XML Schema Part 2: Datatypes. W3C Recommendation, World Wide Web Consortium (W3C), May 2001.

[Bou02] R. Bourret. XML Database Products. Online Article, available under http://www.rpbourret.com/xml/XMLDatabaseProds.htm, May 2002.

[BPSMM00] T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, World Wide Web Consortium (W3C), October 2000.

[BR94] K. Böhm and T.C. Rakow. Metadata for Multimedia Documents. *ACM SIGMOD Record*, 23(4), 1994.

[CD99] J. Clark and S. DeRose. XML Path Language (XPath). W3C Recommendation Version 1.0, World Wide Web Consortium (W3C), November 1999.

[CDF+94] M.J. Carey, D.J. DeWitt, M.J. Franklin, et al. Shoring Up Persistent Applications. In *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data (ACM SIGMOD 1994)*, Minneapolis, Minnesota, May 1994.

[CDG+02] H. Comon, M. Dauchet, R. Gilleron, et al. Tree Automata Techniques and Applications. Unpublished Book Manuscript, October 2002. Available at: http://www.grappa.univ-lille3.fr/tata/tata.pdf.

[CFS99] K. Curtis, P.W. Foster, and F. Stentiford. Metadata – The Key to Content Management Services. In *Proc. of the 3rd IEEE Meta-Data Conference*, Bethesda, Maryland, April 1999.

[Chi00] B. Chidlovskii. Using Regular Tree Automata as XML Schemas. In *Proc. of the IEEE Advances in Digital Libraries 2000 (ADL 2000)*, Washington, D.C., May 2000.

[Cla99] J. Clark. XSL Transformations (XSLT). W3C Recommendation, World Wide Web Consortium (W3C), November 1999.

[Cla01] J. Clark. TREX – Tree Regular Expressions for XML Language Specification. Specification, Thai Open Source Software Center, Ltd., February 2001.

[CLKR02] B. Chang, E. Litani, J. Kesselman, and R. Rahman. Document Object Model (DOM) Level 3 Abstract Schemas Specification. W3C Note Version 1.0, World Wide Web Consortium (W3C), July 2002.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachussetts, 1990.

[CM01] J. Clark and M. Murata. RELAX NG Specification. OASIS Committee Specification, Organization for the Advancement of Structured Information Standards (OASIS), December 2001.

[CRF00] D.D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Proc. of the 3rd International Workshop on the Web and Databases (WebDB 2000)*, Dallas, Texas, May 2000.

[CT01] R. Cowan and R. Tobin. XML Information Set. W3C Recommendation, World Wide Web Consortium (W3C), October 2001.

[DCM99] DCMI. Dublin Core Metadata Element Set. DCMI Recommendation Version 1.1, Dublin Core Metadata Initiative (DCMI), July 1999.

[DFF+98] A. Deutsch, M. Fernandez, D. Florescu, et al. XML-QL: A Query Language for XML. In *Proc. of the W3C Query Languages Workshop (QL'98)*, Boston, Massachussets, December 1998.

[DFS99] A. Deutsch, M. Fernandez, and D. Suciu. Storing Semistructured Data with STORED. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD 1999)*, Philadelphia, Pennsylvania, June 1999.

[DN01] Y. Duchesne and P. Nyfeld. Ozone Developer's Guide. System Documentation Version 1.0, SMB GmbH, 2001.

[EN94] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems – Second Edition.* The Benjamin/Cummings Publishing Company, Redwood City, California, 1994.

[eXc00] eXcelon Corp. Java API User Guide. System Documentation Release 6.0 Service Pack 5, eXcelon Corp., June 2000.

[eXc01a] eXcelon Corp. Managing DXE. System Documentation Release 3.5, eXcelon Corp., December 2001.

[eXc01b] eXcelon Corp. PSE Pro for Java API User Guide. System Documentation Release 6.0 Service Pack 5, eXcelon Corp., April 2001.

[Fal01] D.C. Fallside. XML Schema Part 0: Primer. W3C Recommendation, World Wide Web Consortium (W3C), May 2001.

[FBY92] W.B. Frakes and R. Baeza-Yates, editors. *Information Retrieval – Data Structures & Algorithms.* Prentice Hall, Upper Saddle River, New Jersey, 1992.

[FHK+02] T. Fiebig, S. Helmer, C.C. Kanne, G. Moerkotte, et al. Anatomy of a Native XML Base Management System. *The VLDB Journal*, 11(4), 2002.

[FK99] D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), 1999.

[FK01] N. Fatemi and O.A. Khaled. Indexing and Retrieval of TV News Programs Based on MPEG-7. In *Proc. of the IEEE International Conference on Consumer Electronics (ICCE 2001)*, Los Angeles, California, June 2001.

[FM00] T. Fiebig and G. Moerkotte. Evaluating Queries on Structure with Extended Access Support Relations. In *Proc. of the World Wide Web and Databases, Third International Workshop (WebDB 2000)*, Dallas, Texas, May 2000.

[FM01] T. Fiebig and G. Moerkotte. Algebraic XML Construction and its Optimization in Natix. *World Wide Web*, 4(3), 2001.

[FMN02] M. Fernandez, J. Marsh, and M. Nagy. XQuery 1.0 and XPath 2.0 Data Model. W3C Working Draft, World Wide Web Consortium (W3C), November 2002.

[FS98] M. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Proc. of the Fourteenth International Conference on Data Engineering (ICDE '98)*, Orlando, Florida, February 1998.

[FT98] C. Frankston and H.S. Thompson. XML-Data Reduced. Unpublished Draft of W3C Note Version 0.21, University of Edinburgh, July 1998.

[GB97] R.V. Guha and T. Bray. Meta Content Framework Using XML. W3C Note, World Wide Web Consortium (W3C), June 1997.

[GD02] W. Gietz and C. Dupree. Oracle 9i Data Cartridge Developer's Guide . System Documentation Release 2 (9.2), Oracle Corp., March 2002.

[GG98] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2), 1998.

[Gil98] A.J. Gilliland-Swetland. Defining Metadata. In M. Baca, editor, *Introduction to Metadata: Pathways to Digital Information.* Getty Information Institute, Los Angeles, California, 1998.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *Proc. of the ACM SIGMOD Workshop on The Web and Databases (WebDB '99)*, Philadelphia, Pennsylvania, June 1999.

[Gre00] J. Greenberg, editor. *Metadata and Organizing Educational Resources on the Internet.* The Haworth Information Press, New York, 2000.

[GSN99] G. Gardarin, F. Sha, and T.D. Ngoc. XML-Based Components for Federating Multiple Heterogeneous Data Sources. In *Proc. of the 18th International Conference on Conceptual Modeling (Conceptual Modeling - ER '99)*, Paris, France, November 1999.

[GW97] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. of the 23rd International Conference on Very Large Data Bases (VLDB '97)*, Athens, Greece, August 1997.

[HAA+02] S. Higgins, N. Agarwal, A. Agrawal, et al. Oracle 9i XML Database Developer's Guide – Oracle XML DB. System Documentation Release 2 (9.2), Oracle Corp., March 2002.

[HMF99] G. Huck, I. Macherius, and P. Fankhauser. PDOM: Lightweight Persistency Support for the Document Object Model. In *Proc. of the Workshop "Java and Databases: Persistence Options" of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, Denver, Colorado, November 1999.

[Hos00] H. Hosoya. *Regular Expression Types for XML.* PhD thesis, University of Tokyo, Japan, 2000.

[HS02] J. Hunter and C. Seyrat. Description Definition Language. In B.S. Manjunath, P. Salembier, and T. Sikora, editors, *Introduction to MPEG-7.* John Wiley & Sons, West Sussex, UK, 2002.

[IBM00] IBM Corp. IBM DB2 Universal Database – XML Extender Administration and Programming. System Documentation Version 7, IBM Corp., 2000.

[IBM01a] IBM Corp. IBM DB2 Spatial Extender – User's Guide and Reference. System Documentation Version 7, IBM Corp., June 2001.

[IBM01b] IBM Corp. IBM DB2 Universal Database – Text Extender Administration and Programming. System Documentation Version 7, IBM Corp., March 2001.

[IEE02] IEEE P1484.12 Learning Object Metadata Working Group. Draft Standard for Learning Object Metadata. IEEE Final Draft Standard P1484.12.1-2002, IEEE Learning Technology Standards Committee (LTSC), July 2002.

[Inf02] Infonyte GmbH. Infonyte-DB – User Manual and Programmers Guide. System Documentation Version 2.0.2, Infonyte GmbH, May 2002.

[ISO99] ISO/IEC JTC 1/SC 29/WG 11. MPEG-7: Context, Objectives and Technical Roadmap, V.12. ISO/IEC Document N2861, International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), July 1999.

[ISO00] ISO/IEC JTC 1/SC 34. SGML Applications – Topic Maps. ISO/IEC International Standard ISO/IEC 13250:2000, International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), February 2000.

[ISO01a] ISO/IEC JTC 1/SC 29/WG 11. Information Technology – Multimedia Content Description Interface – Part 1: Systems. ISO/IEC Final Draft International Standard 15938-1:2001, International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), November 2001.

[ISO01b] ISO/IEC JTC 1/SC 29/WG 11. Information Technology – Multimedia Content Description Interface – Part 2: Description Definition Language. ISO/IEC Final Draft International Standard 15938-2:2001, International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), September 2001.

[ISO01c] ISO/IEC JTC 1/SC 29/WG 11. Information Technology – Multimedia Content Description Interface – Part 3: Visual. ISO/IEC Final Draft International Standard 15938-3:2001, International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), July 2001.

[ISO01d] ISO/IEC JTC 1/SC 29/WG 11. Information Technology – Multimedia Content Description Interface – Part 4: Audio. ISO/IEC Final Draft International Standard 15938-4:2001, International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), June 2001.

[ISO01e] ISO/IEC JTC 1/SC 29/WG 11. Information Technology – Multimedia Content Description Interface – Part 5: Multimedia Description Schemes. ISO/IEC Final Draft International Standard 15938-5:2001, International Organization for Standardization/International Electrotechnical Commission (ISO/IEC), October 2001.

[IXI01] IXIASOFT Inc. Creating Client Applications for TEXTML Server – Programmer's Guide. System Documentation Version 2.1, IXIASOFT Inc., December 2001.

[JAC+02] H.V. Jagadish, S. Al-Khalifa, A. Chapman, et al. TIMBER: A Native XML Database. *The VLDB Journal*, 11(4), 2002.

[Jel99] R. Jelliffe. Using XSL as a Validation Language. Draft Technical Document, Academia Sinica, Taipei, Taiwan, January 1999. Available at: http://www.ascc.net/xml/en/utf-8/XSLvalidation.html.

[Jel02] R. Jelliffe. The Schematron Assertion Language 1.5. Specification Version 1.5, Academia Sinica, Taipei, Taiwan, October 2002.

[JLS99] H.V. Jagadish, L.V.S. Lakshmanan, and D. Srivastava. Hierarchical or Relational? A Case for a Modern Hierarchical Data Model. In *Proc. of the IEEE Workshop on Knowledge and Data Engineering Exchange (KDEX'99)*, Chicago, Illinois, November 1999.

[KM99] C.C. Kanne and G. Moerkotte. Efficient Storage of XML Data. Technical Report 8/99, University of Mannheim, Germany, August 1999.

[KMRT96] T. Krauskopf, J. Miller, P. Resnick, and W. Treese. PICS Label Distribution Label Syntax and Communication Protocols. W3C Recommendation Version 1.1, World Wide Web Consortium (W3C), October 1996.

[Kos02] H. Kosch. MPEG-7 and Multimedia Database Systems. *ACM SIGMOD Record*, 31(2), 2002.

[KSS95] V. Kashyap, K. Shah, and A. Sheth. Metadata for Building the Multimedia Patch Quilt. In S. Jajodia and V.S. Subrahmanian, editors, *Multimedia Database Systems: Issues and Research Directions.* Springer Verlag, London, UK, 1995.

[KV00] M. Kempa and V.Linnemann. Efficient Parsing of XML Documents without Limitations: DTD implies LL(1) Grammar (in German). Technical Report: Schriftenreihe der Institute für Informatik und Mathematik A-00-21, University of Lübeck, Germany, December 2000.

[KW01] T. Kunieda and Y. Wakita. XML Schema Dynamic Mapped Multimedia Content Description Tool. In *Proc. of the 20th International Conference on Conceptual Modeling (ER 2001)*, Yokohama, Japan, November 2001.

[LBK02] P.M. Lewis, A. Bernstein, and M. Kifer. *Databases and Transaction Processing – An Application-Oriented Approach.* Addison-Wesley, New York, 2002.

[LC00] D. Lee and W.W. Chu. Comparative Analysis of Six XML Schema Languages. *ACM SIGMOD Record*, 29(3), 2000.

[LLW+00] A. Le Hors, P. Le Hégaret, L. Wood, et al. Document Object Model (DOM) Level 2 Core Specification. W3C Recommendation Version 1.0, World Wide Web Consortium (W3C), November 2000.

[LLW+03] A. Le Hors, P. Le Hégaret, L. Wood, et al. Document Object Model (DOM) Level 3 Core Specification. W3C Working Draft Version 1.0, World Wide Web Consortium (W3C), June 2003.

[LM00] A. Laux and L. Martin. XUpdate – XML Update Language. XML:DB Working Draft 2000-09-14, XML:DB Initiative, September 2000.

[LS99] O. Lassila and R.R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation, World Wide Web Consortium (W3C), February 1999.

[MAA+02a] C. McGregor, O. Alonso, S. Alpha, et al. Oracle Text – Reference. System Documentation Release 2 (9.2), Oracle Corp., March 2002.

[MAA+02b] C. Murray, D. Abugov, N. Alexander, et al. Oracle Spatial – User's Guide and Reference. System Documentation Release 2 (9.2), Oracle Corp., March 2002.

[Mar02] J.M. Martinez. MPEG-7 – Overview of MPEG-7 Description Tools, Part 2. *IEEE MultiMedia*, 9(3), 2002.

[Met98] Metadata Ad Hoc Working Group. Content Standard for Digital Geospatial Metadata. FDGC Standard FGDC-STD-001-1998, Federal Geographic Data Committee (FGDC), June 1998.

[Mey02] W.M. Meyer. eXist User's Guide. System Documentation Version 0.71, 2002.

[Mic00a] Microsoft Corp. Microsoft SQL Server 2000 – Creating and Maintaining Databases. System Documentation, Microsoft Corp., 2000.

[Mic00b] Microsoft Corp. Microsoft SQL Server 2000 – SQLXML 2.0. System Documentation, Microsoft Corp., 2000.

[MKP02] J.M. Martinez, R. Koenen, and F. Pereira. MPEG-7 – The Generic Multimedia Content Description Standard, Part 1. *IEEE MultiMedia*, 9(2), 2002.

[ML02] M. Mani and D. Lee. XML to Relational Conversion using Theory of Regular Tree Grammars. In *Proc. of the First VLDB Workshop on Efficiency and Effectiveness of XML Tools and Techniques (EEXTT 2002)*, Hongkong, China, August 2002.

[MMRW02] A. Malhotra, J. Melton, J. Robie, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Working Draft, World Wide Web Consortium (W3C), November 2002.

[Møl03] A. Møller. Document Structure Description 2.0. Specification Version 2.0, BRICS, University of Aarhus, Denmark, 2003.

[MS99] T. Milo and D. Suciu. Index Structures for Path Expressions. In *Proc. of the 7th International Conference on Database Theory (ICDT 1999)*, Jerusalem, Israel, January 1999.

[MSS02] B.S. Manjunath, P. Salembier, and T. Sikora, editors. *Introduction to MPEG-7.* John Wiley & Sons, West Sussex, UK, 2002.

[Mur99] M. Murata. Hedge Automata: a Formal Model for XML Schemata. Draft Technical Document, Fuji Xerox Information Systems, Fuji Xerox Co., Ltd., Tokyo, Japan, October 1999.

[Net02] Network Development and MARC Standards Office. MARC 21 Concise Format for Bibliographic Data. Library of Congress Standard 2002 Concise Edition, Library of Congress, October 2002.

[NIS02] NISO. Data Dictionary – Technical Metadata for Digital Still Images. NISO Draft Standard NISO Z39.87-2002, National Information Standards Organization (NISO), June 2002.

[NL99a] F. Nack and A.T. Lindsay. Everything You Wanted to Know About MPEG-7: Part 1. *IEEE MultiMedia*, 6(3), 1999.

[NL99b] F. Nack and A.T. Lindsay. Everything You Wanted to Know About MPEG-7: Part 2. *IEEE MultiMedia*, 6(4), 1999.

[PH01] L. Poola and J. Haritsa. SphinX: Schema-conscious XML Indexing. Technical Report TR-2001-04, Database Systems Lab, Indian Institute of Science, Bangalore, India, 2001.

[PM01] S. Pepper and G. Moore. XML Topic Maps (XTM) 1.0. TopicMaps.Org Specification, TopicMaps.org Consortium, August 2001.

[Pre98] P. Prescod. Formalizing XML and SGML Instances with Forest Automata Theory. Draft technical document, School of Computer Science, University of Waterloo, Canada, May 1998. Available at: http://www.prescod.net/forest/shorttut.

[PV03] Y. Papakonstantinou and V. Vianu. Incremental Validation of XML Documents. In *Proc. of the 9th International Conference on Database Theory (ICDT 2003)*, Siena, Italy, January 2003.

[Rei03] M. Reis. Eine optimierte Algebra zur Auswertung von XPath-Ausdrücken in einem Typed DOM (in German). Diploma Thesis, Dept. of Computer Science and Business Informatics, University of Vienna, Austria, 2003.

[RLS98] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). In *Proc. of the W3C Query Languages Workshop (QL'98)*, Boston, Massachussets, December 1998.

[SK98] A. Sheth and W. Klas, editors. *Multimedia Data Management: Using Metadata To Integrate and Apply Digital Media.* McGraw-Hill, New York, 1998.

[SKW+00] A. Schmidt, M. Kersten, M. Windhouwer, et al. Efficient Relational Storage and Retrieval of XML Documents. In *Proc. of the Third International Workshop on the Web and Databases (WebDB 2000)*, Dallas, Texas, May 2000.

[Sof01a] Software AG. Tamino X-Query. System Documentation Version 3.1.1, Software AG, November 2001.

[Sof01b] Software AG. User Guide. System Documentation Version 3.1.1, Software AG, November 2001.

[ST01] A. Salminen and F.W. Tompa. Requirements for XML Document Database Systems. In *Proc. of the ACM Symposium on Document Engineering 2001 (DocEng '01)*, Atlanta, Georgia, November 2001.

[Sta01] K. Staken. dbXML Developers Guide 0.5. System Documentation Version 1.0, The dbXML Project, September 2001.

[Sta02] K. Staken. Xindice Developers Guide 0.7. System Documentation Version 1.0, The Apache Software Foundation, March 2002.

[Ste90] G.L. Steele. *Common Lisp: The Language.* Digital Press, Maynard, Massachussetts, Second edition, 1990.

[STH+99] J. Shanmugasundaram, K. Tufte, G. He, et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proc. of the 25th International Conference on Very Large Data Bases (VLDB '99)*, Edinburgh, Scotland, September 1999.

[SV02] L. Segoufin and V. Vianu. Validating Streaming XML Documents. In *Proc. of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2002)*, Madison, Wisconsin, June 2002.

[SYU99] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *Proc. of the Database and Expert Systems Applications, 10th International Conference (DEXA '99)*, Florence, Italy, September 1999.

[TBM+01] H.S. Thompson, D. Beech, M. Maloney, et al. XML Schema Part 1: Structures. W3C Recommendation, World Wide Web Consortium (W3C), May 2001.

[TDCZ02] F. Tian, D.J. DeWitt, J. Chen, and C. Zhang. The Design and Performance Evaluation of Alternative XML Storage Strategies. *ACM SIGMOD Record*, 31(1), 2002.

[The02] The PostgreSQL Global Development Group. The PostgreSQL 7.3devel Documentation. System Documentation Version 7.3, The PostgreSQL Global Development Group, 2002.

[Thu02] B. Thuraisingham. *XML Databases and the Semantic Web.* CRC Press, Boca Raton, Florida, 2002.

[TIHW01] I. Tatarinov, Z.G. Ives, A.Y. Halevy, and D.S. Weld. Updating XML. In *Proc. of the 2001 ACM SIGMOD International Conference on Management of Data (ACM SIGMOD 2001)*, Santa Barbara, California, May 2001.

[VRA02] VRA Data Standards Committee. VRA Core Categories. VRA Standard Version 3.0, Visual Resources Assocation (VRA), February 2002.

[WK02a] U. Westermann and W. Klas. A Typed DOM for the Management of MPEG-7 Media Descriptions. Technical Report TR-2002301, Dept. of Computer Science and Business Informatics, University of Vienna, Austria, March 2002. *Available at http://www.informatik.univie.ac.at/institute/-index.html?staffPublication-8=8.*

[WK02b] U. Westermann and W. Klas. An Analysis of XML Database Solutions Concerning the Management of MPEG-7 Media Descriptions. Technical Report TR-2002302, Dept. of Computer Science and Business Informatics, University of Vienna, Austria, September 2002. *Available at http://www.informatik.univie.ac.at/institute/-index.html?staffPublication-8=8.*

[WK03a] U. Westermann and W. Klas. A Typed Representation and Type Inference for MPEG-7 Media Descriptions. Technical Report TR-2003301, Dept. of Computer Science and Business Informatics, University of Vienna, Austria, February 2003. *Available at http://www.informatik.univie.ac.at/institute/-index.html?staffPublication-8=8.*

[WK03b] U. Westermann and W. Klas. An Analysis of XML Database Solutions for the Management of MPEG-7 Media Descriptions. *ACM Computing Surveys*, 35(4), 2003.

[WK03c] U. Westermann and W. Klas. PTDOM – A Native Schema-Aware XML Database Solution. Technical Report TR-2003303, Dept. of Computer Science and Business Informatics, University of Vienna, Austria, December 2003. *Available at http://www.informatik.univie.ac.at/institute/-index.html?staffPublication-8=8. Currently under review for VLDB Journal.*

[WK04] U. Westermann and W. Klas. A Typed DOM for the Management of MPEG-7 Media Descriptions. *Accepted for publication in Multimedia Tools and Applications*, 2004.

[Woo99] P.T. Wood. Optimising Web Queries Using Document Type Definitions. In *Proc. of the 2nd Workshop on Web Information and Data Management (WIDM'99)*, Kansas City, Missouri, November 1999.

[X-H02] X-Hive Corp. X-Hive/DB 2.1 – Manual. System Documentation Release 2.0.2, X-Hive Corp., May 2002.

[XML01] XML Global Technologies, Inc. GoXML DB Administrator Help. System Documentation Version 2.0.1, XML Global Technologies, Inc., December 2001.

[XML03] XML:DB Initiative for XML Databases. Frequently Asked Questions about XML:DB. Online Article, available under http://www.xmldb.org/faqs.html, February 2003.

[YBL+01] K. Yoon, S.Y. Bae, J.E. Lee, et al. MPEG-7 Based News Browsing: Description Extraction, Browsing and Exchange. *Proceedings of SPIE*, 4518, 2001.

Lebenslauf

Name:	Gerd Utz Westermann
Wohnhaft:	Hebragasse 5/25 A-1090 Wien
Geboren:	17. Februar 1973 in Hüttental, heute Siegen, Deutschland
Vater:	Gerd Friedrich Westermann, Studiendirektor
Mutter:	Marie-Luise Westermann, geb. Henners, Verkäuferin
Nationalität:	deutsch
Familienstand:	ledig

Schule, Ausbildung und Studium

1979 - 1983	Gemeinschaftsgrundschule Krombach, Deutschland
1983 - 1992	Friedrich-Flick-Gymnasium, Kreuztal, Deutschland Abschluß: Abitur (Note: 1,7)
1992 - 1993	Wehrdienst
1993 - 1998	Studium der Informatik an der Universität Ulm, Deutschland Abschluß: Diplom-Informatiker (Note: 1,0) Diplomarbeitsthema: Repräsentation flexibel wiederverwendbarer multimedialer Dokumente in einem DBMS
Seit 1.3.2001	Doktoratsstudium an der Technischen Universität Wien

Beschäftigungsverhältnisse

1993	Soldat auf Zeit bei der 13./ Technische Schule der Luftwaffe 2, Erndtebrück, Deutschland
1995	Praktikum bei der Liebherr-Aero-Technik GmbH, Lindenberg, Deutschland
1997 - 1998	Praktikum bei der bei der debis Systemhaus GEI mbH, Ulm, Deutschland
1998 - 2000	Wissenschaftlicher Mitarbeiter in der Abteilung Datenbanken und Informationssysteme der Fakultät für Informatik, Universität Ulm, Deutschland
Seit 1.9.2000	Universitätsassistent in der Abteilung Multimedia Informationssysteme des Instituts für Informatik und Wirtschaftsinformatik, Universität Wien