Technical Report, No. TR-2003303, Dept. of Computer Science and Business Informatics, University of Vienna, December, 2003

– PTDOM –
A Native Schema-Aware XML Database Solution

Utz Westermann, Wolfgang Klas
Department of Computer Science and Business Informatics
University of Vienna, Austria
{gerd-utz.westermann,wolfgang.klas}@univie.ac.at

Abstract

Current XML database solutions largely neglect or make insufficient use of available schema definitions. Although XML documents are to some extent self-describing by means of their markup, availability and use of rich structure and type information contained in schema definitions written in languages such as XML Schema is nevertheless essential for their effective management in a database: only with this information at hand, it is possible to ensure database consistency by document validation, to store, index, and access the content of elements and attribute values in an appropriately typed manner and not just as text, and to perform more sophisticated query optimizations. In this paper, we give a system overview of the Persistent Typed DOM (PTDOM), a schema-aware native XML database solution originally developed for the management of MPEG-7 media descriptions. The core of PTDOM is made up of a schema catalog capable of managing schema definitions written in MPEG-7 DDL, a superset of XML Schema. PTDOM exploits the schema information maintained by this catalog in a variety of contexts: for document validation, for typed storage of elements and attribute values, for structural indexing of XML documents providing additional efficient access paths to document contents, and for optimized construction of query execution plans for XPath expressions. This along with its profound extensibility with new datatypes, user-defined functions, and value index structures makes PTDOM a flexible and effective database solution not only for the management of MPEG-7 media descriptions but also for the management of any kind of XML documents for which schema definitions exist.

1 Introduction

Although the structure of XML documents is to some extent self-describing by means of their markup, it has long been recognized that additional structure and type information in form of schema definitions – provided that such definitions are available – can considerably contribute to an effective management of larger numbers of XML documents in a database [2, 15, 65]. In consideration of this fact, it is remarkable that the plenitude of XML database solutions existing today [3, 63, 61] – commercial systems, research prototypes, just as open-source projects; native XML database solutions or XML database extensions just as traditional database management systems (DBMSs) – largely neglect available schema definitions for XML document management. Many XML database solutions completely ignore them; and those solutions that are capable of processing some form of schema definitions – be they Document Type Definitions (DTDs) [4] or, becoming increasingly prevalent, XML Schemas [58, 1] – mainly restrict their use to document validation as a means of ensuring database consistency.

But the structure and type information contained in schema definitions can be exploited for an effective management of XML documents in more ways than mere document validation:

- Modern schema definition languages such as XML Schema provide rich sets of simple types which can be used to precisely specify the nature of element or attribute value content within a schema definition, covering not only strings but also various kinds of non-textual types like numbers, date and time values, and even binary data. An XML database solution can exploit such type information to derive the types of basic document contents and store these contents appropriately [60, 62]. Compared to the alternative of simply treating all these contents as text regardless of their type, such a typed storage allows applications to access and process non-textual content in an adequate and efficient way and also permits an appropriate indexing of this content.

- The information about the allowable structure of XML documents contained within schema definitions permits sophisticated query optimizations promising faster query evaluation times [2, 15, 65]. Queries can be pruned to simplify evaluation by identifying subexpressions that – according to a schema definition – can never yield a result or are redundant. Based on a schema definition, queries can also be equivalently rewritten so

that their evaluation can profit from potentially existing indexes.

- A schema definition itself can serve as an effective path index for the XML documents complying to that definition [21, 15, 45]. If a database solution interconnects the element types and attributes declared in a schema definition with the elements and attribute values which instantiate them inside the XML documents contained in a database, large portions of path traversals in queries can be evaluated directly on top of the schema definition without the need of touching and traversing a potentially large number of documents again promising reduced query evaluation times.

In this paper, we give a system overview of the Persistent Typed DOM (PTDOM), a native XML database solution we have developed in need for an adequate, generic database solution for the management of MPEG-7 media descriptions [40, 39, 26]. A schema catalog capable of managing schema definitions indited in MPEG-7 DDL [27], a superset of XML Schema, forms the heart of PTDOM. Aware of the benefits to be gained from schema information, PTDOM makes use of the schema definitions maintained by this catalog in a variety of contexts. Not only are these definitions employed to validate XML documents during document import and after updates in order to ensure database consistency as well as to infer and construct appropriate typed representations of the basic contents of XML documents in order to give applications adequate access to even non-textual data. Also, PTDOM's schema catalog acts as a highly effective path index that associates the element types and attributes of a schema definition with those elements and attribute values which validly instantiate them within the documents complying to that definition. Finally, PTDOM implements a specialized query algebra for the evaluation of XPath expressions [8] which is tailored to the exploitation of the schema catalog's path indexing capabilities, the typed representation of basic document contents, as well as PTDOM's extensive value indexing support. Though not yet providing a dedicated query optimizer, PTDOM features an optimized translator which transforms XPath expressions to the query algebra employing heuristics that base on the information provided by the schema catalog.

Such an extensive schema-awareness in combination with a profound extensibility comparable to modern object-relational DBMSs permitting the seamless integration of arbitrary new

simple types, user defined functions, as well as value index structures into the system makes PTDOM a flexible and effective XML database solution – not only for the management of MPEG-7 media descriptions but also for the management any kinds of XML documents for which schema definitions written in XML Schema are available.

The remainder of the paper is organized as follows: Section 2 illustrates and elaborates the manifold benefits of considering and exploiting available schema definitions for the management of XML documents. Section 3 analyzes whether or to what extent current XML database solutions make use of schema definitions for XML document management fortifying our claim that available schema information is largely lying idle today. Section 4 gives an overview of the components of the PTDOM system. Section 5 presents some performance evaluations and experimental results. Section 6 concludes the paper and gives an outlook to current and future work.

2 Applications of schema definitions for XML document management

In this section, we discuss the various applications of available schema definitions and illustrate their benefits for the management of XML documents in a database in more detail. For the discussion, we make use of a motivating example schema definition from the domain of metadata management for digital media which constitutes the background of our research work.

Figure 1 provides this example definition. It gives a slightly simplified excerpt of the so-called `Melody` media description scheme [29] as defined by the MPEG-7 standard [40, 39, 26]. MPEG-7 is an ISO standardization effort aiming at establishing a common metadata framework for the extensive description of multimedia content at different levels and from different perspectives that is of use for a broad spectrum of applications, including multimedia archives, search engines, media production support, education and entertainment.

Not only for the purpose of defining the plenitude of ready-to-use media description schemes shipping with the standard [28, 29, 30] but also for giving applications means to create their own or extend existing description schemes, MPEG-7 provides the Description Definition Language (MPEG-7 DDL) [27]. MPEG-7 DDL is a general-purpose schema definition language for XML

```
…

<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 1: Example of an XML schema definition

documents that constitutes a superset of XML Schema, mainly extending XML Schema with support for matrix types, vector types, and some additional temporal types to better cope with the peculiarities of multimedia content description.[1] As a consequence, an MPEG-7 media description scheme is nothing else than a schema definition for XML documents; a conforming MPEG-7 media description is nothing else than a document valid to that definition.

Briefly explained, the schema definition of Figure 1 serves to describe a song's melody and can be used as a basis for the realization of, e.g., query-by-humming applications. The entry

[1]Note that the example media description scheme depicted by Figure 1 does not make use of any of MPEG-7 DDL's extensions. Thus, it is perfectly valid XML Schema as well.

point to the schema definition is given by the complex type `MelodyType` to the upper left of the figure. By extending the predefined type `AudioDSType`, it is expressed that `MelodyType` actually defines an MPEG-7 media description scheme for the description of audio content. 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 defining the permissible content for `Meter` elements, is a fraction consisting of numerator and denominator (element types `Numerator` and `Denominator`). 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 allowable content for `MelodyContour` elements. 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 which can serve as the root element type for any description of audio content. The content valid for elements of that type is given by the complex type `AudioDSType`. Since every MPEG-7 audio description scheme is ultimately derived from `AudioDSType`, `AudioDescriptionScheme` elements can hold descriptions complying to any audio description scheme as long as the `xsi:type` attribute predefined by XML Schema is employed to declare the particular subtype of `AudioDSType` used, such as `MelodyType`.

Figure 2 shows an example XML document (taken from [29], page 101) that constitutes a conforming MPEG-7 `Melody` media description valid to the schema definition of Figure 1. The document describes a small fraction of the melody of the song "Moon River" by Henry Mancini. An `AudioDescriptionScheme` element constitutes the entry point to the description whose content is marked to be compliant to the complex type `MelodyType` by means of an `xsi:type` attribute value.

Having an example schema definition at hand, we now proceed with the illustration of the various uses and benefits of schema definitions for XML document management:

```
<!-- 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: Example of a complying XML document

1. Document validation.

A central task of a DBMS is to ensure the consistency of database contents. Similarly, it should be in the area of an XML database solution's responsibility to warrant the consistency of the XML documents stored with it. Since XML schema definitions like our example MPEG-7 `Melody` media description scheme precisely specify the available element types and attributes for a family of XML documents as well as their permissible content, the utilization of schema definitions is indispensible for an XML database solution to serve this responsibility. On the one hand, an XML database solution can utilize available schema definitions to prevent that inconsistent documents are inserted into a database. During the import of an XML document into the database, the solution can validate that the document does not offend its schema definition. The validation of XML documents has been heavily investigated in literature [6, 43, 49, 35, 33] and is generally well-understood. Moreover, the ability to validate XML documents not only against DTDs but also against schema definitions indited in more expressive languages such as XML Schema already comes for free with most modern XML parsers.

On the other hand, an XML database solution can exploit available schema definitions to ensure that documents already contained in a database cannot be brought to an inconsistent state. During an update of an XML document, the solution can validate that the update operations do not lead to a violation of the document's schema definition. While this can be achieved by simply revalidating the whole document after the update again as during document import, research has recently started to think about more efficient incremental approaches [62, 44] that try to limit the revalidation effort to just those document parts that have been affected by the update. For example, it would be indeed ineffective to validate the whole document of Figure 2 against the schema definition of Figure 1 again, just because an update has validly changed the list of integer values making up the content of the `Contour` element by appending further numbers: if the document was known to be valid before the update, it would be sufficient to check that the affected element still complies to the specification of the element type `Contour` in the schema definition.

An XML database solution that does not make use of existing schema definitions for XML document validation, in constrast, defers the responsibility of maintaining database consistency to the applications working with that solution. Apart from that this can be cumbersome for applications, leaving the valuable good of a consistent database state in the hands of application is dangerous and error-prone.

2. Document typing.

XML, although originally intended as a basis for cross-media publishing allowing the development of presentation-neutral structured document formats, has also become a widely adopted foundation for exchange and file formats for non-document application data. A typical characteristic of XML formats carrying non-document application data is that they consist to considerable extents of non-textual data, like various kinds of numbers, time values, more complex structures such as vectors, lists, and matrices, or even binary objects. The schema definition of Figure 1 providing the MPEG-7 `Melody` media description scheme of is a prime example of such a format as it consists solely of non-textual data: the meter of a melody is defined by integer values, its beat and contour by lists of integer values. As a matter of fact, more than 80% of the schema definitions offered by the MPEG-7 standard for the description of visual

and audio content in [28] and [29] cover primarily non-textual data.

But since XML is a text document format, all data within XML documents – be they textual or not – are self-evidently encoded as text. While this might be appropriate for the platform-independent exchange of XML documents, the textual representation of non-textual data is certainly inadequate for the storage of XML documents within a database: textual representations of non-textual data often not only consume more storage space than corresponding binary representations; typically, they are also less efficient. It is rather obvious, for example, that keeping the list of integer values making up the content of the `Contour` element in Figure 2 in the depicted textual representation, i.e., as a string, is inefficient compared to keeping the list in a more adequate data structure such as an array. Furthermore, having to process non-textual data like this list of integer values by means of string operations is not at all adequate. As a consequence, applications will typically have to manually transform textually represented non-textual data to more adequate data structures during each access – either by using self-written string conversion routines, by performing explicit type casts within queries, or by exploiting implicit type coercion rules of the operators of a query language – so that a more appropriate processing is possible. This is cumbersome, error-prone, and time-consuming. Finally, the textual representation of non-textual information is not always adequate to the semantics of the data. For example, the alphanumeric order of the textual representation of integer values differs from their inherent numeric order. This complicates, for example, meaningful indexing.

It would thus be of great benefit if an XML database solution stored and represented basic document contents, i.e., simple element content and the content of attribute values, in an appropriately typed manner. Typed representation means that these contents are kept in data structures appropriate to the particular content type and that a rich set of type-specific operators (e.g., as given by [38]) is available for their appropriate processing – just as it is the case for the contents of table columns and object properties within traditional relational and object-oriented DBMS.

But even though they convey rich structural information by means of their markup, XML documents do not carry any type information concerning their basic contents that could be used by a database solution for the construction of appropriate typed representations; this information is contained within the schema definitions to which the documents comply. Without

the schema definition of Figure 1, for instance, there is no indication for an XML database solution that the content of our example `Contour` element consitutes a list of integer values and not just an arbitrary text that happens to comply the textual representation of this list by chance. Hence, in order to be able to provide the benefits of typed representation and type-adequate processing of simple element content and the content of attribute values, it is indispensible for an XML database solution to draw on available schema definitions for the management of XML documents and use the type information contained therein to infer the types of these basic contents.

3. Query optimization.

It is widely accepted that a DBMS should possess a query optimizer that is capable of automatically rewriting queries issued by an application into an equivalent form so that they can be evaluated efficiently. Apart from the pure comfort of such a query optimizer that relieves application developers from the burden of constantly having to consider the structure of database contents and their physical storage (e.g., structural constraints, defined indexes, etc.) for the formulation of efficient queries, a query optimizer also to some extent protects applications from subsequent changes to the structure and physical storage of database contents. Without a query optimizer, changes to the indexes defined within a database can quickly leave an application's queries inefficient raising the need to modify the application to reformulate its queries.

The availability of a query optimizer is of course desirable for an XML database solution as well. In this regard, it has been recognized quite early [2, 15, 65] that schema definitions constitute a valuable information source and provide a precise and rich decision base for query optimization within XML database solutions as they give accurrate summaries of the permissible structure and contents of XML documents. Without this information, the means available for query optimization are limited: a query optimizer can at best rewrite queries according to rewriting rules that are universally valid for any XML document complying to any potential schema definition. Furthermore, it can base the decision to apply these rewriting rules solely on heuristics or statistical information that might be gathered by a database solution.

An XML database solution that takes schema definitions into account for the management of XML documents, in contrast, has the benefit of having more sophisticated options for query

optimization at its disposal: one use of schema definitions for query optimization is to reduce the complexity of query evaluation. For instance, the evaluation of subexpressions of a query can be avoided if, according to the schema definition to which the documents of a database comply, these are certain to never yield a result. An admittedly simple but nevertheless striking example of this is point the evaluation of the XPath expression `//Composer` – which retrieves all `Composer` elements contained in an XML document – on a database full of conformant MPEG-7 `Melody` descriptions. With knowledge of the schema definition of Figure 1, an XML database solution can conclude without touching any document that the expression can never deliver a result because no element type `Composer` is declared in the schema definition and every document containg an element of that type would be invalid; without this knowledge, the solution instead is forced to access the potentially large number of documents in the database and traverse all of their elements on the fruitless search for `Composer` elements.

The complexity of query evaluation can also be reduced by employing schema definitions to identify and prune redundant subexpressions. For example, instead of evaluating the XPath expression `//Meter[Denominator/data() <= 128]/Numerator` on the database with MPEG-7 `Melody` descriptions on the search for all numerators of meters with a denominator less than or equal to 128, it is perfectly sufficient to evaluate the simpler expression `//Meter/Numerator`: with knowledge of the schema definition of Figure 1, it can be inferred that the denominator of a meter never exceeds 128 and therefore checking the condition contained in the original expression is unnecessary.

Another use schema definitions for query optimization is to equivalently rewrite queries into a form in which they make use of operands that are cheaper to evaluate or existing indexes can be exploited. For example, a notoriously expensive operand of the XPath language is the `//` operand which traverses all direct and indirect child elements of an element. When performing a search for all numerators of meters within the MPEG-7 `Melody` descriptions of our example database by means of the XPath expression `//Numerator`, an XML database not considering schema definitions is obliged to access each of the database's documents and traverse all of its elements because there might be a `Numerator` element lurking somewhere in the depths of a document. With knowledge of the schema definition of Figure 1, however, an XML database solution can equivalently rewrite the expression to `/AudiodescriptionScheme/Meter/Numerator`

because, according to that definition, `Numerator` elements are only permissible as direct children of `Meter` below `AudioDescriptionScheme` elements, the latter always forming the root elements of conformant documents. The rewritten expression is obviously cheaper to evaluate: though still each of the database's documents has to be accessed, traversal is explicitly directed into the meter part of `Melody` descriptions stopping there.

Furthermore assuming that a path index that precomputes and maintains all elements reachable by the XPath expression `/AudioDescriptionScheme/Meter` is defined on the example database, rewriting the expression `//Numerator` to `/AudiodescriptionScheme/Meter/Numerator` has additional benefits: it is now obvious to the query processor that this path index can be applied for query evaluation because the expression for which the index is maintained constitutes a prefix of the rewritten expression. Thereby, not only the traversal down to the `Meter` elements within the documents which are accessed during evaluation is spared because these elements can be directly obtained from the path index; also the overall number of documents accessed during evaluation can be significantly reduced considering that the occurrence of `Meter` elements within MPEG-7 `Melody` descriptions is optional and the use of the path index avoids access to documents containing no `Meter` elements at all.

4. Path indexing.

Apart from the optimization of queries, an XML database solution can utilize available schema definitions for accelerating query evaluation in a further way: if a database solution's schema catalog not only manages the various element types and attributes that are declared within schema definitions but also interlinks these declarations with those elements and attribute values which instantiate them inside a database's documents, schema definitions can serve as formidable path indexes [21, 15, 45]. With such an indexing at hand, a query's path traversals can be evaluated to large extents on top of comparatively small schema definitions; the potentially large number of potentially large XML documents contained in a database only need to be touched during the last phase of query processing when the element types and attributes whose instantiating elements and attribute values potentially qualify as query results have been localized as precisely as possible.

Schema definition in schema catalog:

Instantiating documents in database:

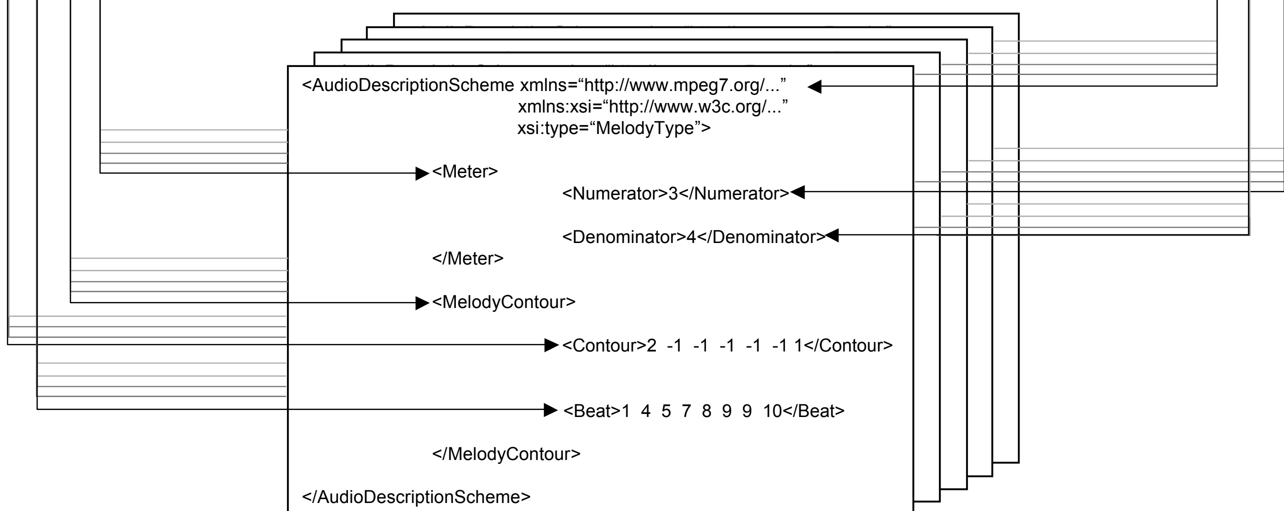

Figure 3: Employing a schema definition as a path index

Consider, for instance, once more the evaluation of the XPath expression `//Meter` searching for all meters within the example database full of MPEG-7 media descriptions complying to the schema definition of Figure 1. Without path indexing support, an XML database solution will

have to wade through every document within the database for the evaluation of that expression and traverse each contained element on the search for query results, if we assume that no query optimization takes places. However if, as illustrated by Figure 3, the database solution's schema catalog maintains for each of the element types declared within the example schema definition (labeled `et1-et7` in the figure) by which elements it is instantiated in the documents of the database (as indicated by arrows), the same query can be evaluated more efficiently, especially on a large database: the query processor just has to look up all element types bearing the name `Meter` within the schema definition – there could be more than one because MPEG-7 DDL just like XML Schema permits the declaration of homonymous element types within the scopes of different complex types. In our example, the only qualifying type is the one labeled `et3`. The query processor can then directly obtain the query result without the need of any further access to the documents of the database by simply returning the instantiating elements of `et3` maintained by the schema catalog.

The proposed path indexing functionality for schema catalogs is also useful in cases of more complex path traversals which cannot be fully evaluated on top of the schema definition. An example of this is the XPath expression `/AudioDescriptionScheme/MelodyContour/Beat` querying all beats of melody contours within an audio description. A reasonable strategie for a database solution's query processor that exploits the path indexing functionality of the schema catalog could be to first look up all element types with the name `Beat` in the schema definition, which would yield the element type labeled `et2` in our example of Figure 3. Although one might be tempted to think that the result of the query could now be immediately obtained from the schema catalog by returning the elements that instantiate `et2`, this is not the case however: as the complex type `MelodyContourType` in which `et2` is defined is derived from the complex type `AudioDSType`, a `Beat` element may also occur as a direct child of an `AudioDescriptionScheme` element provided that element carries a corresponding `xsi:type` attribute value set to `MelodyContourType`. Therefore, there would be the need for an additional processing phase in that selects only those `Beat` elements which occur as direct children of `MelodyContour` elements.

Despite the additional filtering required, the proposed evaluation strategy is nevertheless likely to result in an accelerated query evaluation: one has to consider that the alternative is

to access every document in the database and traverse down from the document's root to the `Beat` element which might actually fail since `MelodyContour` elements are declared optional. In contrast, the elements instantiating `et2` already constitute a good approximation of the query result which can be obtained quickly and leaves out already those documents that do not contain any `Beat` elements at all.

3 Analysis of schema support and utilization

Having illustrated the various uses and benefits schema definitions have for the management of XML documents in a database, we now take a brief look at state-of-the-art XML database solutions including research prototypes, commercial products, as well as open-source projects and examine whether these solutions consider available schema definitions and how they make use of them. For that purpose, the table of Figure 4 presents an extract from an extensive survey we have conducted recently investigating the suitability of XML database solutions for the management of MPEG-7 media descriptions that focuses on this question [63, 61].

In particular, the table summarizes for a representative set of prominent XML database solutions categorized into native XML database solutions and XML database extensions for traditional DBMS whether these solutions feature a schema catalog for the management of the schema definitions to which a database's documents comply, which schema definition languages are supported by that catalog, and how these definitions are utilized: for document validation, document typing, query optimization, or path indexing. In close accordance with definitions like [68], we regard as the major distinctive feature between native XML database solutions and XML database extensions that a native solution permits the modeling of data only by means of XML documents whereas a database extension still offers the modeling primitives of the extended DBMS – thus, it is possible to implement a native XML database on top of a traditional DBMS as long as the data model of the backend is entirely hidden from applications.

The table fortifies our claim that existing XML database solutions – be they native solutions or database extensions, be they commercial products, open-source projects, or research prototypes – fall short of exploiting available schema definitions for an improved management of XML documents although the applications and benefits of schema definitions in this regard

		Schema catalog		Utilization of schema definitions in catalog			
		available	Supported schema definition languages	Document validation	Document typing	Query optimization	Path indexing
Native database solutions	eXcelon XIS	■	DTD XML Schema	■	–	–	–
	GoXML DB	■	DTD limited XML Schema	■	–	–	–
	Infonyte-DB/ PDOM	–	–	–	–	–	–
	Tamino	■	TSD limited XML Schema	■	–	■	–
	TEXTML	–	–	–	–	–	–
	X-Hive/DB	■	DTD	■	–	–	–
	Xindice/ dbXML	–	–	–	–	–	–
	eXist	–	–	–	–	–	–
	Lore	■	DataGuides	–	–	–	■
	Natix	–	–	–	–	–	–
	Timber	–	–	–	–	–	–
Database extensions	IBM DB2 XML Extender	■	DTD	■	–	–	–
	Microsoft SQLXML	–	–	–	–	–	–
	Oracle XML DB	■	DTD XML Schema	■	–	–	–
	ozone/XML	–	–	–	–	–	–
	Monet XML	–	–	–	–	–	–
	Shimura et al.	–	–	–	–	–	–
	XML Cartridge	–	–	–	–	–	–
	Oracle XML DB/ Structured Mapping	■	DTD XML Schema	■	□	□	–

Figure 4: Schema definition support and utilization in XML database solutions, (■ support, □ partial support, – no support)

are long since known. Quite a few of the examined database solutions – on the side of the native solutions namely the commercial systems TEXTML [31] and Infonyte-DB [25] evolved from the research prototype PDOM [23], the open-source projects Xindice [56] formerly known

as dbXML [55] and eXist [41], and the research prototypes Timber [32] and Natix [16, 34]; on the side of the database extensions namely the commercial Microsoft SQLXML [42], the open-source project ozone/XML [10], and the research prototypes Monet XML [48], Shimura et al. [51], and XML Cartridge [18] – do not even feature schema catalogs but completely ignore available schema definitions. These solutions rely on applications to use document validation capabilities of XML parsers prior to inserting documents into a database to ensure database consistency themselves and cannot treat basic document contents in a way appropriate to their type. Though some of these solutions provide means for query optimization and path indexing – e.g., Infonyte-DB features a signature index for path indexing and performs some universal, schema-independent query rewriting based on heuristics – they leave the considerable opportunities opened by schema definitions with regard to these issues lying fallow.

If at all considered for document management, the examined solutions mainly employ schema definitions for document validation as it is the case with the commercial native products eXcelon XIS [12], GoXML DB [67], and X-Hive/DB [66] and the commercial database extensions IBM DB2 XML Extender [24] and Oracle XML DB [22]. Nevertheless, the central role the management and utilization of available schema definitions plays for an adequate and effective management of XML documents remains unrecognized by these systems as well: the attractive opportunities offered by schema definitions for document typing, query optimization, and path indexing remain unexploited.

Three of the examined solutions – notably Lore [20], Tamino [54], and the Oracle XML DB with its Structured Mapping storage option – concede schema definitions a more important role for XML document management than mere document validation. Lore automatically constructs so-called DataGuides as schematic, graph-based summaries of the XML documents contained in a database. These DataGuides serve path indexing purposes: a node of a DataGuide which represents a class of common elements or attribute values maintains references to these elements or attribute values allowing to evaluate significant amounts of path traversals on top of DataGuides as described in Section 2. However, DataGuides are neither used for document validation and typing nor for query optimization.

Tamino maintains a schema catalog based on the proprietary schema definition language TSD [53] capable of expressing a limited subset of XML Schema. Though Tamino does not

exploit the schema definitions contained in this catalog for document typing and path indexing, they are at least not just utilized for document validation but also for an advanced query optimization. Before evaluating a query written in the proprietary XPath-like query language X-Query [52] against documents that comply to a given schema definition, the query is automatically rewritten into a canonical form. In this form, subexpressions contained in the query making use of wildcards and `//` traversals to direct and indirect child nodes are replaced by the disjunction of all those paths that qualify for these expressions according to the schema definition. Thus, the canonical form frequently relieves Tamino's query processor from the need of performing expensive full-document traversals in face of `//` operators and further helps the query processor in deciding whether existing indexes can be employed for query evaluation.

When employing the Structured Mapping option for storing XML documents, the Oracle XML DB database extension – comparable in approach to [50, 59, 9] – creates a dedicated relational database schema for every schema definition to which the documents of a database comply. After validation, document contents are fine-grainedly mapped to these relational schemas during document import. Since elements with simple content and attribute values are mapped to fields with SQL data types coming closest to the simple types definining their content in the respective schema definitions, one can say that Oracle XML DB/Structured Mapping exploits available schema definitions for document typing. Since a query processor is provided that rewrites XPath expressions to equivalent SQL statements which are then evaluated on top of the relational schemas specifically created for the schema definitions and which are subject to optimization by Oracle's relational query optimizer, one can also say that Oracle XML DB/Structured Mapping employs available schema definitions for query optimization.

Closer inspection reveals, however, that this support for document typing and query optimization has considerable limitations. The mapping scheme employed by Oracle XML DB/Structured Mapping does not support more complicated simple types like lists and matrices but simply keeps such content as text. Moreover, the mapping scheme cannot reasonably deal with elements of mixed and arbitrary content as well as with elements that make use of the `xsi:type` attribute: the content of such elements is simply packed unstructuredly into textual overflow stores. As an example, almost the whole MPEG-7 media description of Figure 2 would be assigned to such a textual overflow store because the root element makes use of the

`xsi:type` attribute already. Finally, the rewriting of XPath expressions to SQL statements is not possible when document contents are touched that are kept in textual overflow stores and only works for a very limited subset of XPath. When encountering these restrictions, the XPath query processor costly reassembles each document on which an XPath expression is evaluated into its original textual format and then evaluates the expression in main memory.

Summarizing, one must recognize that even with more schema-aware XML database solutions like Lore, Tamino, and Oracle XML DB/Structured Mapping the use of schema definitions still remains selective and limited. We therefore definitely see a demand for new XML database solutions that tightly incorporate the management of available schema definitions as a central concept and extensively exploit them for the various aspects of XML document management.

4 The Persistent Typed DOM

In the light of the deficiencies of existing XML database solutions with regard to the utilization of available schema definitions, we have developed the Persistent Typed DOM (PTDOM) – a schema-aware native XML database solution that consequently and extensively uses available schema definitions throughout the various facets of XML document management.

In the following, we present the object-oriented architecture of PTDOM whose primary components and their interdependencies are identified by the UML component diagram of Figure 5. The design of this architecture and its components is guided by several objectives:

- *Extensive schema utilization.* Section 2 has shown the high value schema definitions have for XML document management: for ensuring database consistency by document validation, for enabling adequate access to basic document contents by document typing, for improving query performance by path indexing, and as a decision base for query optimization. As a basis for the extensive exploitation of available schema definitions for the mentioned purposes, a schema catalog forms the core of the PTDOM architecture.

- *Fine-grained document management.* To allow applications to reasonably deal with even larger XML documents stored in a database, it proves helpful if an XML database solution permits access to and updates of XML documents as well parts of XML documents at any

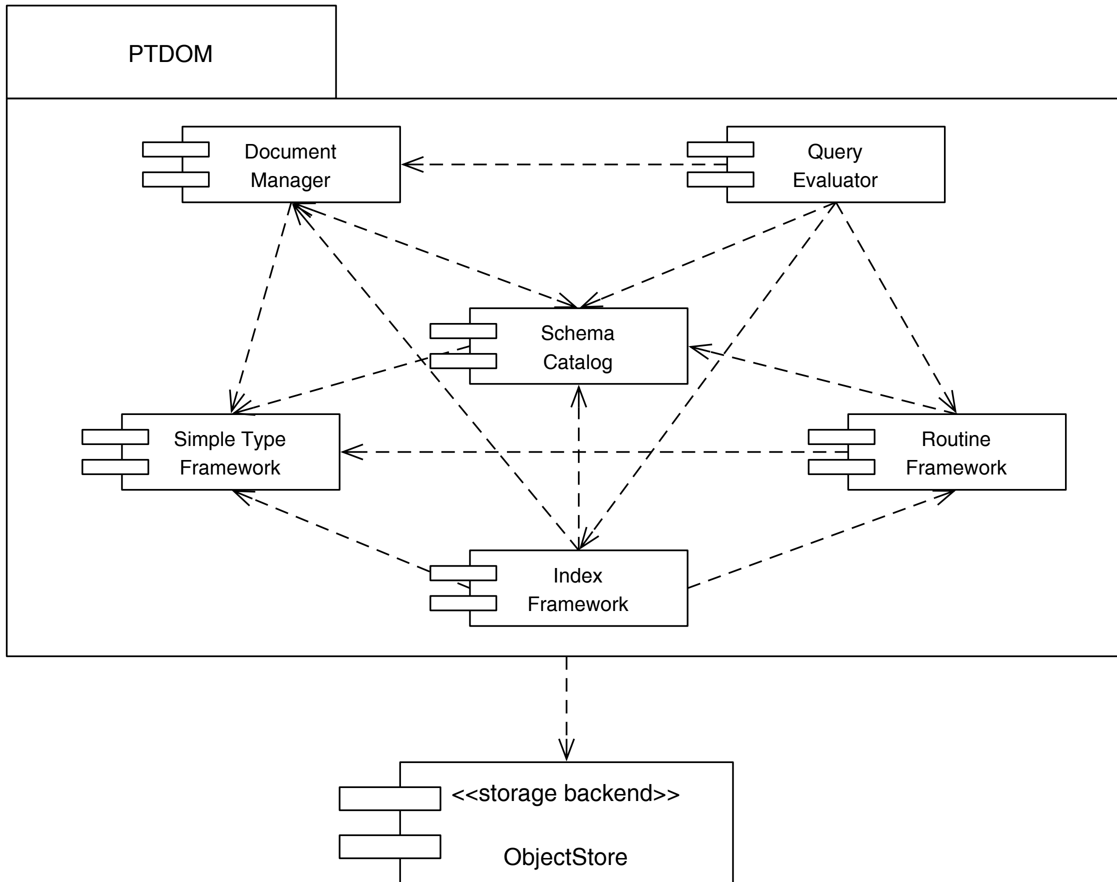

Figure 5: PTDOM architecture (UML component diagram)

level of granularity desired by an application. Therefore, PTDOM's document manager stores and represents XML documents in a fine-grained manner.

- *Typed document management.* Non-textual application data carried by XML document should be stored and represented in a way that is appropriate to the content type declared in the document's schema definition. In combination with the simple type framework and routine framework, the document manager provides typed representations of simple element content and attribute values along with type-adequate routines for an efficient and appropriate representation and handling of even non-textual document contents.

- *Rich indexing capabilities.* For an efficient handling of larger collections of XML documents, rich indexing facilities are indispensible. The architecture addresses this issue not only by employing the schema definitions within the schema catalog as path indexes.

Also, the index framework provides a broad array of value index structures for indexing unordered, ordered, and even multidimensional document contents.

- *Efficient querying.* A high query performance is a natural goal of any database solution. The architecture provides an XPath query evaluator implementing a specialized query algebra that is tailored to the exploitation of PTDOM's advanced features: the use of schema definitions as path indexes, the rich set of value index structures available with PTDOM, and the typed representation of basic document contents.

- *Extensibility.* Profound extensibility is a virtue for any database solution as it allows to flexibly meet individual needs of applications. Similar to modern object-relational DBMSs, the simple type, routine, and index frameworks permit to seamlessly integrate new simple types, user-defined routines, as well new value index structures into PTDOM.

- *Classic DBMS functionality.* Transaction support, fine-grained concurrency and access control, reliable backup and recovery are classic DBMS functionality that are nowadays taken for granted and which we do not want to neglect with PTDOM either. But not to be bothered too much with these complex issues and to permit a rapid prototyping of the object-oriented PTDOM architecture, we have decided to employ the object-oriented DBMS ObjectStore [11] as a storage backend inheriting this system's mature implementations of classic DBMS functionality. Our prototype even gains flexibility as there exists the PSEPro [13] small-scale in-process variant of ObjectStore making the prototype configurable as both a server-based as well as an in-process database solution. In future, we want to replace ObjectStore with a dedicated XML storage manager, like the ones proposed by [34, 16] or [23].[2]

[2]Concerning potential objections with regard to the "nativity" of PTDOM when realized on top of ObjectStore, one should bear in mind that, according to Section 3, we regard PTDOM 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 [5] which constitutes the storage backend of the native research prototype Timber [32].

In the following subsections, we discuss the individual components of the PTDOM architecture and how they collaborate in more detail.

4.1 Simple type framework

One of the aims of PTDOM is to keep simple element content and the content of attribute values contained in XML documents in a typed manner using data structures suiting the particular content types declared in the documents' schema definitions, so that applications can adequately and efficiently access and work with even non-textual data. The simple type framework component, which forms the bottom of the PTDOM architecture, provides PTDOM with such data structures. Not a fixed library with data structures for a predefined set of simple data types, this component instead constitutes a framework that permits to integrate support for arbitrary simple types which might be supported by a schema definition language like MPEG-7 DDL or needed for a certain application. Providing a type extensibility that is comparable to object-relational DBMSs, the simple type framework considerably contributes to the flexibility of the PTDOM architecture.

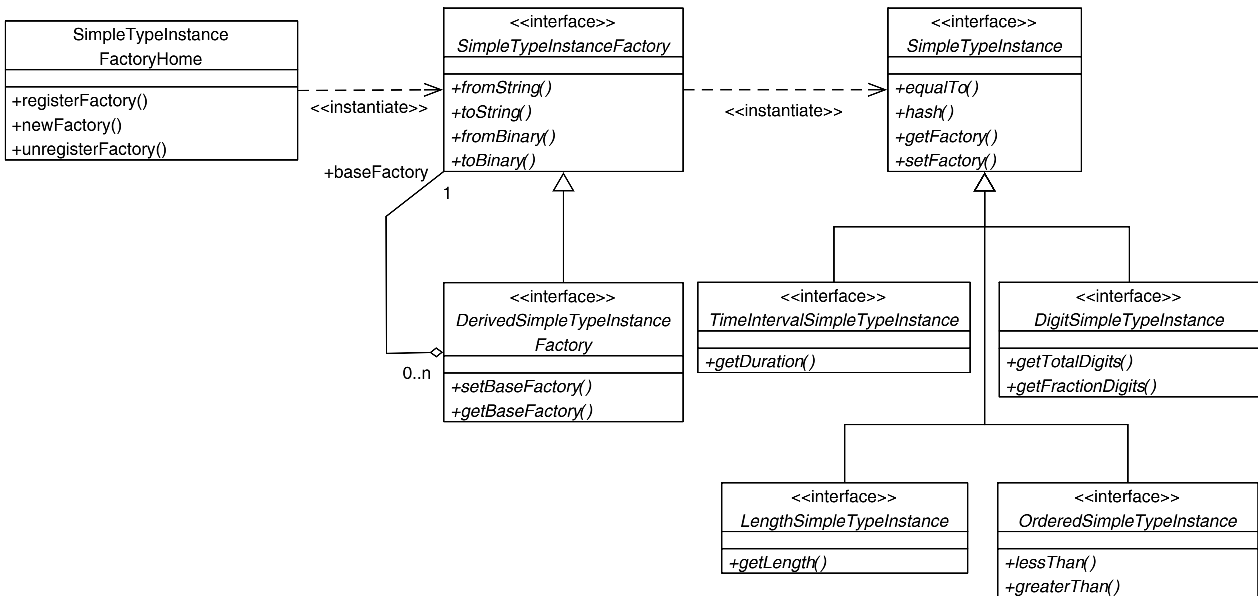

Figure 6: Simple type framework overview (UML class diagram)

The class diagram of Figure 6 presents the overall structure of the simple type framework.

In the framework, each value of a simple type is encapsulated by an object called simple type instance. This object belongs to a class that implements suitable structures for an efficient and adequate representation of the values of the given simple type. There are not much requirements for such a class: it merely has to implement at least the `SimpleTypeInstance` interface. This interface primarily enforces basic support for comparison and hashing thus ensuring a rudimentary handling and lookup of simple type instances in a database.

The simple type framework distinguishes different families of simple type instances that are represented by corresponding specializations of `SimpleTypeInstance`. These demand additional functionality from simple type instances belonging to such a family. The framework provides specialized interfaces for simple type instances that have a duration, e.g., time periods, that possess a notion of length, e.g., lists, that have digits and fraction digits, e.g., decimals, and that are ordered, e.g., integers. This choice of simple type instance families – further families can be easily integrated into the framework by introducing new specializations of `SimpleTypeInstance` – is based on the consideration which functionality has to be offered by simple type instances in order to be able to implement the various simple type derivation methods supported by an expressive schema definition language like MPEG-7 DDL. For example, to be able to implement the restriction of the element type `Numerator` to integer content from 1 to 128 in our example schema definition of Figure 1 by means of MPEG-7 DDL's simple type derivation methods `minInclusive` and `maxInclusive`, the instances of the integer simple type predefined by MPEG-7 DDL must be comparable according to notions of "less than" and "greater than", as ensured by the `OrderedSimpleTypeInstance` interface.

There must be a way to construct simple type instances out of their textual representation in which they are encoded in XML documents. This is the task of simple type instance factories, classes which implement the `SimpleTypeInstanceFactory` interface. It ensures that every such factory can construct instances of a given simple type out of their textual representation and serialize them back into that representation. Also, these factories are able to construct and deconstruct simple type instances from and to a byte array representation which, e.g., can be employed by PTDOM to store them onto disk pages. Every simple type instance keeps track of the factory which created it, as demanded by the methods `setFactory()` and `getFactory()` of the `SimpleTypeInstance` interface.

The simple type framework is not only able to cope with elementary simple types but also simple types which have been derived from other simple types within a schema definition applying simple type derivation methods offered by the language in which the definition is indited. From the perspective of the framework, the major difference between a derived and an elementary simple type is that the construction of the instances of a derived type is not self-contained but typically depends on the construction of the instances of the derived type's base type. The `DerivedSimpleTypeInstanceFactory` specialization of the `SimpleTypeInstanceFactory` interface which should be implemented by factories for instances of derived simple types makes this dependency explicit with an aggregation. Thus permitting to establish arbitrary chains of simple type instance factories, the framework can even handle simple types which are derived applying more than one simple type derivation method.

The final component of the simple type framework is the simple type instance factory home, modeled by the class of the same name in the diagram. It serves as a 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. When the method `newFactory()` is called and passed the name and namespace of the simple type of which a factory is desired, the simple type instance factory home dynamically instantiates the registered factory class.

Figure 7 exemplifies how the simple type framework can be applied in practice to integrate support for simple types and simple type derivation methods into PTDOM. The class diagram to the upper left (1) illustrates the integration of an elementary integer simple type as predefined by MPEG-7 DDL. The class `Integer` is provided for the handling of integer simple type instances along with an appropriate simple type instance factory class. As integers are ordered and have digits, `Integer` implements both the `OrderedSimpleTypeInstance` and `DigitSimpleTypeInstance` interfaces.

The diagram to the upper right (2) of Figure 7 sketches how support for the `minInclusive` and `maxInclusive` simple type derivation methods offered by MPEG-7 DDL can be accommodated. For both derivation methods, corresponding classes implementing `DerivedSimpleTypeInstanceFactory` are provided as factories for instances of simple types that have been derived via these methods. To satisfy this interface, both classes refer to the

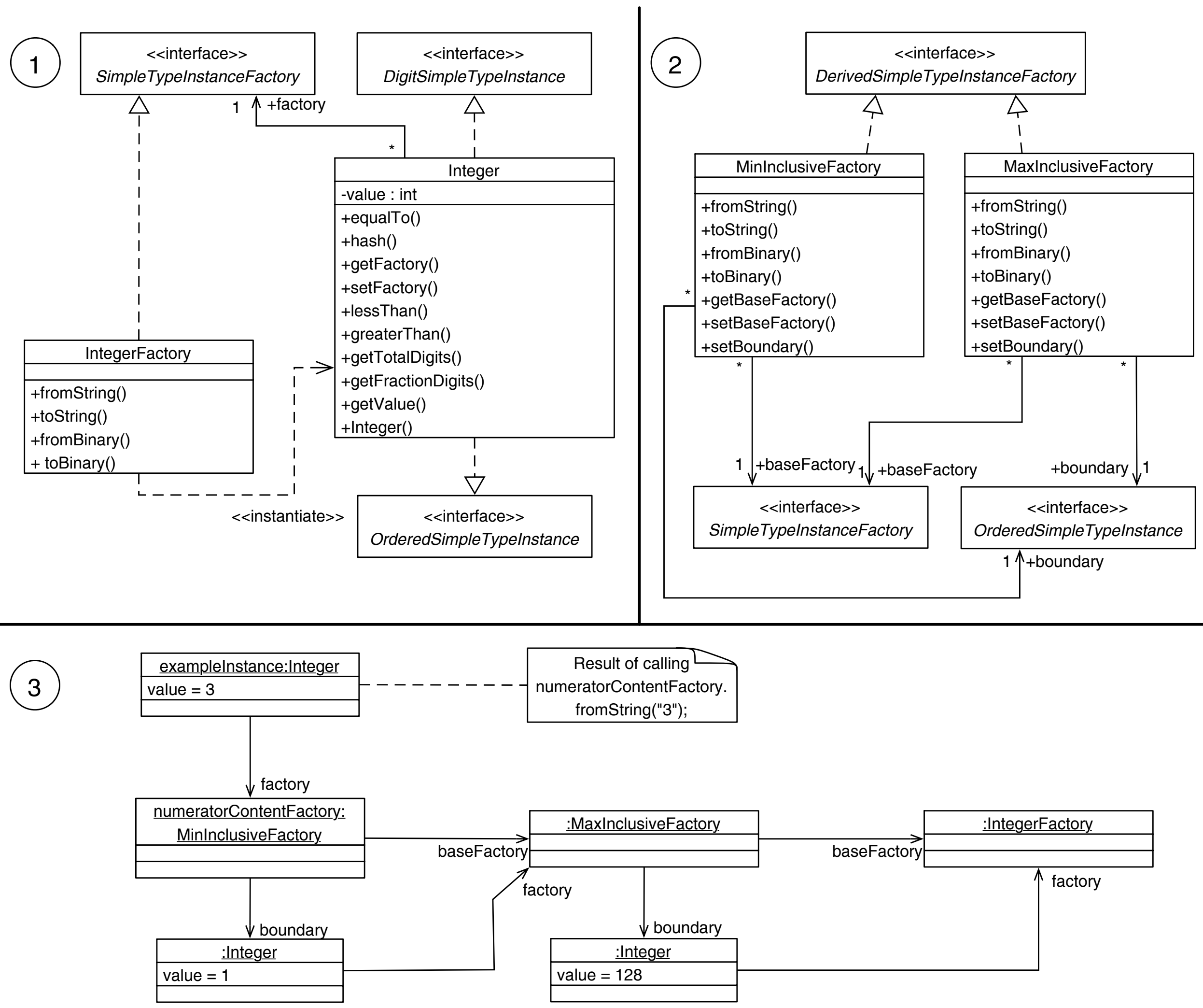

Figure 7: Applying the simple type framework (UML class/object diagrams)

factories used for the construction of the instances of the respective base types as well as to the ordered simple type instances defining the boundaries to which the domains of the base types are restricted. The implementations of the `fromString()` and `fromBinary()` methods would call the corresponding methods of the base type factories and check whether the constructed base type instances obey the specified boundaries. Calls to `toString()` and `toBinary()` would be simply delegated to the base type factories without any further processing.

The object diagram to the bottom (3) illustrates how the simple type instance factories created so far can be chained together to set up a suitable factory for the content of `Numerator` elements, closely mimicking the anonymous simple type derived from `integer` via

the `minInclusive` and `maxInclusive` derivation methods defining that content in Figure 1. The top of the diagram shows the simple type instance that would be constructed by this factory for the content of the `Numerator` element in the example document of Figure 2.

The approach outlined to integrating simple type support into PTDOM via the simple type framework can be systematically followed until all elementary simple types and simple type derivation methods of a given schema definition language like MPEG-7 DDL are covered.

4.2 Schema catalog

As pointed out before, the major aim of PTDOM is to provide a schema-aware XML database solution that extensively utilizes available schema definitions for the management of XML documents. Consequently, the heart of the PTDOM architecture is made up of a schema catalog that serves to accomodate the schema definitions to which the documents of a database comply, to ensure the integrity of these definitions, and to fine-grainedly represent the schema information contained therein as a basis for further exploitation.

The class diagram to the upper left (1) of Figure 8 gives an overview of the basic structure of PTDOM's schema catalog. The entry point to the catalog is formed by the so-called schema home modeled in the diagram by the class of the same name. The schema home acts as a container of all of a database's schema definitions represented in the diagram by the class `Schema`. Schema definitions – adressable by the URI pointing to their storage location and organizable in an inclusion lattice permitting a flexible modularization of schema definitions – in turn constitute containers of the schema components which are declared inside these definitions, such as 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 a namespace with which it can be addressed – potentially scoped as some schema definition languages such as MPEG-7 DDL allow the declaration of homonymous schema components like element types within different scopes of a single schema definition – but leaves the representational details of the different kinds of schema components to subclasses provided for these kinds.

Furthermore, `SchemaComponent` demands that its subclasses supply appropriate implementations of the abstract method `checkIntegrity()` which validate the consistency and integrity

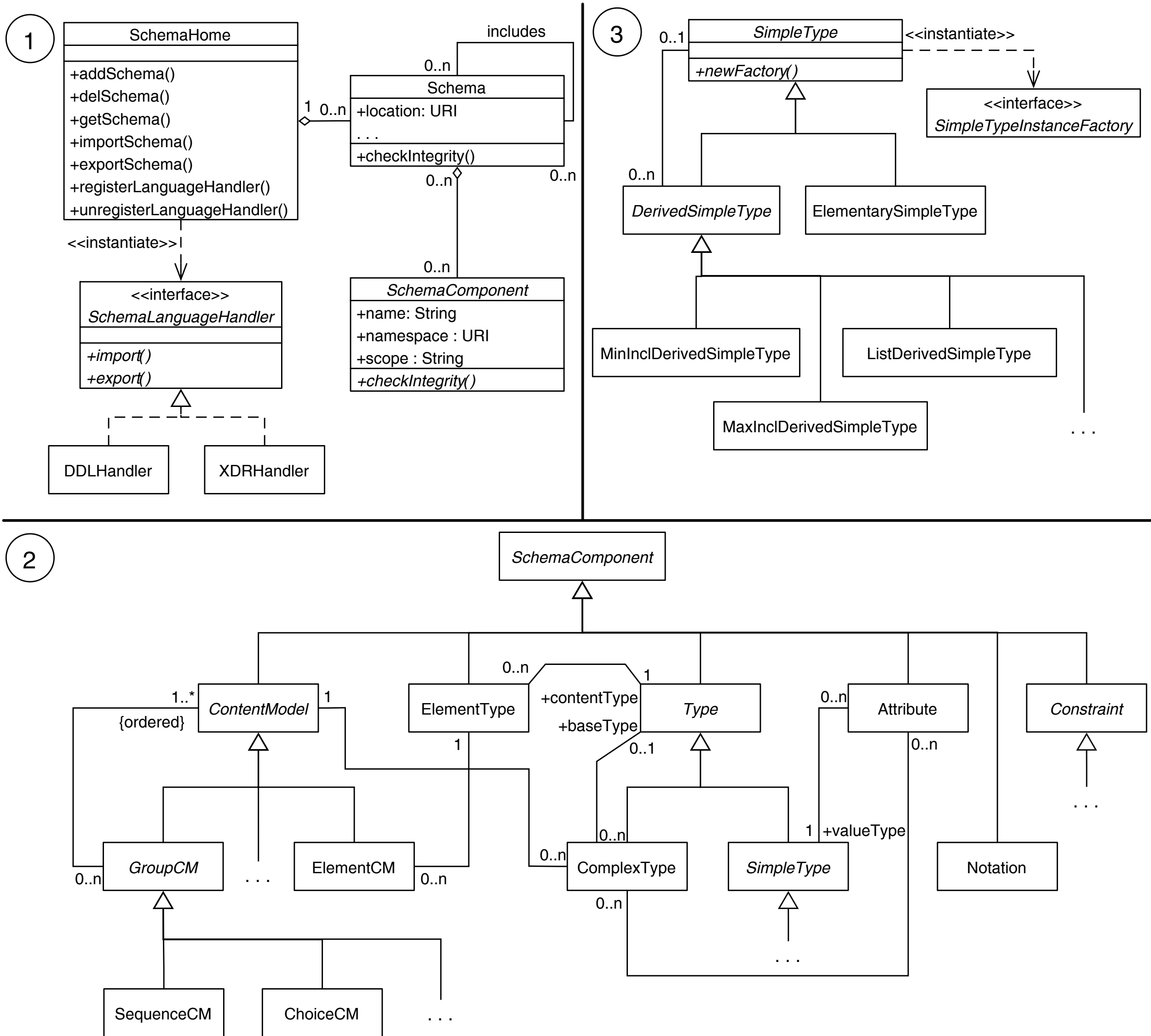

Figure 8: Schema catalog overview (UML class diagrams)

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 the schema components declared within a schema definition as it is done in the `checkIntegrity()` method of the `Schema` class.

For the import and export of schema definitions to and from the schema catalog, handlers for

schema definition languages, i.e., classes that implement the `SchemaLanguageHandler` interface, can be registered with the schema home, in principle allowing the integration of support for arbitrary schema definition languages into PTDOM again contributing to the flexibility of the system. Whenever the methods `importSchema()` or `exportSchema()` of the schema home are called in order to import or export a schema definition, the schema home looks up and dynamically instantiates the particular handler which has been registered for the desired import or export language and defers the task to the corresponding methods of that handler. Within our implementation, we have realized an MPEG-7 DDL and an XDR [17] handler.

Up to this point, we have regarded schema definitions 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 even complex schema definitions indited in a modern schema definition language; in our particular research context, this especially means the ability to cope with MPEG-7 DDL.

We have therefore decided to closely orientate the model for schema definitions used within the schema catalog of PTDOM along the XML Schema Component Data Model [58] 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 DDL schema definition. 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 [37], 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 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.

Being a central component, the schema catalog is closely connected with the other components of the PTDOM architecture. The class diagram to the upper right (3) of Figure 8 illustrates the relation of the schema catalog to the simple type framework. It details how simple type declarations are represented within the catalog's model for schema definitions with an excerpt of the subclass hierarchy below the abstract base class `SimpleType` which had been omitted in Figure 8 (2). The model 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 derived with that method.

To permit 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 to their own factory in the manner we have already illustrated before in Figure 7 (3).

The schema catalog's interplay with the other components of PTDOM will be treated in the subsections to follow when these components are introduced.

4.3 Document manager

Another essential component of the PTDOM architecture is its document manager. It constitutes a central registry for all XML documents that are stored with PTDOM and facilitates their adequate processing by applications: it faithfully captures the individual constituents of each document allowing access to and updates of its content at any desired level of granularity and – aware of and exploiting the schema information available with the schema catalog as well as the simple type framework – keeps simple element content and the content of attribute values in suitable data structures permitting type-adequate access to even non-textual content.

In close cooperation with the schema catalog, the document manager is furthermore in charge of ensuring database consistency by validating XML documents against their schema definitions whenever they are imported into the document manager or updated. The elements and attribute values within the document manager's documents are tightly coupled to the element types and attributes in the schema catalog they validly instantiate, making the catalog's schema definitions usable as path indexes to XML document content.

In the following, we first give an overview of the general structure of the document manager and the way it represents XML documents (4.3.1). We then illustrate how, on the basis of the schema definitions in the schema catalog, the document manager validates the consistency of its documents and types their contents (4.3.2). Finally, we explain how the document manager can perform document validation and typing in an improved manner after updates (4.3.3).

4.3.1 Document storage and representation

The class diagrams of Figure 9 give an overview of the document manager component. As one can see from the upper diagram (1), the basic organization of the document manager closely resembles the schema catalog. Just as the schema home acting as a container for all schema definitions stored with PTDOM forms the entry point to the schema catalog, a document home that serves as a container for all the documents stored with PTDOM forms the entry point to the document manager; just as schema language handlers can be registered with the schema home to integrate support for new schema definition languages for the import and export of schema definitions, document format handlers can be registered with the document home to integrate support for other storage formats than the traditional text format defined by the XML

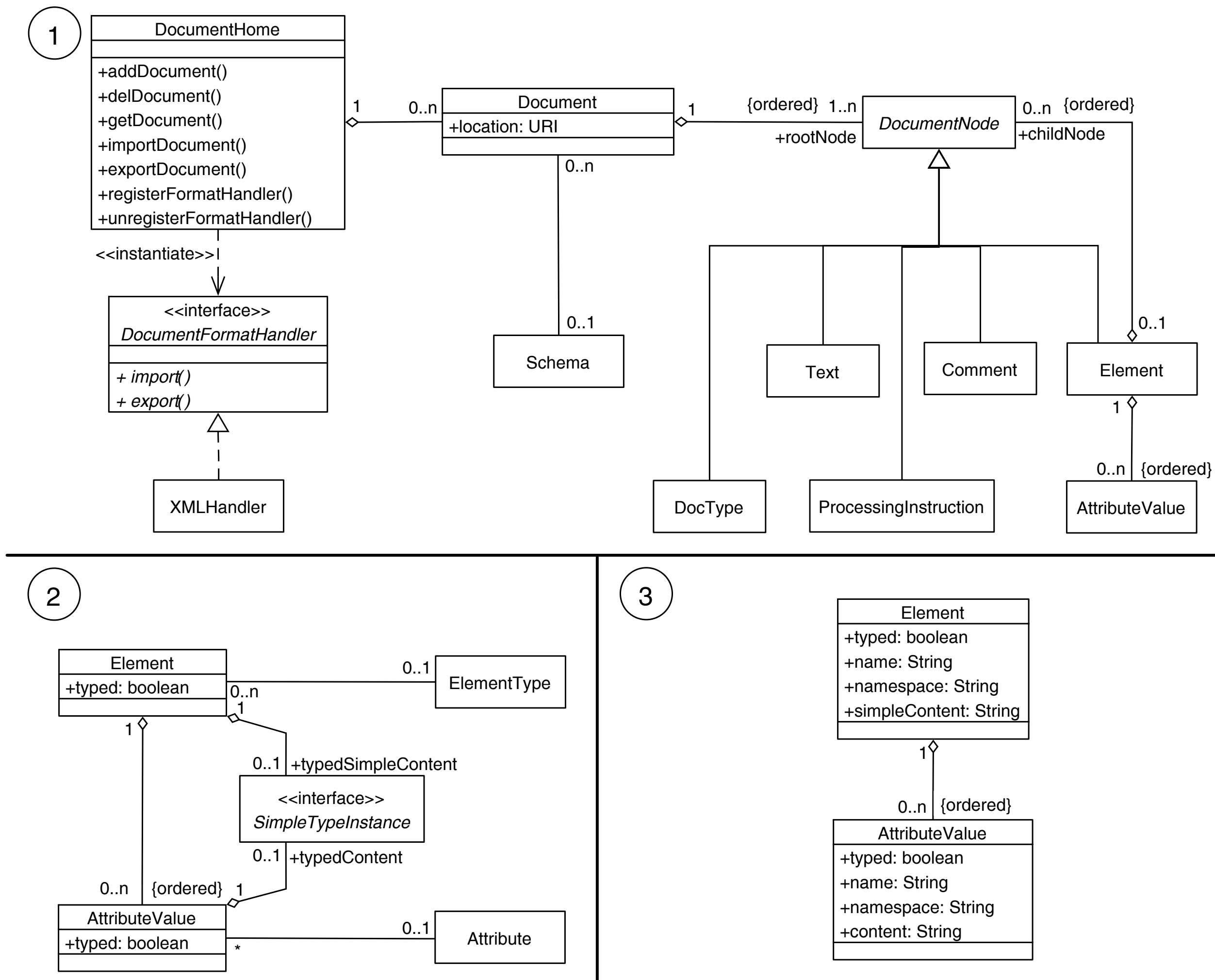

Figure 9: Document manager overview (UML class diagrams)

standard [4] with PTDOM for the import and export of XML documents, e.g., the MPEG-7 binary exchange format BiM [26].

For the representation of the contents of the documents within the document home, the document manager applies the Typed DOM (TDOM) [64, 60], an object model in the tradition of DOM [36] which fine-grainedly captures the hierarchical structure of XML documents and permits navigational access to and manipulation of their contents at any desired level of granularity. As indicated by the different subclasses of the abstract base class `DocumentNode` in the diagram, TDOM, very similar to traditional DOM, reproduces the different kinds of nodes (i.e., markups) of which an XML document consists and their interrelationships, distinguishing elements and their attribute values, comments, processing instructions, document type sections,

as well as text interspersed in mixed element content.

However, TDOM notably differs from DOM in that it tightly couples the representation of XML documents to their schema definitions. On the one hand, this is reflected by the fact that each XML document in PTDOM's document manager refers to that schema definition in the schema catalog to which it complies (if there is any), expressed in the diagram by the association between the classes `Document` and `Schema`. This association is bidirectional allowing not only to navigate from a document to its schema definition in the catalog, but also to immediately obtain all complying documents kept by the document manager for a given schema definition.

On the other hand, the tight coupling of XML documents to their schema definitions in TDOM manifestates itself in the way elements and attributes are represented. In typed representation, which is illustrated by the class diagram to the lower left (2) of Figure 9, an element is explicitly linked to the element type it instantiates, i.e., it is valid to, expressed in the diagram by the association between the classes `Element` and `ElementType`. Being again a bidirectional association, this not only opens up access to schema information by allowing to navigate from an element to the declaration of its type in the schema definition in PTDOM's schema catalog of the document carrying the element. Also, this association makes schema definitions usable as path indexes as it allows to directly obtain all elements that instantiate a given element type in one of the catalog's schema definitions in all the documents within the document manager.

If an element in typed representation has simple content, this content, as another difference to traditional DOM, is not just kept as mere text but instead as a simple type instance provided by PTDOM's simple type framework. Thereby, even non-textual element content is available in data structures that suit the particular type of the content and can be accessed and processed in a adequate fashion. Compared to this, classic DOM's textual representation of simple element content typically requires a manual transformation of non-textual content to internal data structures as an additional step before a reasonable processing can take place. TDOM also supports typed representations for attribute values. Similar to elements in typed representation, attribute values in typed representation refer to the attributes they instantiate and their content is represented by means of simple type instances.

Despite the advantages typed representation of elements and attribute values offers with regard to access to schema information, path indexing, and the representation of non-textual

content, TDOM still offers the concept of untyped representation for cases where the construction of typed representation is not possible: a document might not refer to a schema definition at all or a schema definition might make use of constructs that prohibit the determination of the element types and attributes instantiated by parts of a document's elements or attribute values. For example, the content model `any` supported by MPEG-7 DDL allows elements of arbitrary types originating from a certain namespace to appear as valid content of another element, even elements of types for which no further declarations exist in the schema definition. Moreover, it might be desirable to intentionally loosen the tight coupling of a document's elements and attribute values to the document's schema definition that exists in typed representation to be able to perform larger updates that might temporarily violate that definition.

The class diagram to the lower right (3) of Figure 9 shows the structure of elements and attribute values in untyped representation. Like traditional DOM and with all the problems involved concerning the handling of non-textual content, elements and attribute values in untyped representation are decoupled from schema definitions and carry the name and namespace of their respective element types or attributes as well as their (simple) content in string properties.

It is noteworthy that elements and attribute values in typed representation can coexist with elements and attribute values in untyped representation in a single document: just because the creation of typed representation might not be possible for a few elements or attribute values in a document, there is no reason to prevent other elements and attribute values from being represented in this more advantageous representation. The rule wich has to be obeyed here is that elements in untyped representation are not permitted to have elements in typed representation among their child elements or to have attribute values in typed representation: if the type of an element is uncertain, the types of its child elements and attribute values are uncertain as well. Given its advantages of typed representation, it is of course the intention of PTDOM make use typed representation wherever possible.

4.3.2 Document validation and typing

While typed representations offer considerable benefits, untyped representations of elements and attribute values have one indisputable advantage: they can be immediately constructed from an XML document whereas the construction of corresponding typed representations requires

the document manager's collaboration with schema catalog to know whether the document is valid to its schema definition and to which element types and attributes in particular the individual elements and attribute values of the document comply. Especially during the import of XML documents to the document manager, it is very straightforward to construct a TDOM representation of the document that keeps elements and attribute values in untyped representation only in a first processing step. But whether this document is valid to its schema definition and how corresponding typed representations for its elements and attribute values can then be constructed using the document's schema definition in the schema catalog in a further processing step are more complicated questions.

For this problem, the adoption of a common artifice in the discipline of compiler construction lies close at hand. In compiler construction, grammars are typically translated to various forms of formal automata which serve as executable intermediary representations of grammars for the purpose of parsing. Similarly, the schema definitions within PTDOM's schema catalog could be translated to an executable intermediary representation for the purpose of validating a document and inferring the element types and attributes validly instantiated by its elements and attribute values and constructing appropriate typed representations. Several such executable intermediary representations have been proposed in the literature. Proposals include approaches that translate schema definitions to XSLT stylesheets [7], to LL(1) grammars [35], and to different kinds of formal automata with varying degrees of expressiveness such as finite state automata [49], pushdown automata [49], and regular tree automata [6, 43, 46].

For PTDOM, we have developed an intermediary executable representation for schema definitions called typing automata [62] which constitute an adoption of regular tree automata seamlessly applicable to TDOM. Regular tree automata provide an intuitive executable representation of schema definitions that is able to validate XML documents and infer the element types the elements of these documents instantiate by consecutively evaluating string regular expressions – a problem which is well-understood and for which many efficient libraries are available. Going beyond regular tree automata, typing automata are also able to construct corresponding typed representations of elements in untyped representation afterwards. Moreover, typing automata have been designed to be extensible so that the basic mechanism can reach the expressiveness of MPEG-7 DDL.

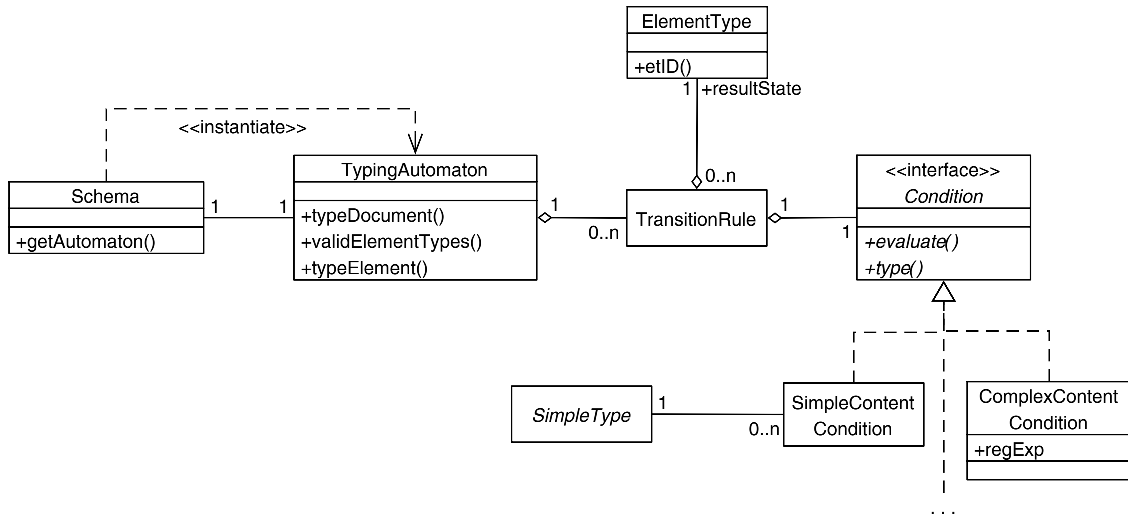

Figure 10: Typing automaton structure (UML class diagram)

The class diagram of Figure 10 presents the general structure of typing automata. As indicated by the association between the classes `Schema` and `TypingAutomata`, each schema definition in PTDOM's schema catalog caches a typing automaton representation of itself used by the document manager for document validation and the construction of corresponding typed representations. The schema catalog is able to automatically compile typing automata from its internal representation of schema definitions and maintains the consistency between schema definitions and their associated typing automata.

A typing automaton consists of a set of transition rules which represent the element type declarations of the original schema definition that is captured by the automaton. A transition rule has two parts: a result state, which is the element type of the schema definition whose declaration is represented by the rule, and a condition, which the content of an element validly instantiating this element type's declaration has to satisfy. Conditions are subsumed under the abstract interface `Condition` which demands two methods to be available with each condition: the `evaluate()` method which accepts an element in untyped representation as its parameter and determines whether the element's content satisfies the condition and the `type()` method which takes an element in untyped representation satisfying the condition as its parameter and transforms its content to typed representation in a suitable manner.

The diagram identifies the two kinds of conditions that form the backbone of the typing au-

tomaton mechanism: simple content conditions and complex content conditions represented by the corresponding classes implementing `Condition`. Simple content conditions serve to capture element type declarations which require elements to have simple content of a certain simple type. As modeled by the association between the classes `SimpleType` and `SimpleContentCondition`, a simple content condition essentially consists of a reference to this simple type.

Complex content conditions serve to represent element type declarations with a complex content model. The idea here is to capture a complex content model by means of a Perl 5 string regular expression kept within the `regExp` within the `ComplexContentCondition` class. Thereby, the problem of evaluating the content of an element against a complex content model is reduced to the problem of evaluating a string regular expression – a task easy and efficient to achieve with one of the many mature regular expression libraries available. In order to be able to express complex content models as string regular expressions, it is necessary to have unique string identifiers for the element types occurring in a complex content model; most constructs offered by a schema definition language to create a complex content model out of these element types such as sequences, choices, minimum and maximum occurrences, etc., quite naturally translate to corresponding regular expression operators. Therefore, the class `ElementType` provides the method `etID()` which mangles the name, namespace, and scope of a given element type into a unique string ID.

With the `Condition` interface abstracting from the concrete condition of a transition rule, it is possible to extend typing automata up to the expressiveness of a schema definition language by providing additional classes implementing that interface to cope with constructs of that language not coverable by simple and complex content conditions. In [62], it is illustrated how typing automata can be extended with additional conditions to cope with attribute declarations, complex type derivation, mixed content, etc., so that they become sufficiently expressive to represent MPEG-7 DDL schema definitions.

The object diagram of Figure 11 demonstrates how the example schema definition of Figure 1 can be translated to a typing automaton. Essentially, every element type declaration is mapped to a corresponding transition rule with the element type declared serving as the rule's result state and complex content or simple content condition as appropriate.[3] The typing au-

[3]For a clear presentation, we use the notation `'et.etID'` as a placeholder to denote the string ID of an

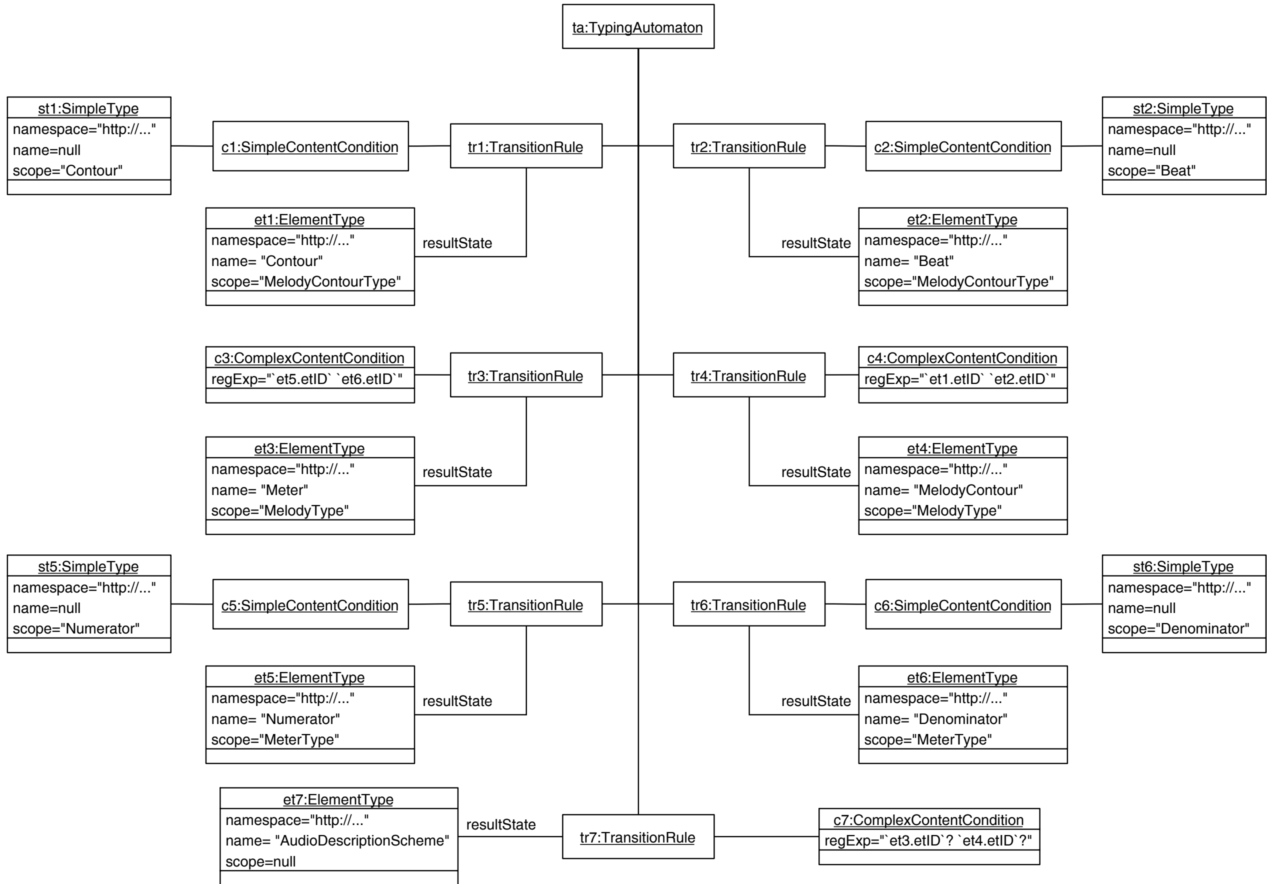

Figure 11: Example typing automaton (UML object diagram)

tomaton faithfully reproduces all the element type declarations of the original schema definition, except the declaration of the `AudioDescriptionScheme` element type. The problem is that the allowable content of `AudioDescriptionScheme` is defined by the complex type `AudioDSType` from which other complex types are derived. But typing automata as introduced so far with simple and complex content conditions do not yet provide an adequate handling of complex type derivation and the permissible use of `xsi:type` attribute values within XML documents to announce compliance of an element to a derived type. For simplicity, we therefore assume for the construction of transition rule `tr7` that `AudioDescriptionScheme` elements are always

element type `et` within the regular expressions of complex content conditions. For example, the content model of the element type `MelodyContour` consisting of a sequence of elements of types `Contour` and `Beat` represented by the objects `et1` and `et2` in the figure is translated to the complex content condition `c4` bearing the regular expression `‘et1.etID‘ ‘et2.etID‘`.

filled according to the complex type `MelodyType` with a sequence of optional elements of types `Meter` and `MelodyContour` and never according to a derived type such as `MeterType`. We refer the reader again to [62] for a discussion of how typing automata can be extended to cope with this situation more adequately.

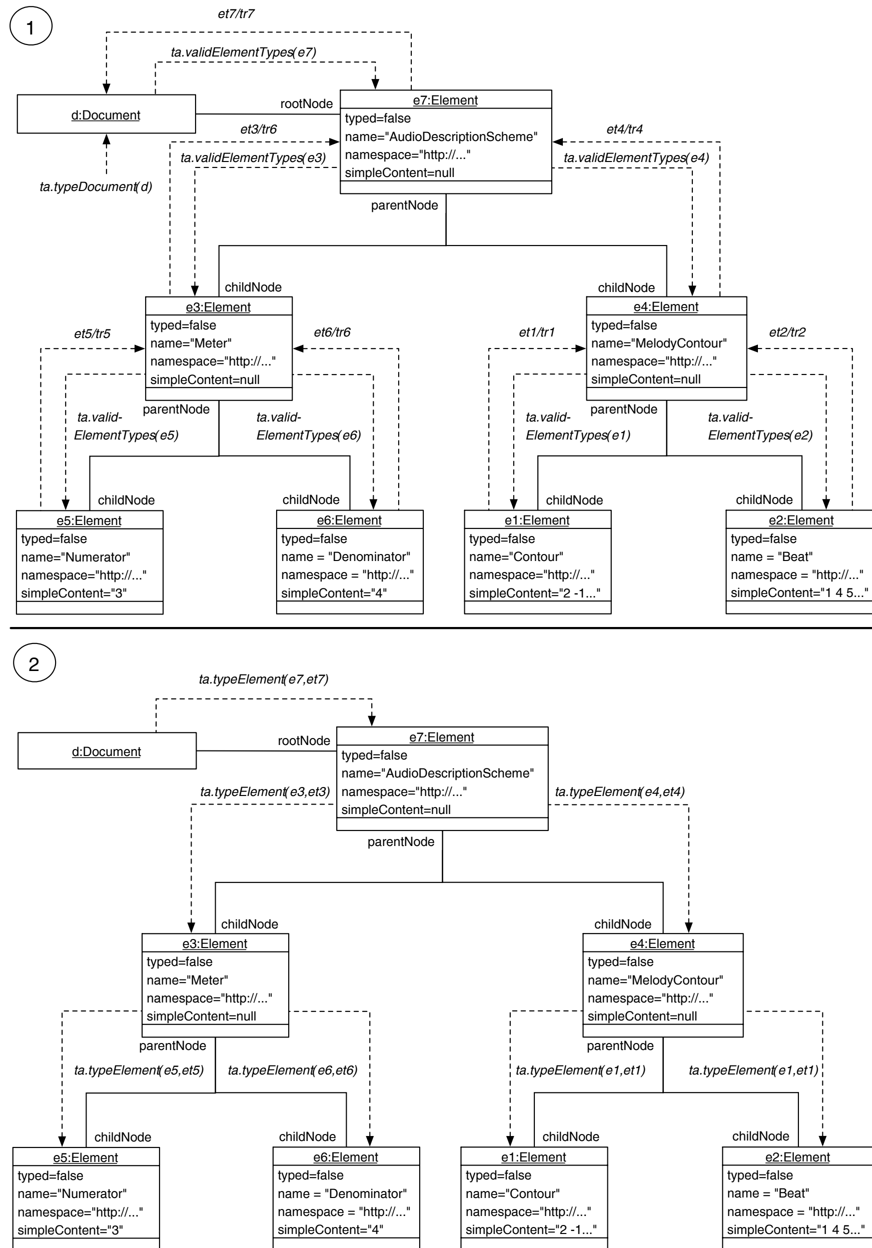

Figure 12: Typing automaton behavior (UML object diagrams)

The object diagrams of Figure 12 illustrate the behavior of typing automata by showing how the typing automaton of Figure 10 reacts when fed by the document manager via the method

`typeDocument()` with the example document of Figure 2 in a TDOM representation where all of the documents elements are kept in untyped representation. As mentioned before, this might happen as a second step during the import of the document. The behavior of the typing automaton can be divided into two phases. During the validation phase which is covered by the upper diagram (1), the automaton tries to validate the document by inferring the element types which the document's elements correctly instantiate. The automaton initiates this phase by trying to determine the valid element types for the document's root element using its method `validElementTypes()`. This method recursively tries to determine the valid element types of the root's direct and indirect child elements thereby descending down to the leaf elements as indicated in the diagram by dashed arrows heading down the document.

On the way back up from the recursion (again indicated in the diagram by dashed arrows heading up the document), the automaton attempts to apply each of its transition rules to each element. A transition rule is applicable to an element if and only if (1.) the name and namespace of the element type forming the rule's result state match the corresponding properties of the element and (2.) the rule's condition evaluates to true for the element's content. If a transition rule is applicable to an element, the element is considered valid to the element type declaration represented by the rule and the automaton memorizes the rule's result state as a valid type of the element. In the diagram, the valid element types of the elements are shown as labels of the dashed arrows heading upwards along with the respective rules which decided validity.

To evaluate the applicability of a transition rule with a simple content condition during the validation phase, the `evaluate()` method of the class `SimpleContentCondition` examines whether the element passed to this method has simple content and whether an instance of the simple type referenced by the simple content condition can be constructed successfully from the textual representation of this content employing the simple type's instance factory. To evaluate the applicability of a transition rule with a complex content condition, the `evaluate()` method of the class `ComplexContentCondition` checks whether the element passed to this method has complex content and whether there exists a sequence of valid element types for the element's child elements such that the concatenation of the string IDs of these element types satisfies the string regular expression of the complex content condition.

When the typing automaton again reaches the root element of the document and finds that

this element is valid to at least one unscoped, globally visible element type, the automaton considers the document to be valid against its schema definition and enters its second processing phase: the typing phase in which the automaton brings all elements to typed representation. As illustrated by the lower diagram (2), the automaton initiates this phase by picking one of the globally visible element types to which the root element is valid and calls its method `typeElement()` passing the element as well as the element type. This method then switches the element to typed representation by interlinking it with the element type (not shown in the figure). It further looks up the transition rule which decided the validity of the element to this type and calls the `type()` method of the condition of that rule to bring the element's content to typed representation as well. In case of a complex content condition, `type()` takes that sequence of valid element types for the element's child elements which satisfied the condition during the validation phase and recursively brings the child elements to typed representation calling the `typeElement()` accordingly as indicated by the dashed arrows in the diagram. In case of a simple content condition, `type()` produces a simple type instance from the element's simple content and attaches it to the element (again not shown in the figure).

The computational complexity of a typing automaton's behavior is moderate: it can be shown that the running time of a deterministic typing automaton[4] never grows more than linear with the number of elements in a document and transition rules in the automaton [62].

4.3.3 Document updates

On the basis of TDOM, applications cannot only fine-grainedly access the structure and contents of the XML documents maintained by the document manager; also, they can modify and update these documents at any desired level of granularity. Thereby, as mentioned previously, TDOM's ability to mix typed and untyped representations in a single document provides a high degree of flexibility as it allows to relieve elements that are to be modified during an update from the yoke of schema correctness for the duration of the update by transforming them from typed to untyped representation. When it comes to validate the correctness of the updates and

[4]A typing automaton is deterministic if any element in a document can be valid to at most one element type. Like regular tree automata [6], non-deterministic typing automata can be algorithmically transformed into equivalent deterministic ones in an additional processing step.

to bring the updated elements back to typed representation after the update, it is often not advisable to transform all the document's elements to untyped representation only to apply the typing automaton for the document's schema definition to the whole document as seen before: the update might just have concerned small fractions of the document and those parts not updated might still be valid to the same element types as before. Throwing away the typed representations of these elements would thus be a waste of precious processing power invested in previous runs of the typing automaton.

It is therefore a more viable strategy try to limit the application of a typing automaton to only those parts of a document that were actually changed by an update [62, 44]. The boundary to those parts is marked by the topmost of those elements which have been brought to untyped representation during the update, i.e., those elements brought to untyped representation whose parent elements are still in typed representation. The document manager can keep track of these elements throughout an update with relatively little effort; it can also memorize the element types these elements had before they were turned to untyped representation.

For each of the topmost elements in untyped representation, the document manager can then use the `validElementTypes()` method of the typing automaton to determine whether the type the element had before the update is still among the element's valid types. If this is the case, the document manager can invoke the automaton's `typeElement()` method to bring the element (and with it its child elements) back to typed representation on the basis of this element type without the need of having to revalidate and retype further parts of the document: the typed representation of the element's parent element is still correct. If this is not the case, the document manager can memorize the type of element's parent element, bring the parent element to untyped representation (and with it all of its child elements), treat it as a new topmost element in untyped representation, and relaunch processing as before.

In the worst case, this might result in a cascading untyping of parent elements until the root element of the document is reached. This is basically the same as if the all the document's elements had been brought to untyped representation prior to the application of the typing automaton. In many practical cases where updates make only local changes to documents without modifying their overall structure and validity, however, the outlined approach can be expected to perform substantially better because it preserves existing typed representation

wherever possible.

4.4 Routine framework

With the simple type framework, PTDOM is able to keep basic document contents – even those of non-textual nature – in data structures that are adequate and efficient for the particular content type. But so far, routines are lacking that provide the functionality to work with these contents in a reasonable and type-adequate manner. The routine framework component of the PTDOM architecture allows to seamlessly integrate arbitrary such routines: not just type-specific functions and procedures of general use for the processing of contents of a certain type but also custom functions and procedures providing functionality addressing individual application or user needs. With the routine framework, PTDOM becomes thus extensible with new functionality in a way that is comparable to object-relational DBMSs with their concept of user-defined routines, again contributing to the architecture's overall flexibility.

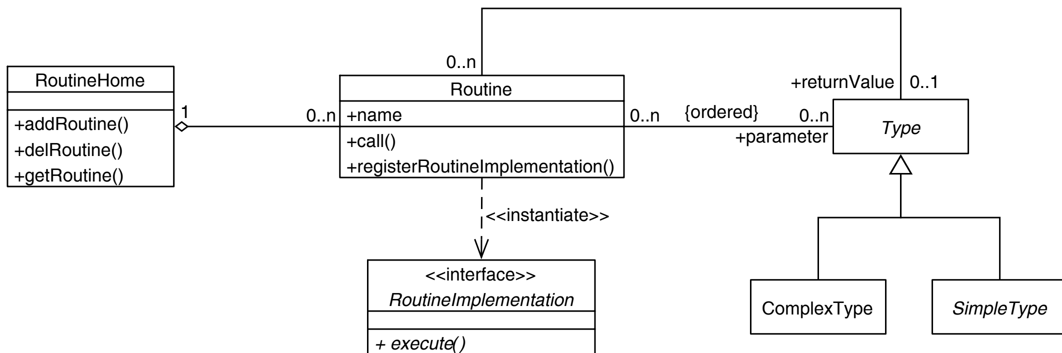

Figure 13: Routine framework overview (UML class diagram)

The class diagram of Figure 13 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 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 [38], can be systematically integrated into PTDOM.

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 [57] returns the most specific one.

4.5 Index framework

The availability of value index structures for indexing elements and attribute values along their content is an important prerequisite for an efficient querying of large collections of XML documents. In order to quickly retrieve all melody descriptions covering songs with a meter of $\frac{3}{4}$ from a collection of XML documents complying to the schema definition of Figure 1, for example, it would definitely be of help if suitable value index structures such as B-Trees were available to index the content of `Numerator` and `Denominator` elements. The index framework component of the PTDOM architecture with hash tables, B-Trees, and R-Trees not only provides a rich set of such value index structures; 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 [19] that allow the integration of new value index structures into object-relational DBMSs and similarly contributes to the extensibility and overall flexibility of the PTDOM architecture.

The class diagram of Figure 14 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 docu-

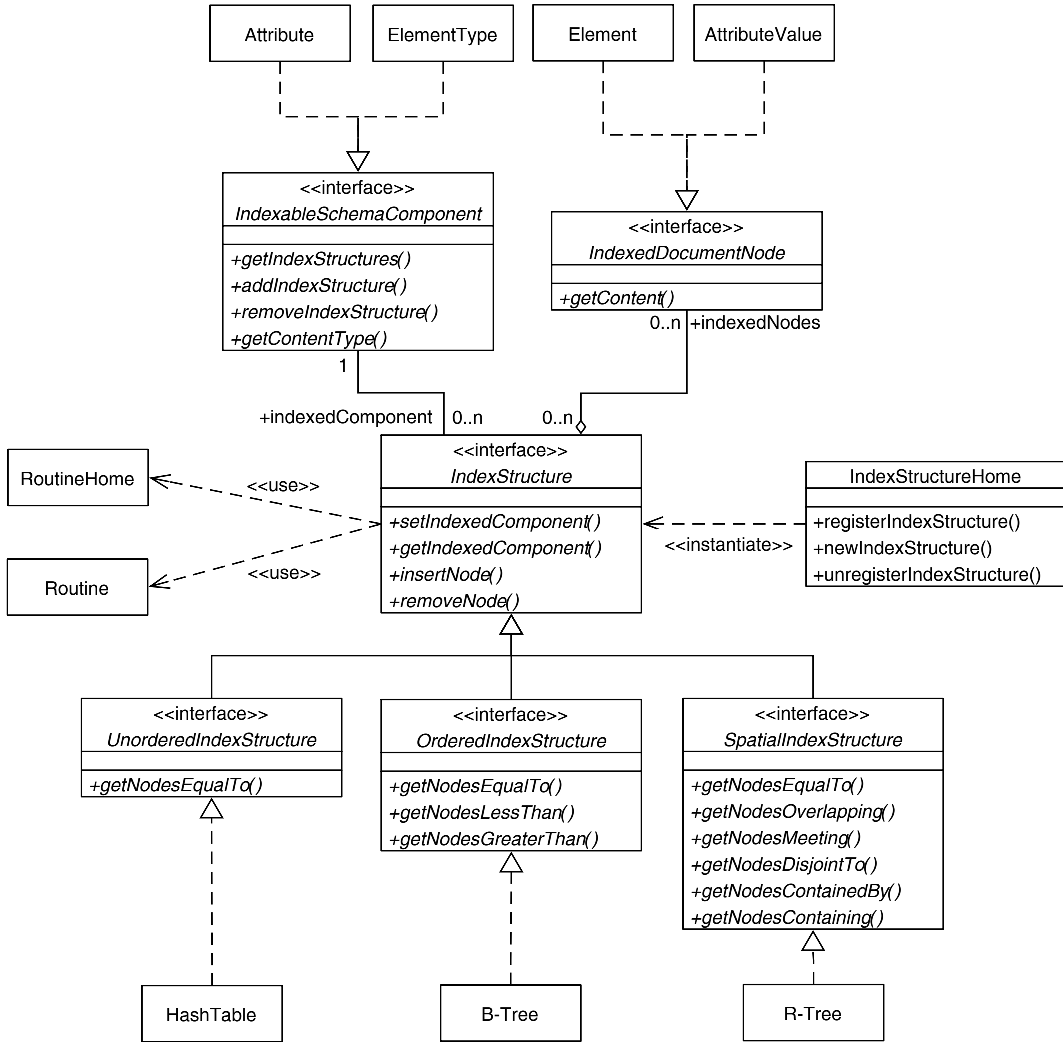

Figure 14: Index framework overview (UML class diagram)

ment 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 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[5] 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 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

[5]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.

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 1 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.

4.6 Query evaluator

The architectural 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 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 type-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.

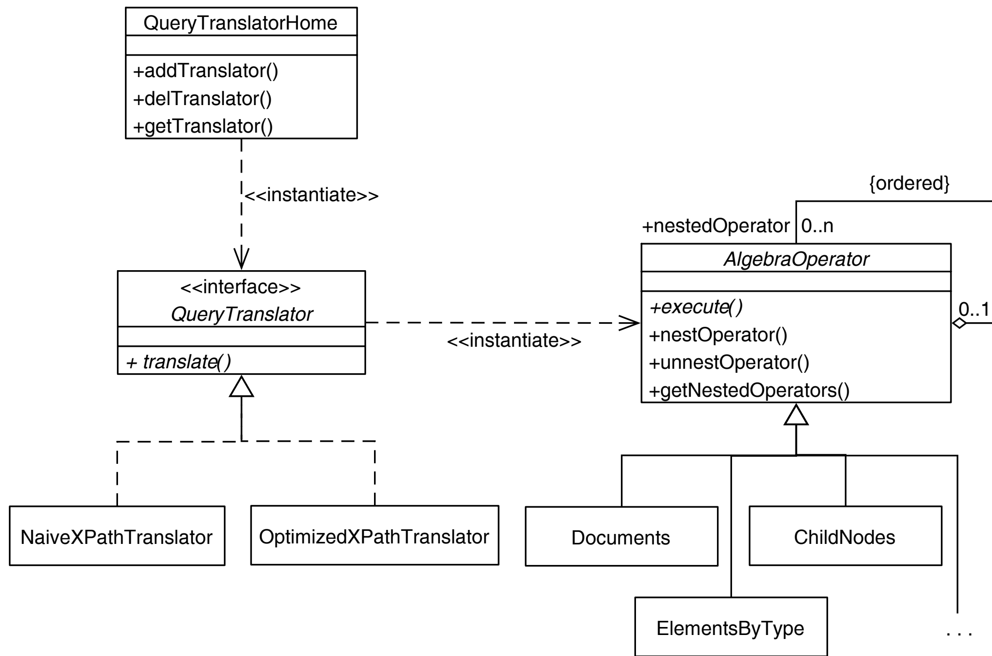

Figure 15: Query evaluator component overview (UML class diagram)

Essentially, the query evaluator component (an overview of which is given by the class diagram of Figure 15) provides an implementation of the PTDOM query algebra [47]. 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.[6] The query evaluator component implements the different operators of the algebra – each of which takes one or more sets of document nodes[7] 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 implementa-

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

[7]For compatibility with the XPath data model for XML documents [14], 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, doctypes, 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.

tions 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 likely generate more efficient operator trees.

Figure 16 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 operators like `filterByConstPredicate` benefit from the fact that (simple) content of elements and attribute values in typed representation is kept in type-adequate and efficient data structures and not just as text and that arbitrary, even 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.

The object diagram to the upper left (1) of Figure 17 illustrates how the XPath expression `//Meter[Denominator/data() = 128]` can be evaluated on the XML documents within PT-DOM's document manager that are valid to the `Melody` description scheme of Figure 1 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 docu-

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 16: PTDOM query algebra overview

ment 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 oper-

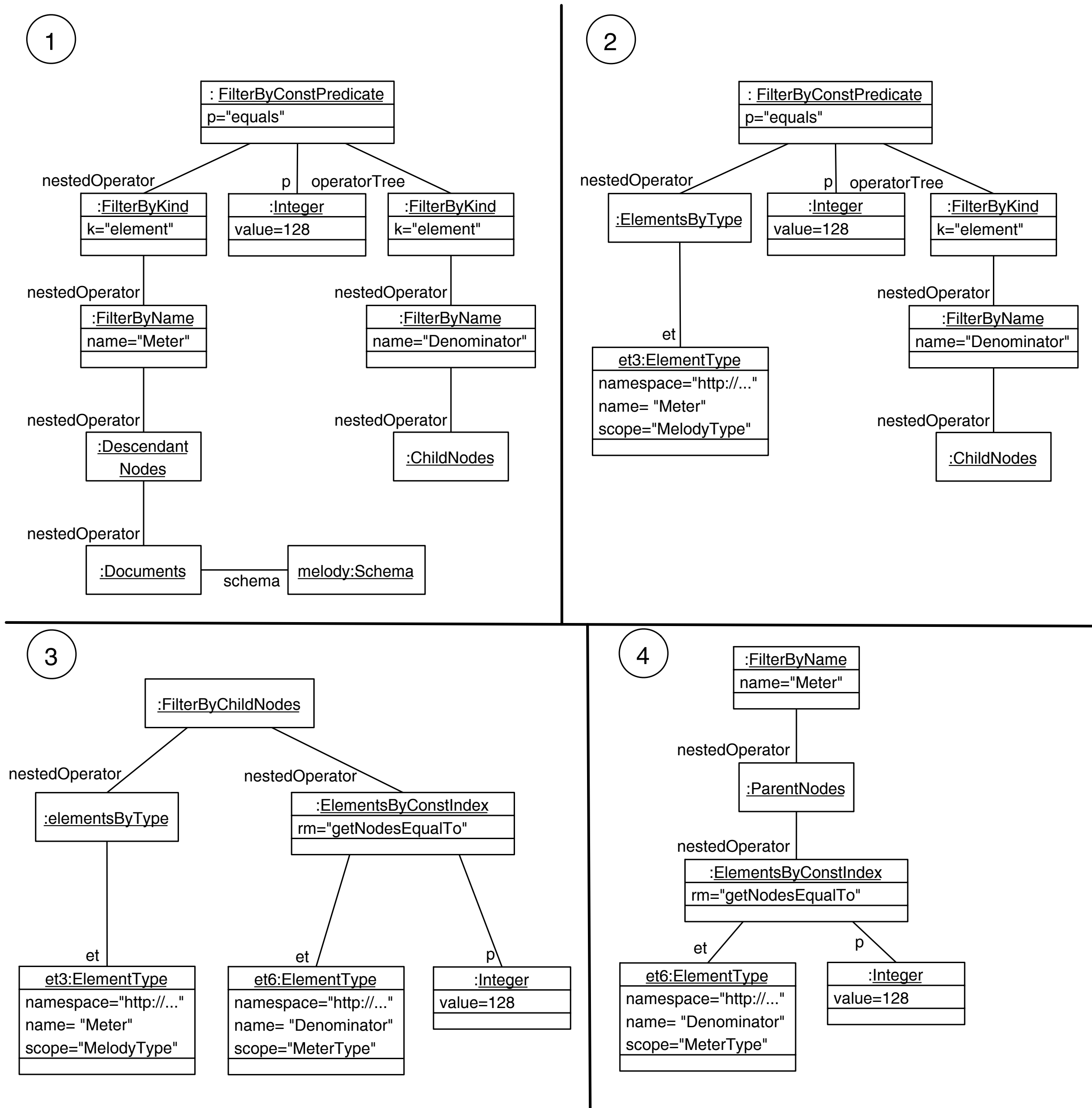

Figure 17: Query algebra example (UML object diagrams)

ators 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 17 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 17 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 18. 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

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(E_1[E_2])= filterByExistence(h_s(E_1),h_s(§E_2))	11. h_s(E_1[E_2/data() p c])= filterByConst- Predicate(h_s(E_1), h_s(§E_2),c,p)	12. h_s(E_1[E_2/data() p E_3/data()])= filterByPredicate(h_s(E_1),h_s(§E_2), h_s(§E_3),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(E_1[et_s/data() p E_3/data()])= filterByChildNodes(h_s(E_1), elementsByIndex(h_s(E_1/E_3),et_s,rm_p))	19. h_s(E_1[et_s/data() p /E_3/data()])= filterByChildNodes(h_s(E_1),elementsBy- Index(h_s(/E_3),et_s, rm_p))	. . .

Figure 18: Heuristic XPath translation overview

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 optimized query translator will transform the example XPath expression `//Meter[Denominator/data() = 128]` to operator tree (2) of Figure 17 by applying rules 11, 16, and 5 of Figure 18 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 query optimizer into the query evaluator component.

5 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 Section 4 and that employs PSEPro as its storage backend. We are considering to make a public release of this prototype in the near future. 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 19: 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 [28, 29, 30] 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 19 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.

The second experiment measures the effort required for importing XML documents follow-

	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 20: Results of document import

ing 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 1. 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 20 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 21: 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 21 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 4.6.

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 4.6, 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.

6 Conclusion

In this paper, we have emphasized the importance of schema definitions for the management of XML documents in a database and illustrated the various roles these definitions can play for that purpose. We have performed an analysis of a broad spectrum of current XML database solutions – ranging from native solutions to database extensions, from commercial products to research prototypes – concerning the ways in which they make use of available schema definitions and

uncovered remarkable deficiencies: many XML database solutions do not at all take account of schema definitions for XML document management; and those who do typically restrict the usage of schema definitions to mere document validation. None of the XML database solutions examined also exploits available schema definitions for profound document typing, path indexing, and query optimization.

We have then presented the system architecture of PTDOM, a highly extensible native XML database solution which, motivated by the deficiencies of current solutions, we have developed with a special focus on utilizing available schema definitions for XML document management. The central component of PTDOM is an MPEG-7 DDL/XML Schema-compliant schema catalog whose schema definitions are employed not just for mere document validation but also for the production of appropriately typed representations of basic document contents, for path indexing, and, though still in a rudimentary fashion, for an optimized evaluation of XPath expressions. Apart from conceptual benefits of such a schema-aware approach to XML document management concerning issues such as database consistency, treatment of non-textual document contents, indexing, and query optimization, we have provided in this paper initial experimental results indicating that the approach should also be viable in practice for many XML database applications and bring noticable benefits with regard to query efficiency.

We are currently developing PTDOM into several directions. We are applying the simple type and routine frameworks to integrate support for all the elementary simple types offered by MPEG-7 DDL [39, 1] and the basic functions and operators proposed for the XQuery 1.0 and XPath 2.0 standards [38] with PTDOM. Furthermore, we are investigating how the PTDOM query algebra can be extended to reach the expressiveness of the XQuery language and how a sophisticated query optimizer that exploits the rich and detailed schema information available within PTDOM's schema catalog can be integrated with the query evaluator component. Finally, we are exploring the application of PTDOM for the development of an MPEG-7-based, database-driven multimedia metadata tool suite.

References

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

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

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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

[33] 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.

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

[35] 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.

[36] 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.

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

[38] 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.

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

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

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

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

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

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

[53] Software AG. Tamino XML Schema Language (TSD3). System Documentation Version 3.1.1, Software AG, November 2001.

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

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

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

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

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

[59] 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.

[60] 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.*

[61] 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.*

[62] 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.*

[63] 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.

[64] 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.

[65] 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.

[66] X-Hive Corp. X-Hive/DB 2.1 – Manual. System Documentation Release 2.0.2, X-Hive Corp., May 2002.

[67] XML Global Technologies, Inc. GoXML DB Administrator Help. System Documentation Version 2.0.1, XML Global Technologies, Inc., December 2001.

[68] XML:DB Initiative for XML Databases. Frequently Asked Questions about XML:DB. Online Article, available under http://www.xmldb.org/faqs.html, February 2003.