Technical Report, No. TR-2003301, Dept. of Computer Science and Business Informatics,
University of Vienna, February, 2003

A Typed Representation and Type Inference for MPEG-7 Media Descriptions*

Utz Westermann, Wolfgang Klas

Department of Computer Science and Business Informatics

University of Vienna, Austria

{gerd-utz.westermann,wolfgang.klas}@univie.ac.at

Abstract

MPEG-7 is a promising standard for multimedia content description. Adequate means for the management of large amounts of MPEG-7 media descriptions are needed in the near future. Essentially, MPEG-7 media descriptions are XML documents following media description schemes defined with an extension of XML Schema named MPEG-7 DDL. However, XML database solutions available today are not well-suited for the management of MPEG-7 media descriptions. They typically neglect type information available with media description schemes and represent the basic contents of media descriptions as text. But storing non-textual multimedia data typically contained in media descriptions such as melody contours and object shapes textually and forcing applications to access and process such data as text is neither adequate nor efficient. In this paper, we therefore propose the Typed Document Object Model (TDOM), a data model for XML documents that can benefit from available schema definitions and represent the basic contents of a document in a typed fashion. Through these typed representations, applications can access and work with multimedia data contained in MPEG-7 media descriptions in way that is appropriate to the particular type of the data. Along with TDOM, we propose typing automata as an

*This report constitutes a significantly extended version of Technical Report TR-2003301, Dept. of Computer Science and Business Informatics, University of Vienna, February, 2003

executable intermediary representation of media description schemes. Typing automata are not only capable of validating MPEG-7 media descriptions against their description schemes but also of inferring appropriate typed representations of their basic contents by exploiting type information. Thereby, TDOM and typing automata constitute solid foundations for an XML database solution that enables the adequate management of MPEG-7 media descriptions.

1 Introduction

Recently, there have been considerable efforts to standardize the description of multimedia content. This has resulted in a variety of standards, such as the Dublin Core Metadata Element Set [15], Learning Object Metadata [27], VRA Core Categories [58], and the Multimedia Content Description Interface (MPEG-7) [30, 29, 45, 46]. Being an ISO standardization effort which is backed by prominent broadcasting companies, consumer electronics manufacturers, and telecommunication service providers and which has reached a mature state by the end of 2001, MPEG-7 receives considerable attention in the multimedia community. What makes MPEG-7 particularly attractive is that it is targeted at the description of multimedia content on a technical, feature-oriented level as well as on a semantic level. For instance, it is not only possible to describe the frequency spectrum of a song recording in an MPEG-7 media description. It is also possible to refer to the lyrics and the musical score, all within the same description.

The scope of standardization basically comprises two parts: a Description Definition Language (MPEG-7 DDL) [31] with which schemes for the description of media can be specified, and, defined via MPEG-7 DDL, a comprehensive set of media description schemes that are useful for a variety of applications. The media description schemes standardized include schemes for visual media [32] and audible media [33] as well as schemes of general use [34]. Applications are not limited to these standardized media description schemes: new description schemes can defined with MPEG-7 DDL, either from scratch, or by extending or combining existing description schemes.

By the diversity of aspects with which content can be described and by the extensibility

of the standard with new description schemes, MPEG-7 is expected to face wide-spread use in a broad range of applications: media archives, journalism, education, entertainment, etc. Therefore, means for the effective management of large amounts of MPEG-7 media descriptions are certainly needed.

Basically, MPEG-7 media descriptions are XML documents that are valid to a media description scheme expressed in MPEG-7 DDL. Thus, it seems natural to employ XML database solutions for the management of MPEG-7 media descriptions. Closer examination of current XML database solutions for their suitability for MPEG-7, however, reveals difficulties. One of the main problems with current solutions is that they typically represent the basic contents of XML documents, i.e., simple content of elements and the content of attribute values, as text. But textually representing multimedia data such as melody contours and object shapes often contained in MPEG-7 media descriptions is inadequate: textual representations of non-textual data consume unnecessary storage space, do not preserve the meaning of the data (e.g., with respect to indexing), and are inefficient and cumbersome to handle such that applications are forced to constantly translate the textual representations to data structures better suiting the particular data type at the cost of considerable processing power. Clearly, more adequate database solutions are required for MPEG-7.

In this respect, we make several substantial contributions in this paper: we motivate and present several basic but nevertheless essential requirements for the management of MPEG-7 media descriptions. Along these requirements, we analyze existing XML database solutions – native XML database solutions as well as extensions for relational systems, commercial systems as well as research prototypes – fortifying the demand for more suitable MPEG-7 database solutions. As a solid foundation for such a solution, we then propose the Typed Document Object Model (TDOM). TDOM is an object-oriented data model for XML documents specifically designed bearing the requirements for the management of MPEG-7 media descriptions in mind. The model's outstanding feature is that type information contained in media description schemes written in MPEG-7 DDL is exploited to represent the basic contents of an XML document in a typed fashion and not just as text. In typed representation, simple element content and the content of attribute values is kept in data structures that are appropriate for the respective content type and that come with type-specific operations to reasonably work with the

content. Thereby, applications can process the multimedia data contained in MPEG-7 media descriptions more adequately and efficiently. Finally, we introduce typing automata as a formal mechanism for use with TDOM that allows to infer such typed representations out of media description schemes expressed in MPEG-7 DDL.

The remainder of the paper is organized as follows: Section 2 derives essential requirements for the adequate management of MPEG-7 media descriptions. Section 3 evaluates existing XML database solutions according to these requirements. Section 4 introduces and gives a thorough definition of TDOM. Section 5 introduces and formally specifies typing automata. Section 6 concludes the paper with a summary and an outlook to current and future work.

2 Requirements for the management of MPEG-7 media descriptions

In this section, we briefly illustrate the nature of MPEG-7 media descriptions (2.1). We then derive a set of fundamental requirements for the management of such descriptions (2.2).

2.1 MPEG-7 media descriptions

MPEG-7 is strongly committed to XML and related standards. MPEG-7 DDL, the language used for the definition of media description schemes, is a superset of XML Schema [56, 3], a schema definition language for XML documents recently standardized by the W3C. Certain extensions to XML Schema were considered neccessary to better cope with the peculiarities of multimedia data. In particular, support for array and matrix data types as well as additional data types for time points and time durations were added to XML Schema. Regarded as an extended XML Schema, MPEG-7 DDL is thus just another schema definition language for XML documents. Since media description schemes are defined with MPEG-7 DDL, an MPEG-7 media description complying to a given media description scheme is consequently an XML document that is valid with respect to the schema definition given by the description scheme.

We would like to illustrate the concepts of media description schemes and media descriptions with an example. The MPEG-7 `Melody` media description scheme [33] is a representative

description scheme that can serve as the basis for the realization of query-by-humming applications. A slightly simplified version of this description scheme expressed in MPEG-7 DDL is shown in Figure 1.

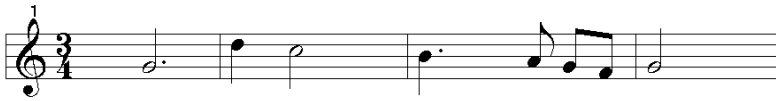
```
...

<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: Simplified MPEG-7 `Melody` media description scheme

According to the media description scheme depicted, the melody of a song (see the declaration of the complex type `MelodyType` in the upper left column) can be described by its meter and melody contour (element types `Meter` and `MelodyContour` declared in `MelodyType`). The meter of a melody is a fraction number consisting of a numerator and denominator (see element types `Numerator` and `Denominator` declared in the complex type `MeterType` in the right column). There is the restriction that the numerator must be an integer value in the interval from 1 to 128, while the denominator must be a power of two in the same interval. The melody contour consists of an optional contour and beat (see element types `Contour` and `Beat` declared in the complex type `MelodyContourType` in the lower left column). The contour of a melody is a list of integer values giving a measure for the distance between every two consecutive notes

of the melody while the beat is a list of integer values associating every note of the melody to its position in the beat.



```
<!-- 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 an MPEG-7 media description

A media description complying to the `Melody` media description scheme is an XML document that is valid to the schema definition presented. Figure 2 gives an example of such a document describing a small fraction of the melody of the song “Moon River” by Henry Mancini (taken from [33], page 101).

There are some general observations that can be made on MPEG-7 media descriptions:

- MPEG-7 media descriptions are XML documents.
- MPEG-7 media descriptions comply to media description schemes expressed in MPEG-7 DDL, a schema definition language for XML documents.
- The set of available media description schemes is not fixed by MPEG-7. The standard ships with a multitude of predefined schemes such as our example `Melody` media description scheme but applications may create new description schemes with MPEG-7 DDL if desired.

- Much of the information encoded in XML documents that constitute MPEG-7 media descriptions is not of a textual nature. Large portions of the information consist of numbers and mathematical structures such as lists, vectors, and matrices – usually to describe rather technical aspects of media content. As a matter of fact, about 84% of the media description schemes predefined by the standard for visual and audible content in [32] and [33] consist primarily of non-textual data.

2.2 Requirements

As we have observed, MPEG-7 media descriptions are XML documents. Thus, the problem of adequately managing MPEG-7 media descriptions can be curtailed to the problem of managing XML documents in principal. In literature, general requirements for XML database solutions have already been identified [48]. Among the desired features of an XML database solution are rich document modeling capabilities, the availability of a query language, support for document updates, the availability of index structures, the management of access rights, support for transactions as well as backup and recovery.

In the following, however, we want to specifically look onto the management of XML documents from the perspective of MPEG-7: we motivate and present four very basic but nevertheless critical requirements for the effective management of MPEG-7 media descriptions. Namely, the requirements are fine-grained representation of description structure, typed representation of description content, support for updates, and support for MPEG-7 DDL. As it will turn out in Section 3, current XML database solutions fail in fulfilling all of these basic requirements.

Fine-grained representation of description structure. With MPEG-7 DDL, schemes of arbitrary complexity for the description of media content can be defined, describing media content from possibly very different points of view. However, not every application working with media descriptions conforming to a complex description scheme can be expected to process the full scope of a description. Rather, applications will access only those parts of a description that are necessary to fulfil their particular tasks.

Enabling fine-grained access to the constituents of a media description is therefore essential for the adequate management of MPEG-7 media descriptions. This calls for the fine-grained

representation of the structure of the media description. With a faithful reproduction of the hierarchy of the various nodes, i.e., markups, of which the description consists, applications can access exactly those parts that they are interested in. In contrast, if the description was represented as a single unstructured object, an application would always have to access the complete description and decompose it into its constituents – even if the application is interested in just a small fraction.

Typed representation of description content. As we have already noticed by the means of the `Melody` media description scheme of Figure 1 as a typical representative of many description schemes predefined by MPEG-7, much of the information encoded in media descriptions consists of non-textual data like numbers and rather complex structures like lists. Since MPEG-7 media descriptions are XML documents and, as such, a form of text documents, these data are encoded as text.

This might be adequate for the platform-independent exchange of media descriptions. It is doubtful for several reasons, however, whether textual representation of non-textual data is also reasonable for the management of media descriptions within a database: textual representations of non-textual data typically consume more storage space than equivalent binary representations. Moreover, they are less efficient and cumbersome to handle. A good example for this point is the list of integer values being the content of the `Contour` element in the media description of Figure 2. The effort necessary to retrieve the 4th element of the list on the basis of the list's textual representation, i.e., through string operations, is significantly higher than the effort necessary for the same action on the basis of an adequate data structure, e.g., an array. Therefore, applications must usually translate textual representations of non-textual data to internal data structures more appropriate to the particular type of data before they can adequately work with the data – at the cost of considerable processing power. Finally, textual encoding of non-textual data does not necessarily preserve the semantics, e.g., with respect to ordering. For instance, the alphanumeric order of the textual representations of integer values differs from their numeric order hindering reasonable indexing.

Given these problems, a suitable database solution should represent the basic contents of an MPEG-7 media description – namely, simple content of elements and content of attribute values

– in a typed fashion and not as text. With typed representation, we mean that these contents are kept in data structures that are adequate to the particular content type and come with type-specific operations to reasonably work with the content. To that end, a database solution should not only support the whole lot of simple data types predefined with MPEG-7 DDL [31, 3] but also the variety of derivation methods for simple types coming with the standard – MPEG-7 DDL allows to flexibly derive new simple types from existing ones in a schema definition. Examples of such derived simple types are the list type and the range-restricted integer type defining the allowed contents of the `Contour` and `Numerator` element types in Figure 1: both are based on the predefined type integer.

Fine-grained updates. It is unrealistic to assume that MPEG-7 media descriptions are produced in one shot and then never touched again. Just like the media content they describe, media descriptions evolve and are constantly subject to change during the different phases of the content's lifecycle. Acknowledging that media descriptions are subject to change, we demand that a database solution should offer adequate means for updating media descriptions. Just like it is necessary to offer applications fine-grained access to the nodes of a possibly complex media description, applications should be allowed to perform fine-grained updates on any part of the description – having to unload the complete description from the database, to modify it outside the database, and to reinsert it into the database just to update a small fraction is definitely not efficient and prevents concurrent access.

Support for MPEG-7 DDL. For the suitable management of MPEG-7 media descriptions, it is of advantage to be capable of processing media description schemes expressed in MPEG-7 DDL. The exploitation of the information available in these schema definitions is on the one hand required to ensure the consistency of the database contents by validating whether an XML document constitutes a correct media description with respect to a given media description scheme. Typical occasions where such a validation is reasonable are during the import of a media description into a database and during the update of a media description to decide whether an update operation can be permitted without violating the description scheme. On the other hand, processing of type information contained in schema definitions is necessary to be able to infer the types and, by these types, appropriately typed representations of the basic

contents of a media description.

One might argue that the use of media description schemes to type the basic contents of a media description could prohibitively increase the complexity of inserting new media descriptions into a database and the complexity of database updates such that the benefits of typed representations are overweighed by performance problems. It is our experience, however, that this does not constitute a problem for most applications. If one admits that it is a vital task of a database solution to ensure the consistency of database contents, an MPEG-7 database solution must take the effort anyway to validate a media description against its associated media description scheme whenever the description is newly inserted into a database or updated.

During validation, the database solution already has to verify whether the textual representation of the simple content of an element or the content of an attribute value matches the type for the content as declared in the media description scheme. The additional effort necessary to bring the content into an appropriate typed representation after a successful verification is likely to be negligible for most applications, except for ones with very high update and insert frequencies and with harsh real-time constraints.

But for such applications, a database solution will already have to abstain from the validation of media descriptions after insertions and updates to noticeably save processing time – a time saving, the applications will presumably pay for later when they must ensure the consistency of media descriptions themselves and bring the basic contents of a description from their textual representations to internal ones more suitable for further processing whenever they access the description.

3 Analysis of XML database solutions

On the basis of the requirements elaborated in the previous section, we have examined current XML database solutions with regard to their suitability for managing MPEG-7 media descriptions. Figure 3 gives a summary of our analysis, showing whether or to what extent each of the solutions meets each of our requirements.

In the figure, two coarse categories of database solutions for XML documents are distinguished: native database solutions specifically designed for the storage of XML documents and

		Fine-grained Representation of Description Structure	Typed Representation of Description Content	Fine-grained Updates	Support for MPEG-7 DDL
Native Products	dbXML [43]	+	-	+	-
	eXcelon XIS [12]	+	Textual representation - interpretation as string or number can be explicitly specified for indexing purposes	+	Limited XML Schema support
	GoXML DB [50]	+	Though claimed [50] apparently not realized in Version 2.0.2	+	Limited XML Schema support
	Infonyte-DB [22]	+	-	+	-
	Tamino [42]	-	Textual representation - interpretation as string, real, or integer can be explicitly specified for indexing purposes	-	Limited XML Schema support
	TEXTML [29]	-	Textual representation - interpretation as string, date, or number can be explicitly specified for indexing purposes	-	-
	X-Hive/DB [48]	+	-	+	-
	Xindice [44]	+	-	+	-
Native Research	Lore [17]	+	Limited set of elementary simple types supported for attribute values only	+	-
	Natix [31]	+	Limited set of elementary simple types supported	+	-
	PDOM [19]	+	-	+	-
Relational Products	IBM DB2 XML Extender [20]	-	-	-	-
	Microsoft SQLXML [34]	-	-	-	-
	Oracle XML DB [18]	-	-	-	-
Relational Research	Monet XML [39]	+	-	+	-
	Shimura et al. [41]	+	-	+	-
	XML Cartridge [16]	+	-	+	-

Figure 3: Analysis of current XML database solutions (+ support, - no support, comment in case of partial support)

solutions based on relational database management systems (DBMS). Both categories are further subdivided into products – commercial ones as well as open source projects – and research prototypes. In the following, we briefly discuss the results of Figure 3 for native XML database solutions (3.1) and for relational XML database solutions (3.2) before concluding this section

with a summary (3.3).

3.1 Native XML database solutions

Taking a look at Figure 3, we can see that the analyzed native XML solutions have strong deficiencies with respect to the typed representation of the basic contents of a media description: most of the native solutions investigated represent simple content of elements and the content of attribute values of XML documents as text – with all the incurring problems with regard to the appropriate processing of non-textual data often contained in MPEG-7 media descriptions. eXcelon XIS [17], TEXTML [35], and Tamino [53] somewhat alleviate the lack of typed representations by allowing to manually specify whether the content of an element or attribute value is to be interpreted as a string, number, or date for indexing purposes. Apart from the fact that support for strings, numbers, and dates does not come even close to the broad variety of elementary simple types and simple type derivation methods available with MPEG-7 DDL, the content still remains represented as text.

To some extent, three of the examined native XML database solutions, the commercial GoXML DB [61] and the research prototypes Lore [22] and Natix [38], address the issue of typed representations. GoXML DB claims that the basic contents of an XML document are represented in a typed fashion. Experiments with the current Version 2.0.2, however, have revealed that this feature has apparently not been implemented yet. Content of attribute values and simple content of elements is represented and interpreted as text. Compared to that, Lore and Natix offer some elementary simple types for the typed representation of basic document contents. However, both solutions support just small subsets of the simple types predefined by MPEG-7 DDL. Simple type derivation methods, such as lists and matrices, are not supported at all. Furthermore, Lore limits the use of typed representations to the content attribute values; simple element content remains represented as text.

Even if the native solutions sufficiently supported the typed representation of the basic contents of an XML document, they would not be able to infer adequate typed representations for the contents of an MPEG-7 media description: no solution investigated can fully process MPEG-7 DDL schema definitions. In fact, most systems do not make use of schema definitions at all for XML document storage. Just eXcelon XIS, Tamino, and GoXML DB support more

or less limited subsets of XML Schema for the purpose of document validation not reaching the expressive power of MPEG-7 DDL.

3.2 Relational XML database solutions

Three principal approaches to storing XML documents in a relational DBMS can be distinguished. In the first approach, XML documents are directly stored in their textual format in a character large object (CLOB). Special stored procedures are provided to access the contents of a document from SQL, e.g., via XPath expressions [9]. Most XML database extensions of the major relational DBMS vendors support this approach, such as Oracle XML DB [23], IBM DB2 XML Extender [26], and Microsoft SQLXML [43]. With respect to the management of MPEG-7 media descriptions, however, it is obvious from Figure 3 that this approach is not adequate. As documents are stored as a whole in their textual format in a CLOB, there is neither a fine-grained representation of the structure of a media description nor a typed representation of the basic contents of a media description. Updates are only possible by replacing complete documents and any existing schema definitions are not exploited for the storage of XML documents.

The second approach to the management of XML documents in a relational DBMS keeps the nodes of a document and the hierarchical relationships between them in tables. There has been considerable research in this area (see [19] for an overview) and a lot of research prototypes have been developed. For our analysis, we have focused on three representative systems: Monet XML [49], Shimura et al. [52], and the XML Cartridge [21]. These systems have in common that they represent the structure of XML documents with a fine granularity and that they also allow fine-grained updates. Nevertheless, they are not suitable for the management of MPEG-7 media descriptions, since they do not make use of schema definitions to represent the basic contents a document in a typed manner but rather represent all contents as text.

The third approach to the management of XML documents in a relational DBMS maps the data conveyed in XML documents to application-specific database schemas. Even though this would in principal place all modeling capabilities available with the relational DBMS at the disposal for representing document content, we have decided to ignore this approach for the management of MPEG-7 media descriptions. The definition of an application-specific database

schema as well as the specification of the mapping between an XML format to this schema are elaborate manual tasks. Bearing in mind that MPEG-7 allows to define arbitrary description schemes in excess to those predefined with the standard, the effort necessary to cope with a media description following a previously unknown media description scheme is prohibitive. In literature, there are some approaches for the automatic derivation of relational database schemas from schema definitions for XML documents [51, 57, 16] and the automatic mapping between them. However, these are based on Document Type Definitions and whether they scale to the far more complex MPEG-7 DDL remains to be proven.

3.3 Summary

As we have seen, none of the XML database solutions examined, native as well as relational, is adequate for the management of MPEG-7 media descriptions. Their main limitations are the lack of typed representations for the basic contents of an XML document and the inability to process schema definitions expressed in MPEG-7 DDL to derive such typed representations. For the suitable management of MPEG-7 media descriptions, we therefore certainly see a need for a solution that focuses on exactly these points.

4 The Typed Document Object Model

As we have seen in Section 3, database solutions more suitable for the management of MPEG-7 media descriptions are needed. The heart of such a solution – just as with any other database solution – is a data model. The purpose of this data model is to provide a detailed, accurate, and adequate representation of the structure and contents of MPEG-7 media descriptions at a logical level. On the basis of such a logical representation, applications can gain access to the contents of media descriptions and process them more appropriately compared to the alternative of constantly working on the original textual format through parsing operations.

Furthermore, the data model can serve as the foundation for the physical storage scheme implemented by an MPEG-7 database solution. Closely orienting the physical storage scheme of MPEG-7 media descriptions in a database along the data model has the advantage that exactly those parts of a media description can be loaded from a database that are actually accessed

during the processing of a description on the basis of the data model thereby minimizing I/O operations and main memory consumption. Storing media descriptions in their original textual format – which requires to constantly load the entire description into main memory whenever it is accessed in order to parse it and bring it into the data model representation for further processing – is not very attractive as we could observe with relational XML database solutions like Oracle XML DB and IBM DB2 XML Extender in Section 3.

Since MPEG-7 media descriptions are XML documents, a suitable data model should be a model for XML documents that considers the basic requirements for the management of these descriptions that we have presented earlier in Section 2. In this section, we propose the Typed Document Object Model (TDOM), a conceptual object-oriented model for XML documents that we have created bearing exactly these requirements in mind. We begin by taking a brief look onto existing data models for XML documents unveiling their limitations concerning the representation of MPEG-7 media descriptions (4.1). Having thus fortified the need for a new data model, we then illustrate and give a thorough definition of TDOM (4.2).

4.1 Data models for XML documents

A variety of data models for XML documents have been proposed in literature, e.g., [36, 22, 21, 52, 49, 38]. Concerning their application for the representation of MPEG-7 media descriptions, however, these models suffer from mainly two weaknesses: firstly, they typically constitute variations of rather simple edge-labeled tree and graph data models. Though they provide fine-grained representations of MPEG-7 media descriptions in principle, these models often ignore more subtle aspects of the descriptions' structure like the ordering the child nodes of an element, markup different from elements and attribute values such as processing instructions and comments, or the distinction between attribute values and elements. Secondly, they usually do not support typed representations: simple element content and the content of attribute values is typically represented as text hindering the reasonable processing of non-textual data on the basis of these models. Those few models that support typed representations (e.g., [38, 22]) only offer limited subsets of the elementary simple types predefined with MPEG-7 DDL; none of the models to our knowledge supports the simple type derivation methods of MPEG-7 DDL.

In addition to the models originating from research, several data models for XML documents have appeared in the context of standardization efforts. Prominent representatives are the XPath Data Model which constitutes the foundation for the XPath language [9], the DOM Structure Model which is specified along with the DOM API by the Document Object Model (DOM) standard [40], and the XML Information Set [13]. These models generally offer detailed and accurate representations of XML documents. But their applicability for the processing of MPEG-7 media descriptions is limited, since they neglect the types of the basic contents of a description representing them always as text.

There are two recent developments with regard to standard data models for XML documents which are of particular interest for our aim to adequately represent MPEG-7 media descriptions, namely DOM Level 3 [41] and the XQuery 1.0 and XPath 2.0 Data Model [18]. The current DOM Level 3 standardization effort originally not only aimed at the fine-grained representation of XML documents but also at the fine-grained representation of the schema definitions to which the documents comply. In that context, Abstract Schemas [5] have been proposed as a schema-dialect-neutral model for the representation of schema definitions. Hence, Abstract Schemas might serve as a basis for the representation of media description schemes expressed in MPEG-7 DDL. However, Abstract Schemas do not reach the expressiveness of MPEG-7 DDL. It is furthermore noteworthy that work on Abstract Schemas has recently been canceled and that they will not be included in the final version of DOM Level 3.[1]

The XQuery 1.0 and XPath 2.0 Data Model is currently being defined as the foundation of the XQuery standardization effort for a common XML query language [4]. What makes the model interesting with regard to the representation of MPEG-7 media descriptions is that it supports the elementary data types predefined by XML Schema for the typed representation of simple element content and the content of attribute values. Nevertheless, difficulties concerning the use of the XQuery 1.0 and XPath 2.0 Data Model for MPEG-7 still remain: with the exception of lists, the current working draft does not support the far majority of the simple type derivation methods offered by XML Schema and MPEG-7 DDL for typed representations. Furthermore, the model is still in a very unstable state. For example, the paradigm followed for the specification of the data model has just been changed fundamentally compared to the

[1] See http://lists.w3.org/Archives/Public/www-dom/2002JulSep/0010.html.

previous working draft issued in April 2002. Instead of an open structural definition, the model is now opaquely defined similar to abstract data types.

4.2 TDOM in seven points

On the way to an adequate MPEG-7 database solution, the deficiencies of the existing data models for XML documents fortify the need for a new model that pays more attention to the basic requirements for the management of MPEG-7 media descriptions. With the Typed Document Object Model (TDOM), we now propose a data model which does exactly that.

TDOM is an object-oriented model for XML documents that carries on traditional DOM [40] to allow an appropriate representation of MPEG-7 media descriptions. In the following, we provide a detailed and illustrated definition of TDOM which we present in seven points closely oriented along our requirements for the management of MPEG-7 media descriptions.

Since TDOM is an object-oriented model, we employ UML class diagrams [2] for the definition of the various classes of the model and their interrelationships.[2] Whenever it is necessary to make formal statements about TDOM, we make use of the Object Constraint Language (OCL) defined as part of the UML standard.

1. TDOM is fine-grained.

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

The class diagram of Figure 4 introduces the classes of TDOM that are responsible for the representation of an XML document and the detailed reproduction of its structure. The class

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

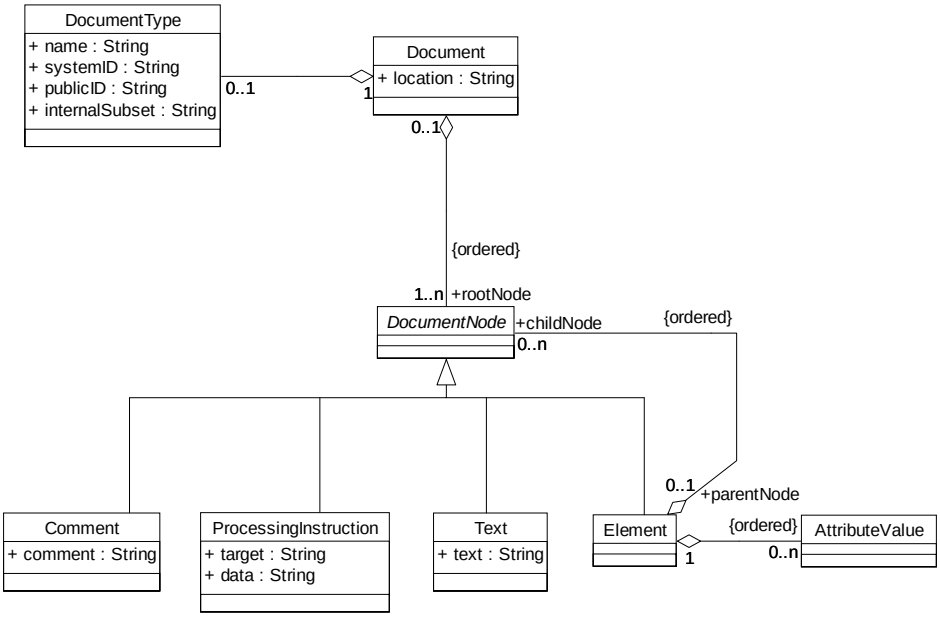


Figure 4: Representation of document structure (UML class diagram)

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

Being an abstract base class, `DocumentNode` subsumes one class each for the representation of the primal kinds of nodes of which an XML document may consist: `Comment` represents comments, `ProcessingInstruction` represents processing instructions together with their associated source and target declarations, `Text` copes with text interspersed with other document nodes in mixed content, and `Element` represents elements. Through elements, the hierarchical structure of a document is established – elements are the only kind of document nodes that may contain other nodes as their child nodes. This is expressed by the aggregation between `Element` and `DocumentNode`. Since elements can be further described by attribute values, TDOM introduces the class `AttributeValue` for their representation which is aggregated by `Element`.

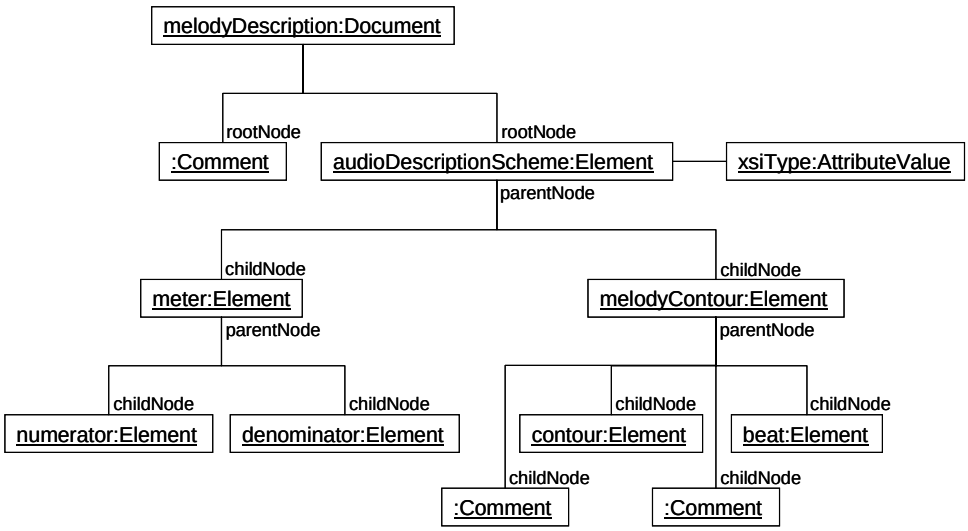


Figure 5: Structural representation of the example MPEG-7 media description with TDOM (UML object diagram)

The UML object diagram of Figure 5 exemplifies the structural representation of an MPEG-7 media description with TDOM using the example melody description of Figure 2. The description itself is represented by the object of the class `Document` depicted at the top of the diagram; the document nodes contained in the description are represented by objects of the TDOM-classes corresponding to the particular kind of node. Via the references of the `Document` object to its root nodes and the references of the `Element` objects to their child nodes and attribute values, TDOM reconstructs the hierarchical structure of the example description. Outgoing from the `Document` object, an application can thus traverse the structure of the example media description and access and manipulate any desired document node at any granularity.

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

Constraint 1 (Unique attribute values)

context `Element`

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

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

Constraint 2 (Root nodes)

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

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

Definition 1 (Root element)

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

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

Definition 2 (Child elements)

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

2. TDOM is typed.

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

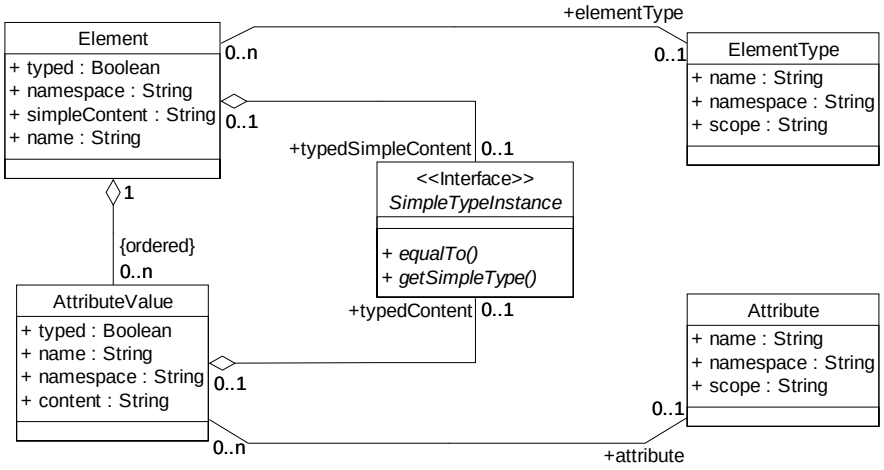


Figure 6: Representation of elements and attribute values (UML class diagram)

In typed representation, elements and attribute values are tightly coupled to the element types and attributes declared in the schema definition accompanying an XML document. According to the class diagram of Figure 6 which unveils more details regarding the representation of elements and attribute values with TDOM, an element or attribute in typed representation (indicated by the boolean attribute `typed` of the classes `Element` and `AttributeValue`) is explicitly associated with the respective element type or attribute it instantiates, i.e., it is valid to. This is expressed by the associations between the classes `Element` and `ElementType` and `AttributeValue` and `Attribute` respectively. Element types and attributes are characterized by their names and namespaces and an optional scope.

Furnishing the classes `ElementType` and `AttributeValue` with the `scope` attribute is a tribute to the fact that MPEG-7 DDL, just like other schema definition languages for XML documents, not only allows to declare element types and attributes that are globally visible

but also those that are only visible within a certain scope, e.g., a complex type. In order to distinguish different element types and attributes with identical names and namespaces that might exist within different scopes of one and the same schema definition, the attribute `scope` contains a string uniquely describing the scope in which the element type or attribute is visible.

The explicit association of elements and attribute values in typed representation with their element types and attributes declared in the schema definition not only provides an index allowing to efficiently look up all instances of a certain element type or attribute in a document. It also opens up type information that is used to acquire an adequate representation of the content of elements and attribute values: elements with simple content and attribute values in typed representation do not keep their content as text, but rather encapsulate their content within an object. This is captured in Figure 6 by the aggregations between the classes `Element` and `AttributeValue` and the interface `SimpleTypeInstance`, which the objects holding the content have to implement as a minimum (we will describe this interface in detail later under point 5). Inside these objects, the content is kept in a way adequate to the content type declared for the element type or attribute in the schema definition. The objects offer methods specific to the content type that allow applications to appropriately access and operate on the content.

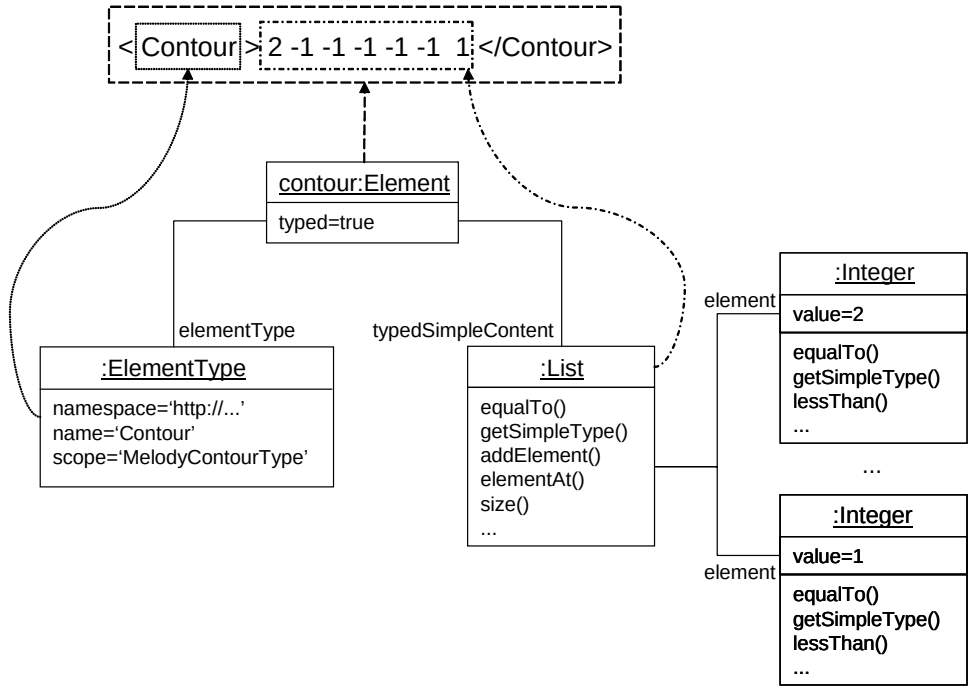


Figure 7: Typed representation of the example `Contour` element (UML object diagram)

In Figure 7, we give an example for a better understanding of typed representations. At the top of the Figure, the `Contour` element of our example MPEG-7 media description of Figure 2 is shown. Below the `Contour` element, the objects used for its representation are depicted in UML object diagram notation. A dashed arrow between an object and the `Contour` element indicates which part of the element is represented by the object. TDOM represents the whole element by an object of the class `Element`. In typed representation, an element is explicitly associated with the element type it instantiates. This is captured in the example by the reference from the `Element` object to the `ElementType` object representing the element type `Contour` that has been declared within the complex type `MelodyContourType` in the media description scheme of Figure 1. It is known from this element type declaration that the valid contents for elements of the type `Contour` are lists of integer values. As the example element is kept in typed representation, its content is thus encapsulated within an object of a class that offers an implementation for lists with reasonable methods to work with them – the class `List`. Since the elements of the list are known to be integer values, they are encapsulated in objects of the class `Integer` providing an implementation for integer values.

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

There are some constraints that have to be obeyed with regard to typed representations though. It must be ensured that the content of an element in typed representation is either simple, i.e., it is represented by an object implementing the interface `SimpleTypeInstance`, or complex, i.e., its content consists of further child nodes via the aggregation between `Element` and `DocumentNode` given in the class diagram of Figure 4, both not both. This is expressed by Constraint 3.

Constraint 3 (Typed element content)

context `Element`

inv: `typed` **implies**

```
(typedSimpleContent -> notEmpty() implies
    childNode -> isEmpty()) and
```

```
(childNode -> notEmpty() implies
    typedSimpleContent -> isEmpty())
```

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

Constraint 4 (Typed root element)

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

3. TDOM needs not to be typed.

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

As a fallback for such situations, TDOM offers the notion of untyped representations. In untyped representation, elements or attribute values are decoupled from the schema definition, not being explicitly associated with the definition's element types or attributes. They maintain the name and namespace of their respective element type or attribute as well as their content in the corresponding textual attributes of the classes `Element` and `AttributeValue` that are depicted in the class diagram of Figure 6 – with all the problems involved related to the appropriate access to the content.

The UML object diagram of Figure 8 illustrates the concept of untyped representations. The diagram once more depicts the `Contour` element taken from our example MPEG-7 media

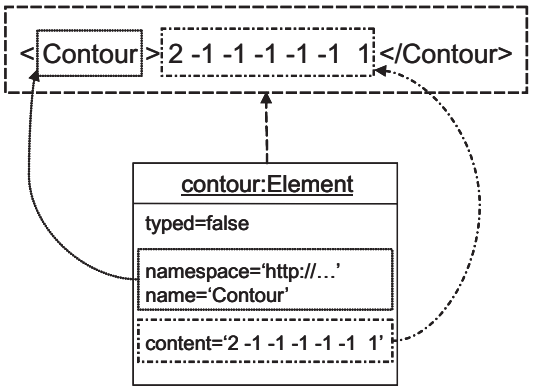


Figure 8: Unyped representation of the example `Contour` element (UML object diagram)

description of Figure 2 – this time, however, in untyped representation. Again, dashed arrows indicate which parts of the object diagram correspond to which part of the element shown at the top of the figure. The encoding of the list of integer values constituting the content of the element in the textual attribute `content` is especially noteworthy. There is no indication for an application that this string represents a list. Without further knowledge, the content of the element can thus only be processed as a string with doubtable usefulness. Even if the application had that knowledge, it would always have to parse the string and cast it to an appropriate internal representation before adequate access to the list of integer values could take place.

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

Constraint 5 (Untyped element content)

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

Moreover, elements and attribute values must be created in a consistent manner: an element or attribute value has to be either in typed or in untyped representation but not in an odd

mixture of both. I.e., an element or attribute value in typed representation should not make use of the attributes of the classes `Element` and `AttributeValue` that are intended for untyped representations and vice versa. This is covered by Constraint 6.

Constraint 6 (Consistency of representations)

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

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

Finally, we are now able to provide the reader with the definition of the formal shorthands `attName` and `attNamespace` that we have used in Constraint 1 to address the name and namespace of the attribute to which an attribute value belongs:

Definition 3 (Attribute name and namespace)

```
context AttributeValue def:
let attName :  String =
```

```
    if typed then
        attribute.name
    else
        name
    endif
let attNamespace :  String =
    if typed then
        attribute.namespace
    else
        namespace
    endif
```

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

Definition 4 (Element type name and namespace)

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

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

We have already mentioned before that MPEG-7 DDL offers constructs, e.g., `<any>` and

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

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

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

Constraint 7 (Untyped representation of elements)

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

5. TDOM supports arbitrary simple types.

From the perspective of MPEG-7 DDL, an object representing the simple content of an element or the content of an attribute value in typed representation constitutes an instance of a simple type. MPEG-7 DDL predefines a comprehensive set of elementary simple types whose instances

may occur as the content of elements and attribute values in MPEG-7 media descriptions, as well as a variety of derivation methods for the definition of new simple types.

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

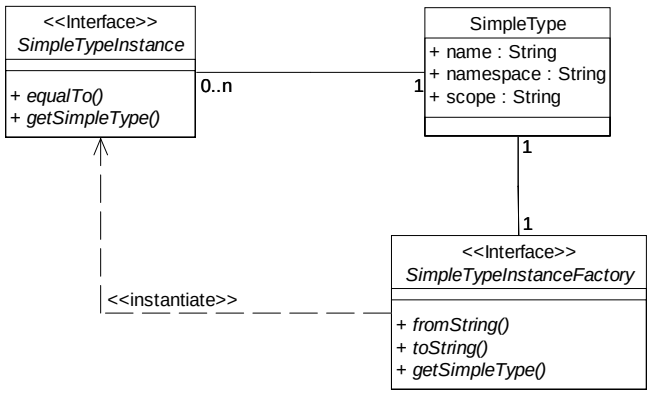


Figure 9: Simple type framework (UML class diagram)

The simple type framework of TDOM is presented in the class diagram of Figure 9. As shown in the diagram, the framework represents simple types by the class `SimpleType`. A `SimpleType` object serves to represent either an elementary simple type predefined by MPEG-7 DDL or a simple type specific to a certain schema definition that has been derived from a predefined simple type using the constructs for type derivation available with MPEG-7 DDL. TDOM attributes a simple type with its name, namespace and an optional scope in which it is visible in a schema definition.

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

which provides basic lookup functionality for simple type instances as it can be used to compare two simple type instances for equality, and `getSimpleType()`, which delivers simple type of the instance. Each simple type keeps track of its instances which is expressed by the association between `SimpleType` and `SimpleTypeInstance`.

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

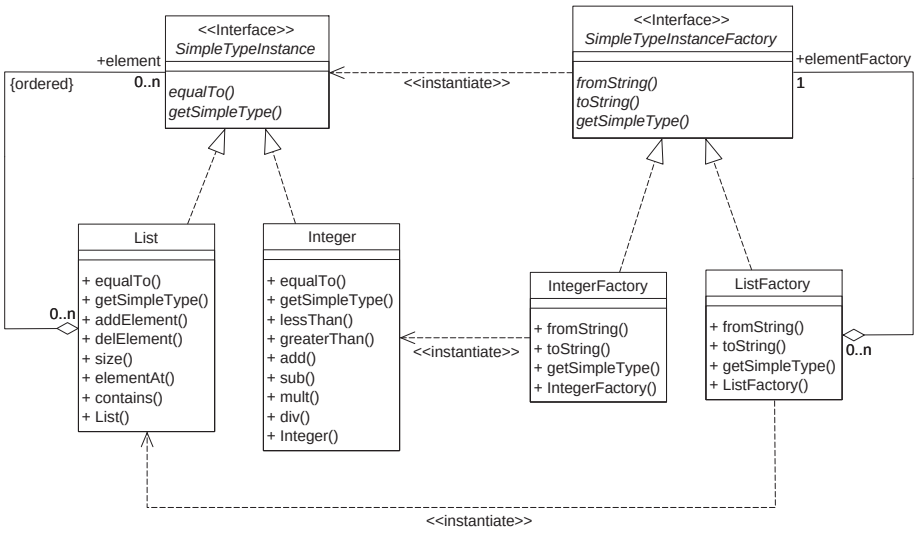


Figure 10: Example implementation of simple type support (UML class diagram)

Figure 10 gives an impression of how the simple type framework can be utilized to support a set of simple types. In our example, support for the simple type integer and its instances is provided. This is achieved by defining the class `Integer` which, beyond type-specific methods,

e.g., for adding and substracting, implements the interface `SimpleTypeInstance` so that its objects are usable as the content of elements and attribute values in typed representation. For the construction of `Integer` objects from the textual representations in which integer values are encoded in XML documents, the class `IntegerFactory` is supplied implementing the Interface `SimpleTypeInstanceFactory`.

Likewise, TDOM can accommodate derivation methods for simple types. Figure 10 exemplifies the integration of a list type with TDOM. Similar to other simple types, the classes `List` and `ListFactory` provide support for the instances of the list type and for their construction by implementing the interfaces `SimpleTypeInstance` and `ListFactory`, respectively. In contrast to elementary simple types such as integer, however, the construction of instances of a derived simple type typically includes the construction of instances of the base type. In our example, the construction of a list includes the construction of instances of the simple type of its elements. Therefore, the factory for the list type must refer to the factory of its base type, modeled by the aggregation between `ListFactory` and `SimpleTypeInstanceFactory`.

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

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

Constraint 8 (Simple type instance)

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

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

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

Constraint 9 describes the interface `SimpleTypeInstanceFactory` in more detail.

Constraint 9 (Simple type instance factory)

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

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

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

successfully constructed from a textual representation that is the result of the call of the method `toString()` on another instance of the same type, then both instances are equal. In other words, `fromString()` constitutes the inverse method to `toString()`.[3] In general, we can say that if calling `toString()` on two instances of the same simple type yields the same textual representation, then both instances are also equal to each other. This is formally described by the first invariant of the class `SimpleType` in the constraint above. Finally, the second invariant of `SimpleType` defines that the result of the call of the method `getSimpleType()` on a simple type instance factory is always the simple type to which the factory belongs.

6. TDOM facilitates flexible, fine-grained updates.

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

To illustrate the benefit of having both typed and untyped representation available, we consider an update on our example media description of Figure 2. An application might want to replace the `Beat` element by a new one. A natural way to perform this task would be the deletion the `Beat` element followed by the insertion of the new `Beat` element as a child of the element `MelodyContour`.

Did TDOM only support typed representations, it would have to be ensured after every single update operation that every element and attribute value affected by the update is valid with respect to the declaration of the particular element type or attribute it is associated with in typed representation. This is very rigid. In our example, the deletion the `Beat` element already violates the validity of the `MelodyContour` element, since, according to the schema definition of Figure 1, an element of type `MelodyContour` must contain exactly one element of type `Beat`. Thus, the deletion and thereby the whole sequence of update operations would have

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

to be refused – even though the subsequent insertion of the new `Beat` element would restore schema consistency.

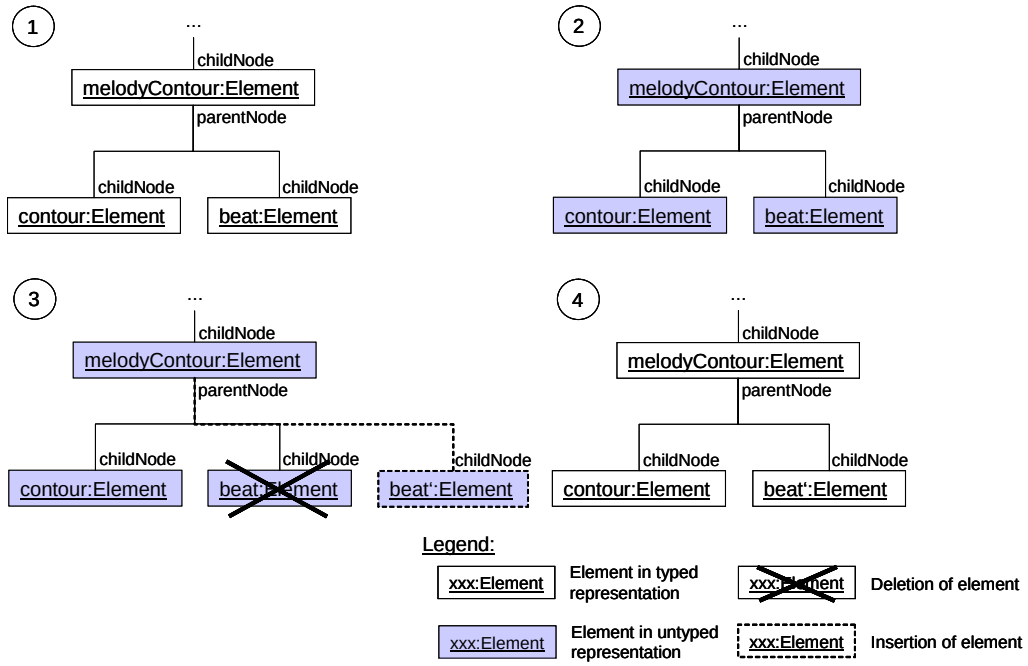


Figure 11: Switching between corresponding representations for an update

But having the additional means of untyped representations at hand (see Figure 11), applications can transform elements and attribute values in typed representation (1) that are affected by an update to a corresponding untyped representation (2). Thereby, they are decoupled from the element types and attributes of the schema definition. Any desired sequence of update operations can then be performed without being concerned with schema validity (3). After all update operations have been completed, the updated elements and attribute values can be brought back to corresponding typed representations (4) as long as the document is still valid with respect to the schema definition.

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

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

Definition 5 (Corresponding representations of attribute values)

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

Definition 6 formally introduces a notion of correspondence for elements.[4] Following that definition, an element `e'` in untyped representation constitutes a corresponding untyped representation of an element `e` in typed representation (formally: `e.CUR(e')`), if `e'` refers to the name and namespace of the element type associated with `e`. If `e` has simple content, it is furthermore demanded that `e'` has simple content as well and the simple content of `e'` is a textual representation of the simple type instance forming the simple content of `e`. If `e` has complex content, however, it is demanded that `e'` also has complex content and the child nodes of `e'` are equal to the child nodes of `e` – with the exception of elements: the child elements of `e'` are expected to be corresponding untyped representations of the respective child elements of `e`. Finally, every attribute value of `e'` must appear among the attribute values of `e` or be

[4] In the definition, we assume the existence of the method `deepEqualTo()` to compare two objects for deep equality.

a corresponding untyped representation of an attribute value of `e`. With all these conditions fulfilled, we can conversely say that `e` constitutes a corresponding typed representation of `e'` (formally: `e'.CTR(e)`).

Definition 6 (Corresponding representations of elements)

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

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

Definition 7 (Untyping attribute values)

context `AttributeValue::untype()`

pre: `self.typed`

post: `self@pre.CUR(self)`

pseudocode:

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

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

Definition 8 (Untyping elements)

context `Element::untype()`

pre: `self.typed`

post: `self@pre.CUR(self)`

pseudocode:

```
-- change all child elements to untyped representation
foreach e1 in self.childNode ->
    select(e2 :  Element | e2.typed) do
        e1.untype()
endforeach
-- change all attribute values to untyped representation
foreach av1 in self.attributeValue ->
    select (av2 | av2.typed) do
        av1.untype()
endforeach
-- change element to untyped representation
self.typed := false
-- get name and namespace of element type from
-- element type definition
ElementType et := self.elementType -> any(true)
self.name := et.name
self.namespace := et.namespace
-- remove reference to element type definition
```

```
self.elementType := self.elementType -> excluding(et)
-- construct textual representation from simple type
-- instance representing potentially existing simple
-- content
if self.typedSimpleContent -> notEmpty() then
    SimpleTypeInstance sti := self.typedSimpleContent ->
        any(true)
    self.simpleContent := sti.simpleType.
        simpleTypeInstanceFactory.toString(sti)
    -- remove reference to simple type instance
    self.typedSimpleContent := self.typedSimpleContent ->
        excluding(sti)
endif
```

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

To this end, we have developed typing automata. A typing automaton constitutes a well-defined, executable representation of the schema and type information carried in a schema definition that is capable of traversing an XML document, inferring the element types and attributes to which the elements and attribute values of the document comply, and obtaining corresponding typed representations of these elements accordingly. We will treat typing automata in detail later in Section 5.

7. TDOM takes account of MPEG-7 DDL.

TDOM has been designed to take advantage of media description schemes written in MPEG-7 DDL which accompany MPEG-7 media descriptions. These can be used to obtain typed representations of elements and attribute values such that the document's basic contents are kept in a fashion appropriate to the respective content type. To facilitate extensive construction of such typed representations for the basic contents that may occur in media descriptions, TDOM furthermore is capable of embracing the plenitude of predefined simple types and simple type derivation methods that come with MPEG-7 DDL via the simple type framework.

Since the purpose of TDOM is to effectively represent media descriptions and not the description schemes to which they comply, however, the detailed representation of an MPEG-7 DDL media description scheme coming with a media description has been left out of the scope of the model. Abstracting from the schema definition language, TDOM just presumes the existence of element types, attributes, and simple types in a schema definition for the modeling of typed representations.

The decision to abstract from the details of the schema definition language has the convenient side effect that it leaves TDOM, though primarily intended for MPEG-7, applicable to other application domains. In other domains, the typed representation of the basic contents of an XML document might also be desirable, but schema definition languages different from MPEG-7 DDL might play dominant roles. As an example taken from the domain of electronic data interchange, the structure of business documents following the XML Common Business Library (xCBL) [60] is defined with the schema definition languages SOX [14] and XDR [20].

In order to be able to validate an MPEG-7 media description against its description scheme and to construct typed representations of the basic contents of the description, an MPEG-7 database solution using TDOM as its data model must, of course, be able to process the description scheme and provide means for its detailed representation. For this purpose, we have complemented TDOM with the already-mentioned typing automata. Typing automata are expressive enough to capture the schema and type information contained in media description schemes written in MPEG-7 DDL but are nevertheless independent of MPEG-7 DDL. They can therefore be used to represent schema definitions for XML documents indited in other schema definition languages as well that might be encountered in different application domains.

5 Typing

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

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

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

The discussion starts with some basic considerations on the problem (5.1). Then, the con-

cept of typing automata as a formal, executable, and language-neutral means for representing the schema and type information carried by MPEG-7 media description schemes is proposed (5.2). Typing automata are capable of inferring and creating typed representations of elements and attribute values. The computational complexity of the behavior of typing automata is examined (5.3) and, in order to reduce the effort necessary for creating typed representations in many practical situations, optimizations are suggested (5.4). This section concludes showing how the basic typing automaton mechanism can be extended, so that even the more complex constructs of MPEG-7 DDL are supported and the expressiveness of that language is reached (5.5).

5.1 Basic considerations

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

Figure 12 intends to get across a presentiment of this. It shows the sample MPEG-7 media description known from Figure 2 in a TDOM representation consisting solely of untyped representations along with the `Melody` media description scheme of Figure 1 to which the description complies. The element type declarations spread all over the description scheme are highlighted. For the first step in constructing typed representations, a TDOM implementation must find out which element types the elements of the media description validly instantiate – i.e., the implementation somehow has to infer exactly those relationships between elements and element types which have been marked by dashed arrows in the figure.

But this inference is difficult: MPEG-7 DDL is a declarative schema definition language. It defines no directly executable algorithm for inferring those declarations in a schema definition that are validly instantiated by a particular element or attribute value. As it can be seen at

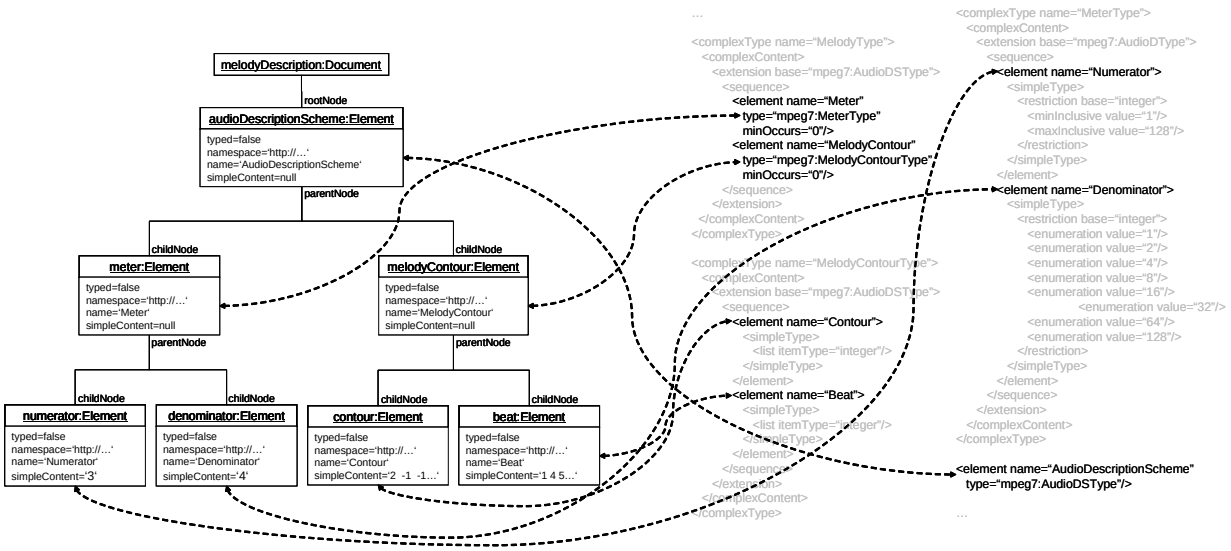


Figure 12: Typing problem

hand of our example `Melody` media description scheme, MPEG-7 DDL furthermore supports highly complex constructs for the structuring of schema definitions such as complex types and complex type derivation that further complicate validation directly on the basis of DDL syntax.

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

In literature, several executable intermediary representations of schema definitions have been proposed for XML document validation. Proposals include rather exotic approaches that translate schema definitions to XSLT stylesheets [7] which transform XML documents to HTML pages highlighting those places inside these documents that are not valid [37]. Another approach is to transform schema definitions to LL(1) grammars [39] which are then fed into standard parser generators used for compiler construction to generate code for specialized parsers specifically tailored to these schema definitions. Further approaches use various kinds of formal automata for the intermediary representation of schema definitions, such as finite state automata [50] (which can cover a restricted subset of non-recursive schema definitions

only due to their limited expressiveness), pushdown automata [50], and regular tree automata [6, 42, 44, 47, 24]. The latter have heavily inspired the design of several schema definition languages such as TREX [8] and RELAX-NG [10].

With regard to our typing problem, the adoption of regular tree automata as means for the intermediary representation of schema and type information conveyed in MPEG-7 media description schemes is especially attractive: regular tree automata essentially reduce the problem of validating an XML document to the problem of successively evaluating string regular expressions. The evaluation of string regular expressions is well-understood and there exists a broad variety of highly efficient software libraries for this purpose. Apart from the fact that these libraries not only simplify the implementation of regular tree automata in practice, most of these libraries – for instance, libraries that support Perl 5 regular expressions – additionally offer powerful extensions to traditional regular expressions that prove useful to cope with more complex constructs of MPEG-7 DDL. Regular tree automata also have manageable computational complexity: it is known that a deterministic regular tree automaton consumes a tree with a running time linear to the number of tree nodes [11]. Last but not least, regular tree automata permit a natural and intuitive representation of MPEG-7 media description schemes as we will see.

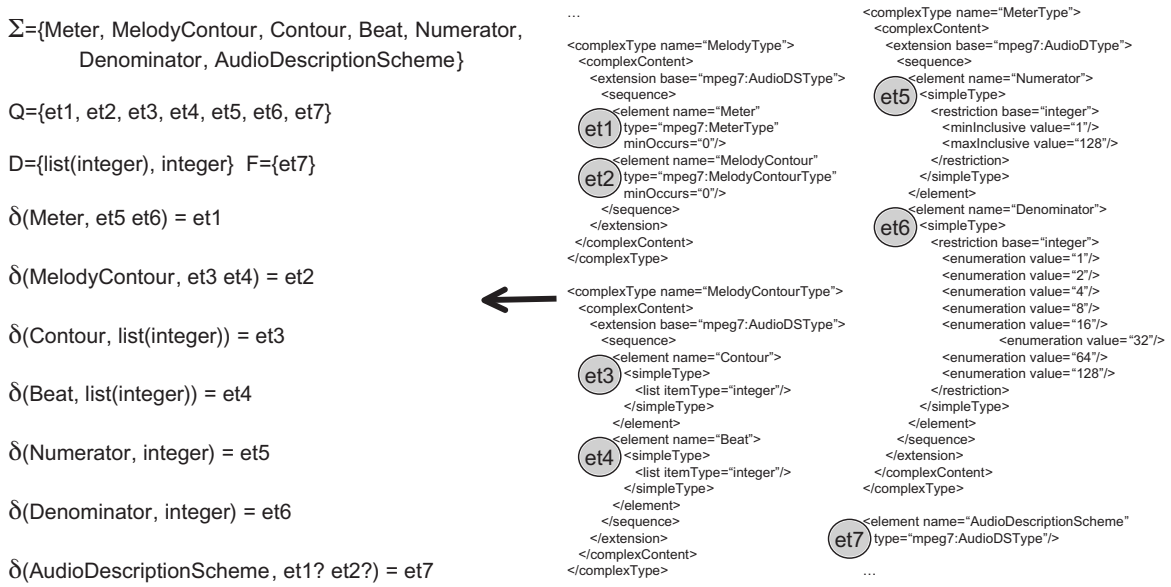


Figure 13: Example tree automaton

To give an impression of regular tree automata and how they can serve as a means for

the intermediary representation of media description schemes, Figure 13 depicts the example `Melody` media description scheme (depicted on the right-hand side of the figure) represented as a bottom-up regular tree automaton[5] (depicted on the left-hand side). The figure employs the notation for regular tree automata introduced in [6].

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

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

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

As Figure 13 exemplifies, the formal mechanism of regular tree automata can be utilized for the representation of the `Melody` media description scheme in a straightforward manner. Every element type declared in the media description scheme is given a textual label ($\mathit{et1}, \ldots, \mathit{et7}$ in this case) which is indicated in the figure by grey circles next to the declarations. These

[5]Note that literature also knows of top-down regular tree automata [11]. Since these classes are generally equivalent to each other, our limitation to bottom-up regular tree automata implies no loss of generality.

labels make up the set of states Q. The idea is that, while consuming an XML document from the bottom up, the tree automaton applies to an element exactly the labels of those element types that are validly instantiated by the element. The labels of all unscoped element types in the description scheme make up the set of final states F. Whenever the tree automaton attaches one of these states to the root element, the document is considered valid because the root element correctly instantiates a globally visible element type declaration.

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

Finally, every element type declaration in the description scheme is translated to a corresponding transition rule of the tree automaton. Each of these transition rules takes the name of the declared element type as the first argument and yields the element type's label as its result. Depending on whether the content of the element type is declared as simple or complex, a transition rule of the first or second variant is created. For element types with simple content, the second argument of the transition rule is the simple type used for the content declaration. For element types with complex content, the content model is translated to an equivalent regular expression based on the labels of those element types that occur in the content model. For instance, the content model of the element type `MelodyContour` (which is labeled *et2*) consisting of a sequence of elements of types `Contour` and `Beat` (labeled *et3* and *et4*, respectively) is translated to the regular expression *et3 et4*. The regular expression created in this manner consitutes the second argument of the transition rule.

Figure 14 illustrates the consumption of our example MPEG-7 media description in TDOM representation by the constructed bottom-up regular tree automaton. Beginning at the leaf elements of the description, the automaton ascends through the tree structure as indicated by the dashed arrows. For every element, the automaton fires as much transition rules as possible. The figure shows the states yielded for the different elements of the media description as grey circles. As the applicable states are labels representing the different element type declared in the

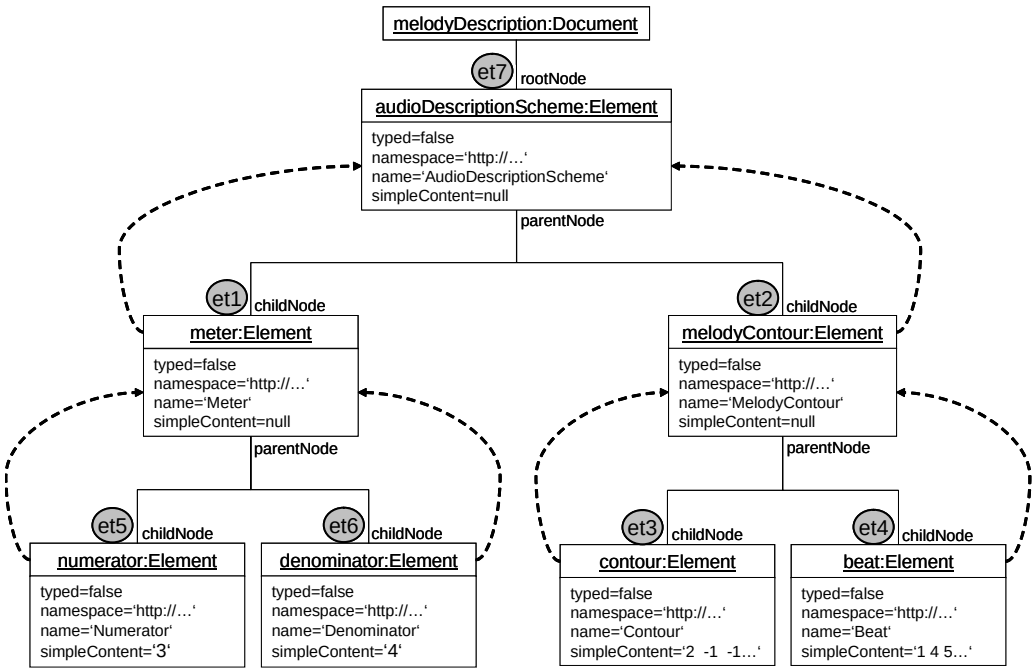


Figure 14: Tree automaton application (UML object diagram)

media description scheme, the automaton thus infers exactly those types that are instantiated by the respective elements. Since the label *et7* attached to the root element refers to the globally visible element type `AudioDescriptionScheme`, the automaton has also found that the media description as a whole is valid with regard to the `Melody` media description scheme.

5.2 Typing automata

As we have seen, bottom-up regular tree automata provide an intuitive formal foundation for the intermediary, language-neutral representation of MPEG-7 media description schemes. Their execution behaviour permits the inference of the element types and attributes that are instantiated by the elements and attribute values in an MPEG-7 media description. For our aim of constructing corresponding typed representations of elements and attribute values in untyped representation within TDOM, we have therefore decided to pick up that mechanism and to develop it further to what we call typing automata.

At their core, typing automata still constitute bottom-up regular tree automata. However, they have been remodeled in an object-oriented fashion in order to be compatible and seamlessly applicable to XML documents represented with TDOM. During remodeling, special care

has been taken to keep typing automata extensible so that they can reach the expressiveness of MPEG-7 DDL and are thus suitable for the representation of arbitrary MPEG-7 media description schemes. Further exceeding the functionality of regular tree automata, typing automata are not only capable of validating XML documents and finding the element types and attribute values instantiated by the elements and attribute values of an XML document; they are additionally able to produce corresponding typed representations of the elements and attribute values on the basis of these inferred element types and attributes in a second processing phase.

In the following, we introduce and formally specify typing automata by means of UML and OCL. We begin by providing some basic definitions and by specifying the overall structure of typing automata (5.2.1). For simplicity, we neglect the existence of attributes and attribute values in the ensuing definitions (we will come back to attributes and attribute values and how they can be incorporated into typing automata later in Section 5.5). We then specify the behavior of typing automata when they are applied to TDOM-represented XML documents. We have broken down the behavioral specification into two phases: the validation phase (5.2.2), in which the element types instantiated by the elements of an XML document are inferred, and the typing phase (5.2.3), in which elements in untyped representation are brought to corresponding typed representations accordingly.

5.2.1 Structure

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

Definition 9 serves exactly that purpose. It introduces the formal shorthand `etID` which delivers a textual identifier for an element type consisting of four parts separated by the delimiter `"::"`: the first part is always the string `"et"` indicating that the ID refers to an element type. The second part consists of the scope the element type, followed by the namespace and the name of the element type as the third and fourth part. The inclusion of the scope, namespace, and name of an element type into its ID has the advantage that these data can be accessed

within string regular expressions. As we will see later, this facilitates the implementation of more complex constructs supported by MPEG-7 DDL on the basis of regular expressions.

Figure 15: Example element type IDs (UML object diagram)

Figure 15 illustrates element type IDs by showing the element types occurring the example `Melody` media description scheme represented as instances of `ElementType` together with their IDs.

Definition 9 (Element type ID)

context `ElementType` **def:**

let `etID : String =`

```
    "et::".concat(scope.concat("::".concat(
        namespace.concat("::".concat(name)))))
```

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

Definition 10 (Unknown Element type ID)

context `Element` **def:**

let `uetID : String =`

```
"uet::null::".concat(etNamespace.concat("::".concat(etName)))
```

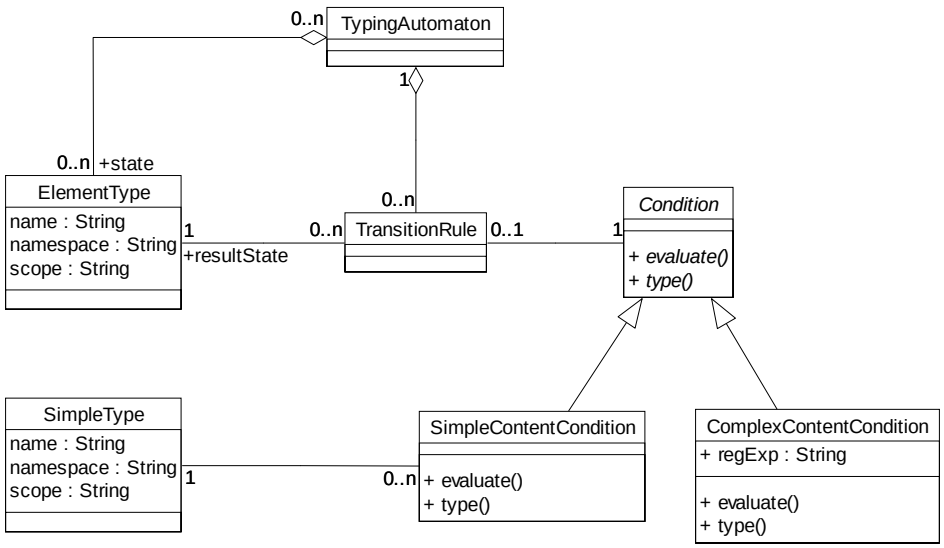


Figure 16: Typing automaton structure (UML class diagram)

After these preliminaries, we can now provide the specification of typing automata. Figure 16 defines the structure of a typing automaton by means of an UML class diagram. As it can be

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

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

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

For the beginning, we restrict ourselves to the expressiveness of bottom-up regular tree automata. We introduce two kinds of conditions, namely simple content conditions and complex content conditions represented by the classes `SimpleContentCondition` and `ComplexContentCondition`, respectively. Essentially, a simple content condition refers to a simple type which is represented by the class `SimpleType` of TDOM's simple type framework. The condition is fulfilled if an element has simple content and if this simple content is a valid instance of the simple type referenced. A complex content condition consists of a Perl 5 string regular expression (kept in the attribute `regExp`) and is fulfilled if an element has complex content and the concatenation of the IDs of the element types applicable to the element's child elements satisfy the regular expression. Hence, both variants of transition rules that are offered by traditional regular tree automata can be expressed using simple content conditions

and complex content conditions within a typing automaton as well.

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

Constraint 10 (Typing automaton)

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

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

We conclude the structural definition of typing automata with an example. The UML object diagram of Figure 17 depicts all transition rules of a typing automaton capturing our example `Melody` media description scheme. Just as with the regular tree automaton of Figure 13, a corresponding transition rule has been constructed for every element type declaration contained in the scheme. Each transition rule refers to the corresponding element type declared as its result state. Depending on whether the element type declaration defines a simple or complex content model, simple content conditions or complex content conditions have been created appropriately. Due to limitations of space, we refrain from using the full element type IDs as given by Figure 15 within the regular expressions of complex content conditions. Instead, we use `‘et.etID‘` as a placeholder for the ID of element type `et`.

5.2.2 Validation phase

Having specified the structure of typing automata, we are now able to continue with the specification of their behavior. As already mentioned, the consumption of an XML document in TDOM representation by a typing automaton proceeds in two phases. During the first of these

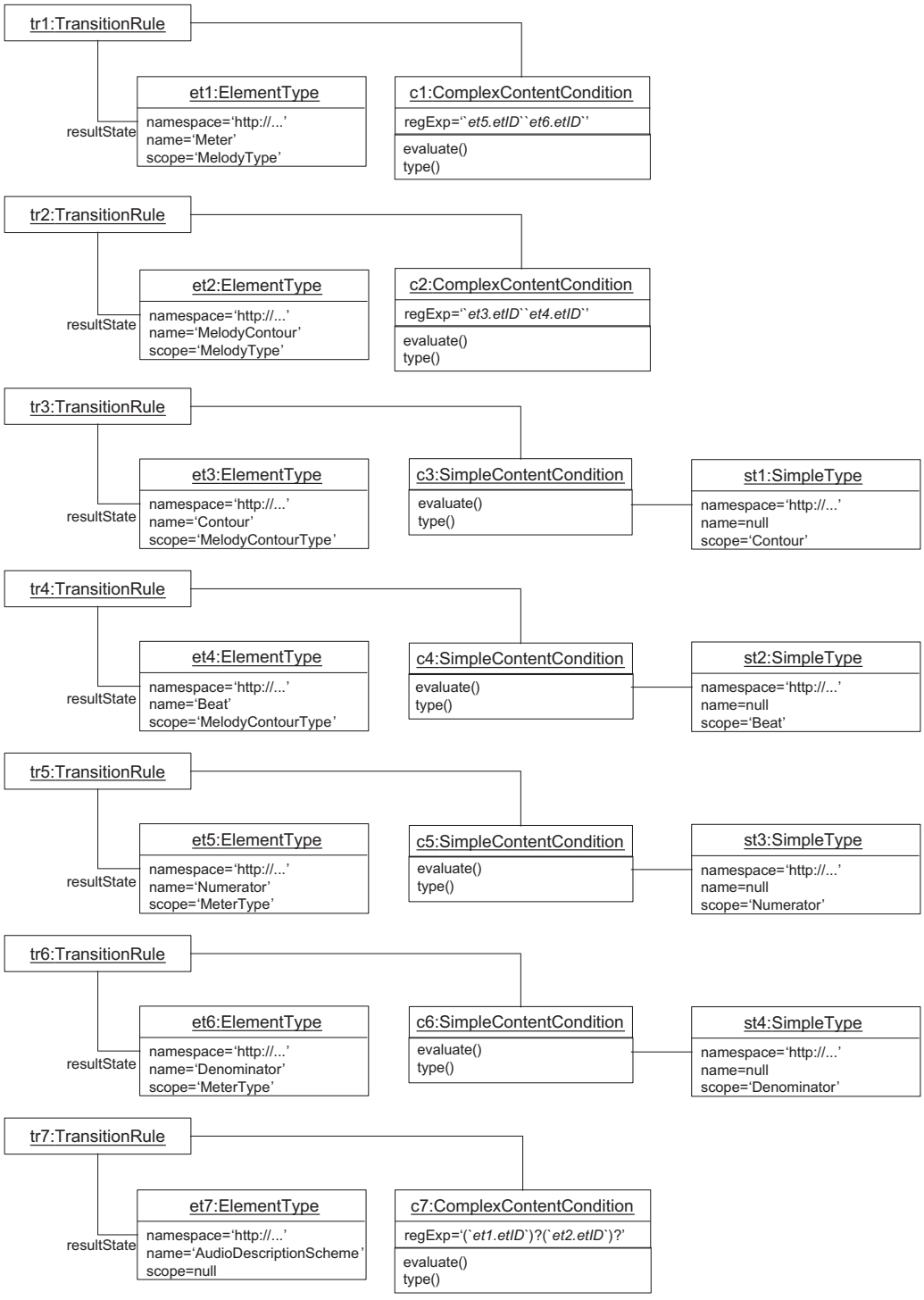


Figure 17: Example transition rules (UML object diagram)

phases, the validation phase, the element types which the elements of the document validly instantiate are inferred. A typing automaton does this in a way similar to bottom-up regular tree automata: the automaton attempts to apply all transition rules to each of the document's

elements. The element types serving as the result states of all those transition rules that are applicable to a given element are called the element's applicable element types.

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

Definition 11 (Applicable element types)

```
context TypingAutomaton def:
let applicableElementTypes(Element e) :  Set(ElementType) =
    states -> select(et | transitionRule -> exists(tr |
        et = tr.resultState and
        e.etName = et.name and
        e.etNamespace = et.namespace and
        tr.condition.evaluate(e, tr)))
```

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

Definition 12 (Valid document)

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

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

Definition 13 specifies the behavior of the method `evaluate()` for the class `SimpleContentCondition`. The method checks whether an element has simple content and

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

Definition 13 (Evaluation of simple content condition)

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

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

argument has simple content. If the element does not have simple content, `evaluate()` returns `true` if and only if there exists a sequence of element types applicable to the element's child elements for which holds that the sequence's signature, i.e., the concatenation of the element type IDs in the sequence, matches the condition's regular expression.[6]

Definition 14 (Evaluation of complex content condition)

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

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

Definition 15 (Applicable child element types)

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

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

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

Definition 16 (Signature)

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

One might get the impression that due to the mutually recursive definition of `applicableElementTypes()` and `applicableChildElementTypes()` via the indirection of the

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

5.2.3 Typing phase

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

Figure 18 illustrates the typing phase of a typing automaton using our example MPEG-7 `Melody` media description. Beginning at the root element, the typing automaton selects an applicable unscoped element type for the root element – `et7` in this case as it is the only one available – and uses this element type to bring the root element into a corresponding typed representation (1). Having transformed the root element to typed representation, the automaton proceeds with the root's child elements and selects one of their applicable element types to produce corresponding typed representations as well (2). In that fashion, the automaton continues on descending down the document (3) until the leaf elements of the document have been reached and transformed to corresponding typed representations (4).

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

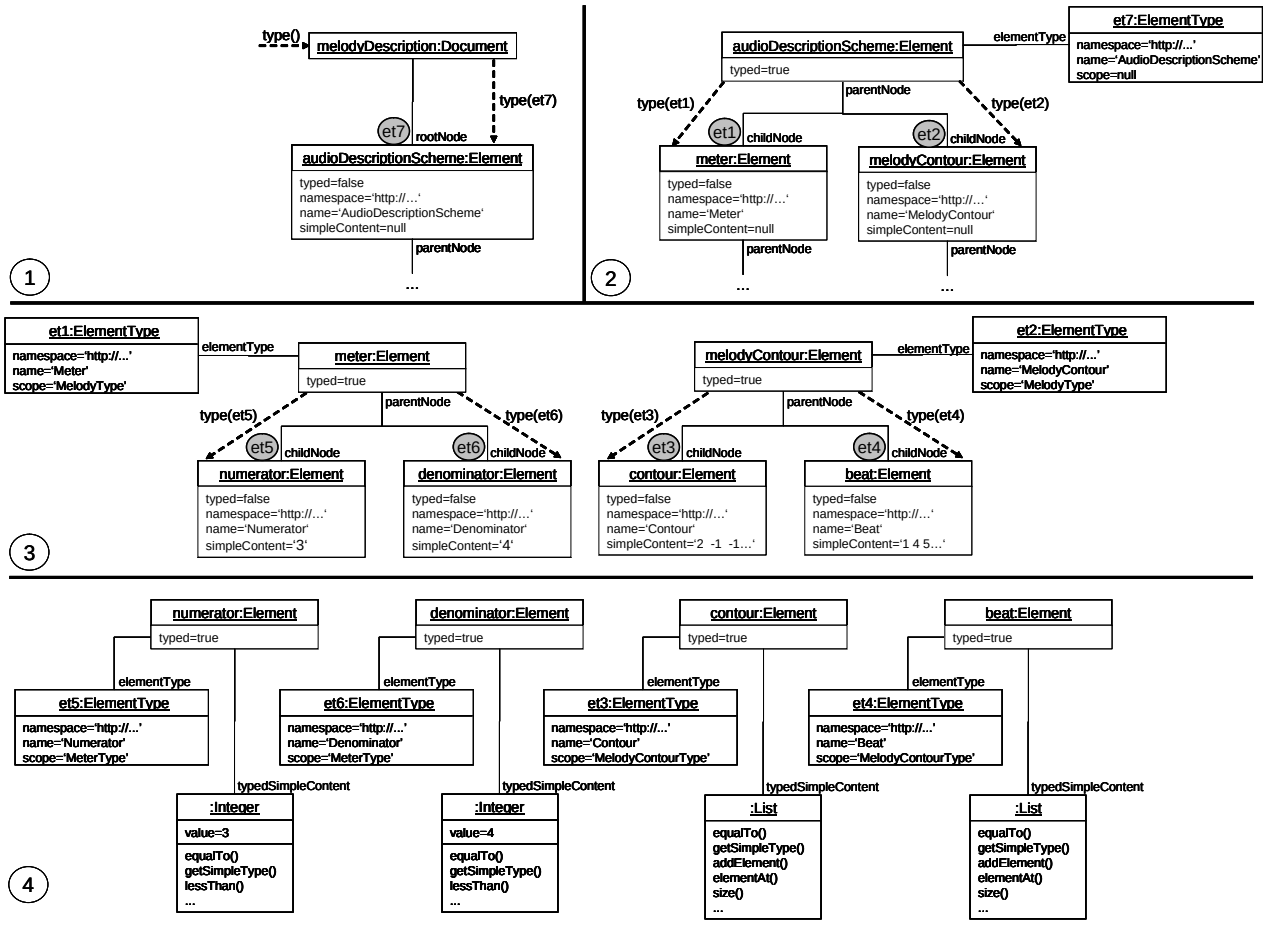


Figure 18: Typing phase (UML object diagrams)

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

The pseudocode given in the definition proposes a simple algorithm for the implementation of this method that consists of two major steps: in the first step, a transition rule of the typing

automaton is selected that has decided in the validation phase that `et` is applicable to `e`. I.e., the chosen transition rule must return `et` as its result state and its condition must evaluate to `true` for `e`. In the second step, the element is passed on to the `type()` method of the transition rule's condition which brings it into a corresponding typed representation in a way that depends on the particular kind of condition.

Definition 17 (Typing elements)

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

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

applicable element types of `e`'s child elements which has already been done in the validation phase as well.

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

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

Definition 18 (Typing functionality of Conditions)

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

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

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

Definition 19 (Typing elements with simple content)

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

```
e.typed := true
e.name := null
e.namespace := null
e.elementType := e.elementType -> including(tr.resultState)
-- Use the simple type instance factory associated
-- with the simple type of the condition to produce
-- an appropriate simple type instance for use as
-- typed element content
stif := self.simpleType.simpleTypeInstanceFactory
sti := stif.fromString(e.simpleContent)
-- Set simple type instance as simple content of element
e.simpleContent := null
e.typedSimpleContent := e.typedSimpleContent
    -> including(sti)
```

Definition 20 covers an algorithm for the implementation of the `type()` method for complex content conditions. The algorithm transforms elements with complex content, that are in untyped representation and on which a given complex content condition evaluates to `true`, into a suitable corresponding typed representation. Similar to the `type()` method for simple content conditions, the implementation first changes the basic structure of the passed element to typed representation. Then, `type()` chooses a sequence of element types applicable to the element's child elements that satisfies the condition, i.e., whose signature matches the Perl 5 string regular expression of the condition. Again, this is always possible. The `evaluate()` method already checks the existence of such a sequence. At last, the child elements of the element passed as the method's parameter are brought to corresponding typed representations in a way that the complex content condition remains satisfied. This is achieved by synchronously iterating over the chosen sequence of applicable child element types and the sequence of child elements. In case that the current member of the sequence of applicable child element types is an element type, the method uses this element type to bring the corresponding member in the sequence of child elements into typed representation by recursively calling the typing automaton's `typeElement()` method.

Definition 20 (Typing elements with complex content)

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

```
                acet -> at(i))
        endif
    endforeach
```

The proposed implementation of the `type()` method once more contains a possible source of inefficiency. Selecting a suitable sequence of applicable element types for the child elements of the passed element such that the complex content condition evaluates to `true` for that sequence is inefficient if implemented naively: this calculation has already taken place during the typing phase and repetition of that calculation implies the repeated inference of the applicable element types of the child elements.

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

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

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

document be invalid with regard to the typing automaton, `type()` transforms all elements of the document to untyped representation.

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

Definition 21 (Typing documents)

```
context TypingAutomaton::type(Document d)
post:   self.valid(d) implies
            Element.allInstances -> forAll(e |
                e.document = d and not(e.CUR(e@pre) and
                self.applicableElementTypes(e) -> includes(e.elementType)) implies
                    (e.typed@pre and e.typed and
                    self.applicableElementTypes(e) -> includes(e.elementType)) or
                    (not(e.typed) and
                    self.applicableElementTypes(e) -> isEmpty()) or
                    (not(e.typed) and e.parentNode -> notEmpty() and
                    not(e.parentNode.typed)))
post:   not(self.valid(d)) implies
            not(d.rootElement.typed)
pseudocode:
    -- Untype document if root element is typed
    if d.rootElement.typed then
        d.rootElement.untype()
    endif
    -- Select an arbitrary applicable unscoped element type to type
    -- root element with if possible.
```

```
if self.valid(d) then
    rootElementType := self.applicableElementTypes(d.rootElement)
        -> any(et | et.scope = null)
    -- Type root element
    self.typeElement(d.rootElement, rootElementType)
endif
```

5.3 Computational complexity

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

For the estimation, we make two assumptions: firstly, we assume that a typing automaton caches the element types applicable to the elements of the document during the validation phase along with the transition rules that decided applicability and the sequences of child element types what were applied to these transition rules. We have already suggested this in Section 5.2.3 for the implementation of the `typeElement()` and `type()` methods of the classes `TypingAutomaton` and `ComplexContentCondition`. Such a caching ensures that applicable element types only need to be calculated once for each element and that the complexity of repeated access to the cached results of this calculation is negligible for our estimation, i.e., $O(1)$, if applying a suitable hashing technique.

Secondly, we assume that a typing automaton is deterministic, i.e., the number of applicable element types for each element is at most one. This restriction does not imply a loss of generality. It has been proven in literature that the classes of non-deterministic and deterministic bottom-up regular tree automata are equivalent to each other [6, 11]: for each non-deterministic bottom-up regular tree automaton an equivalent deterministic one can be algorithmically constructed. Given the structural similarity between bottom-up regular tree automata and typing automata – both types of automata support the same kinds of transition rules and the mapping between

them is straightforward as we have illustrated by means of Figures 13 and 17 – this result also applies to typing automata.

Besides, unambiguousness is a natural quality criterion for schema design. It is no surprise that most schema definitions for XML documents occurring in practice are intuitively designed to be unambiguous and thus straightforwardly translate to deterministic typing automata. The example `Melody` media description scheme of Figure 1 and its typing automaton representation given by Figure 17 perfectly illustrate this point.

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

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

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

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

$$U_{e_i} = O(1), i = 1 \ldots n \tag{2}$$

Similarly, the production of a corresponding typed representation of a single element e_i on the basis of an applicable element type mainly involves changes to the `Element` object representing e_i that are independent of n and t. Moreover, since we assume a caching of applicable element types, the selection of the particular applicable element type and transition

rule that is used for the creation of the corresponding typed representation constitutes an effort that is independent of n and t as well. Therefore:

$$C_{e_i} = O(1), i = 1 \ldots n \tag{3}$$

Inserting Equations 2 and 3 into Equation 1 yields:

$$T(n,t) = \sum_{i=1}^{n} O(1) + \sum_{i=1}^{n} A_{e_i} + \sum_{i=1}^{n} O(1) = O(n) + \sum_{i=1}^{n} A_{e_i} \tag{4}$$

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

In case that a transition rule has a complex content condition, the test for the transition rule's applicability mainly involves checking whether e_i has complex content and evaluating a string regular expression on the signature of the sequence of applicable child element types of e_i (see Definition 14). As we assume that the typing automaton is deterministic, there exists only one such signature. It is a well-known fact that the evaluation of string regular expressions on a string takes linear time with regard to the length of the string [1]. Since the length of the signature of the sequence of applicable child element types of e_i is roughly proportional to the number of e_i's child elements c_{e_i}, the running time for checking a complex content condition should not exceed $O(c_{e_i})$ in practice.

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

$$A_{e_i} = t \max(O(c_{e_i}), O(1)) = t\ O(c_{e_i}), i = 1 \ldots n \tag{5}$$

Inserting Equation 5 into Equation 4 we obtain:

$$T(n,t) = O(n) + t \sum_{i=1}^{n} O(c_{e_i}) \tag{6}$$

As in general $\sum_{i=1}^{n} O(f_i(n)) = O(\sum_{i=1}^{n} f_i(n))$ [12], Equation 6 becomes:

$$T(n,t) = O(n) + t\ O(\sum_{i=1}^{n} c_{e_i}) \tag{7}$$

Since the only element among the n elements contained in an XML document that is not a child element of another and that is thus not covered by the sum $\sum_{i=1}^{n} c_{e_i}$ is the root element, it follows that $\sum_{i=1}^{n} c_{e_i} = n - 1$. Hence, Equation 7 can be rewritten as:

$$T(n,t) = O(n) + t\ O(n-1) = O(t\ n) \tag{8}$$

Given the bound of Equation 8, we can state that, subject to our preliminary assumptions concerning caching and determinism, the running time of the `type()` method of the class `TypingAutomaton` never grows more than linearly with the number of transition rules t of the typing automaton on which the method is executed. Its running time also never grows more than linearly with the number of elements n in the XML document that is passed to `type()`. Given this linearity, we conclude that a typing automaton's behaviour can be considered sufficiently efficient to allow the application of even complex typing automata with large numbers of transition rules to large XML documents.

5.4 Optimizations

Having discussed of the computional complexity of the behavior of typing automata, it is now time to spend some thoughts on possible optimizations. The proposed algorithm for the `type()` method of the class `TypingAutomaton` (see Definition 21), which validates a document and transforms as much elements as possible to corresponding typed representations, is suboptimal in many practical cases. The algorithm behaves reasonably well when applied to an XML document whose TDOM representation consists of elements in untyped representation only. This is typically the case during the import of a document to TDOM where usually a TDOM representation based solely on untyped representations is produced as a first step, before applying a typing automaton in order to validate the document and to create typed representations.

In such a situation, the algorithm traverses the elements contained in the document from the bottom up in order to calculate their applicable element types while checking the document's validity. Assuming a caching as described above, the algorithm then traverses the elements from the top down for a second time and brings them to corresponding typed representations.

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

Therefore, we want to propose an alternative implementation of the `type()` method of `TypingAutomaton` that makes use of already existing typed representations of elements. We call this variant local document typing as it aims at limiting the effects of bringing the document's elements from untyped to corresponding typed representations to the immediate vicinity of the elements in untyped representation. It attempts to preserve already existing typed representations of elements thereby significantly reducing the number of elements that need to be traversed for the purpose of document typing in many cases occurring in practice.

Figure 19 illustrates the different steps of the local document typing approach at hand of an excerpt of the example `Melody` media description known from Figure 2 in TDOM representation. The excerpt covers the `Meter` element and its child elements. It is assumed that all elements of the media description are in typed representation with the exception of the `Numerator` element which is in untyped representation because its simple content has been changed to `5` during an update operation. It is now intended to bring as much elements of the description as possible, especially the `Numerator` element of course, to appropriate typed representations by employing

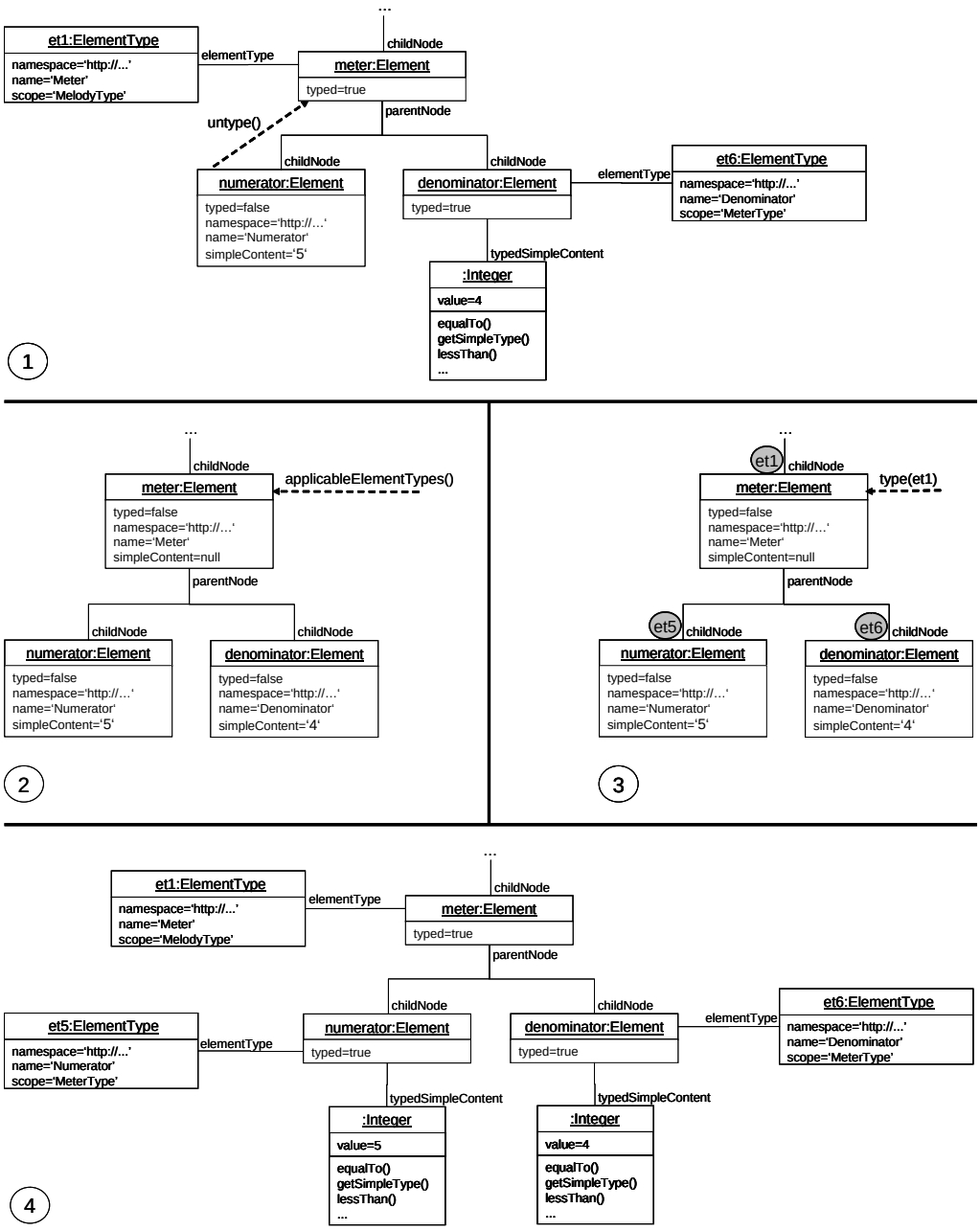


Figure 19: Local document typing (UML object diagrams)

the typing automaton that represents the `Melody` media description scheme with transition rules as given by Figure 17.

To achieve that aim, local document typing starts the typing process at the topmost untyped elements of the document. These are all those elements that are in untyped representation but whose parent elements are in typed representation or, in case that the root element is in untyped

representation already, the root element itself. The topmost untyped elements subsume all elements of the document that need to be brought to corresponding typed representations as their direct and indirect child elements. They further constitute the boundary to those parts of the document which already are in typed representation and which ideally should be affected by the typing process as little as possible.

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

Figure 20 illustrates the behaviour of local document typing in case that the memorized element type of a topmost untyped element's parent element is no longer applicable. For this purpose, we assume that the simple content of the `Numerator` element has been set to the nonsense-string `invalid` instead of `5` during the update operation. In the beginning, local document typing proceeds as usual by memorizing the element type of the parent of the `Numerator` element (1) and by transforming the parent element to a corresponding untyped representation (2). When determining the applicable element types of the parent element, however, we find that the memorized element type is no longer applicable (3). The parent element even does no longer have any applicable element types. This is due to the fact that the `Numerator` element has no applicable element types (the only transition transition rule of the typing automaton potentially suitable for the `Numerator` element, `tr5`, expects integer content according to Figure 17) and thus transition rule `tr1` of the typing automaton is no longer satisfied by the parent element as well. As a consequence, the implicit assumption underlying the typed representa-

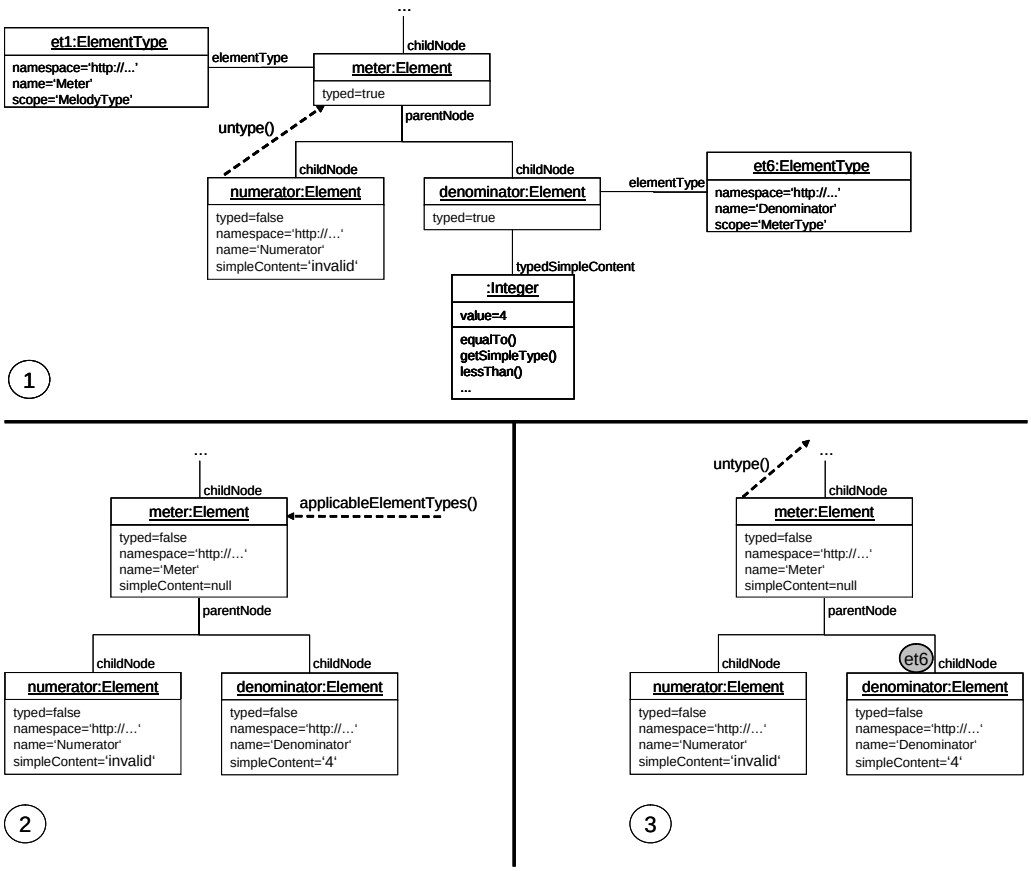


Figure 20: Failing local document typing (UML object diagrams)

tion of the element located above the parent element in the document hierarchy that the parent element constitutes a valid instantiation of the element type `Meter` does not hold anymore.

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

In the following, we specify an alternative implementation of the method `type()` of the

class `TypingAutomaton` that realizes the local document typing algorithm. As a prelimary, we first formalize the notion of the topmost untyped elements of a document in Definition 22.

Definition 22 (Topmost untyped elements)

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

Definition 22 provides the pseudocode describing the new implementation of the `type()` method. Throughout the implementation, a set of elements that are to be brought to typed representation is maintained. This set is initialized with the topmost untyped elements of the document. As long as there are still elements in this set and the document has not been found to be invalid, one of these elements is selected successively. For each selected element, the implementation distinguishes whether the element constitutes the root element of the document or not. In case that the selected element is the root element, the implementation behaves exactly like the conventional implementation of the `type()` method of Definition 21: it chooses an applicable unscoped element type and brings the root element to a corresponding typed representation accordingly. If such an element type exists, typing of the document is finished; if not, the document is considered invalid and processing terminates.

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

If the stored element type is no longer applicable to the parent element, the parent element remains in untyped representation. It is further added to the set of elements to be brought to

typed representation. Again, its child elements are removed from this set as well, since their typing will be covered by the typing of the parent element.

Definition 23 (Local variant of document typing)

```
context TypingAutomaton::type(Document d)
post:   self.valid(d) implies
            Element.allInstances -> forAll(e |
                e.document = d and not(e.CUR(e@pre) and
                self.applicableElementTypes(e) -> includes(e.elementType)) implies
                    (e.typed@pre and e.typed and
                    self.applicableElementTypes(e) -> includes(e.elementType)) or
                    (not(e.typed) and
                    self.applicableElementTypes(e) -> isEmpty()) or
                    (not(e.typed) and e.parentNode -> notEmpty() and
                    not(e.parentNode.typed)))
post:   not(self.valid(d)) implies
            not(d.rootElement.typed)
pseudocode:
    -- Assume valid document
    invalid := false
    -- Retrieve the elements that need to be brought to typed
    -- representation
    toType := d.topmostUntypedElements
    -- As long as the document is not invalid, iteratively
    -- select one these elements
    while toType -> notEmpty() and not(invalid) do
        -- Select arbitrary element for typing
        element := toType -> any(true)
        if element = d.rootElement then
            -- The root element needs to brought to typed representation.
            -- Use applicable unscoped element type for that purpose
            if not(self.applicableElementTypes(element)
```

```
        -> exists(et | et.scope = null)) then
        -- As there is no such element type, the document
        -- is invalid
        invalid := true
    else
        -- Create corresponding typed representation of the
        -- root element using the unscoped element type.
        rootElementType := self.applicableElementTypes(element)
            -> any(et | et.scope = null)
        self.typeElement(element, rootElementType)
        -- Typing of the document is finished
        toType := Set{}
    endif
else
    -- Memorize the type of the chosen element's parent
    parentType := element.parentNode.elementType
    -- Bring parent element to untyped representation
    element.parentNode.untype()
    if self.applicableElementTypes(element.parentNode)
        -> includes(parentType) then
        -- As the memorized element type is still applicable,
        -- bring the parent element to a corresponding typed
        -- representation on the basis of that type
        self.typeElement(element.parentNode, parentType)
        -- Ignore all of the parent's child elements for the
        -- further creation of typed representations.
        toType := toType -> excludingAll(
            element.parentNode.childNode)
    else
        -- The memorized element type is no longer applicable.
        -- Add the parent element to the set of elements that
        -- are to be brought to typed representation.
```

```
                toType := toType -> including(element.parentNode)
                -- Ignore all of the parent's child elements for
                -- the further creation of typed representations
                toType := toType -> excludingAll(
                    element.parentNode.childNode)
            endif
        endif
    endwhile
```

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

typed representations.

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

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

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

5.5 Extensions

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

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

We start out by examining the representation of any, repeated, and empty content declarations within typing automata (5.5.1). We then investigate the representation of mixed content declarations (5.5.2) as well as complex type declarations (5.5.3). Finally, we explore how attribute declarations can be covered within a typing automaton (5.5.4).

5.5.1 Any, repeated, and empty content declarations

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

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

Since the namespace of an element type is an integral part of its ID, complex content conditions are also suited to model occurrences of `<any>` that further restrict the

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

content model of an element type to a certain namespace. For instance, a transition rule with a complex content condition containing the string regular expression `((et::.*|uet::null)::http://www.example.org::.*)*` can be used to limit the contents of the elements of a given type to elements with types that originate from the namespace `http://www.example.org`.

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

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

5.5.2 Mixed content declarations

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

automata can be brought to reject mixed content whenever it is not permitted.

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

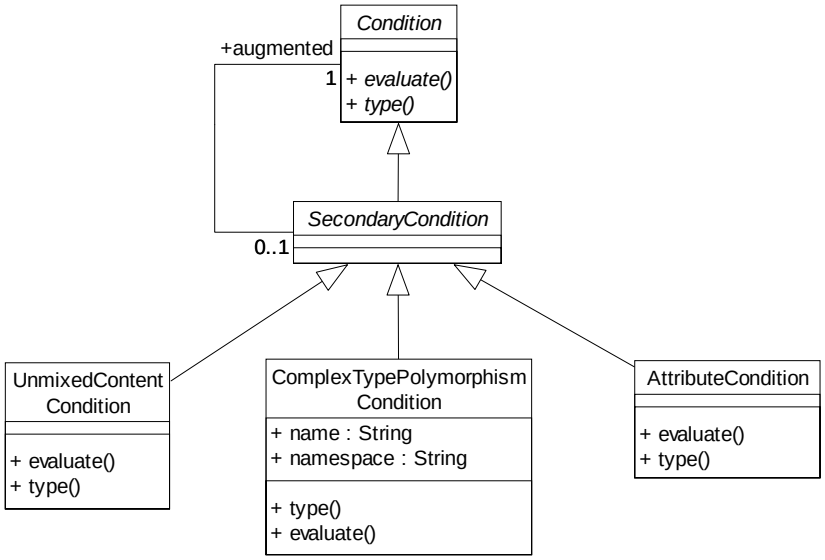


Figure 21: Secondary conditions (UML class diagram)

The basic idea of secondary conditions is shown in the class diagram of Figure 21. The diagram introduces an abstract class `SecondaryCondition` as a subclass of `Condition`. This class subsumes all secondary conditions, among others the class `UnmixedContentCondition` for unmixed content conditions. The association between `SecondaryCondition` and `Condition` ensures that a secondary condition always augments another condition, which may be a simple content condition or complex content condition but just as well another secondary condition. Thus, arbitrarily long chains of secondary conditions can be produced that ultimately refer to a simple content condition or complex content condition. In effect, this paves the way

to the representation of very complex content models within the transition rules of a typing automaton.

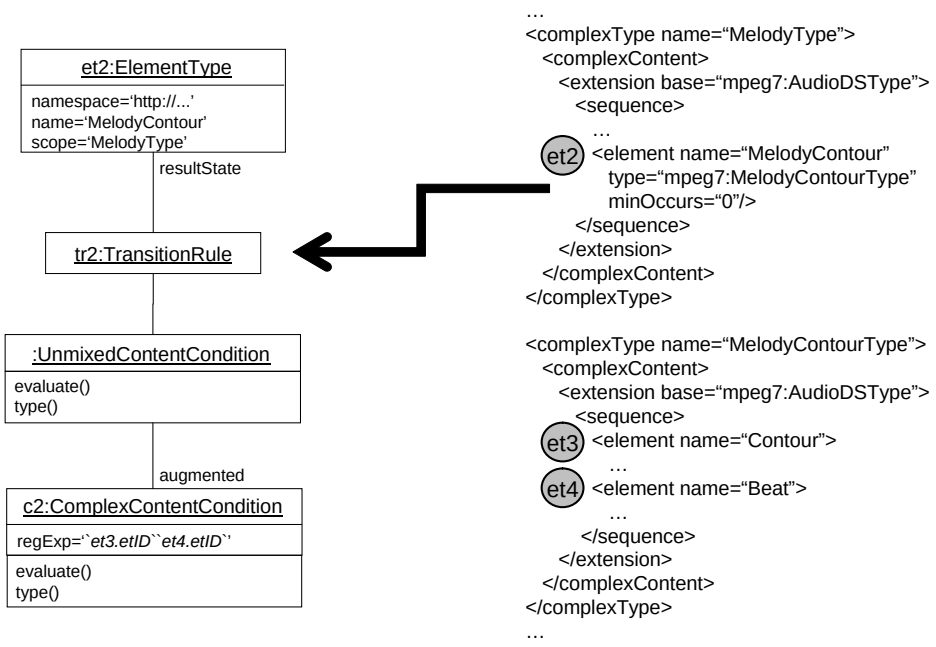


Figure 22: Example mapping of unmixed element content declaration (UML object diagram)

Figure 22 illustrates the use of unmixed content conditions for the representation of element type declarations contained in MPEG-7 DDL schema definitions whose content models do not permit mixed content. In that figure, we have mapped the declaration of the element type `MelodyContour` contained in the complex type `MelodyType` within the `Melody` media description scheme to transition rule `tr2` just as we did before in Figure 17. But as the `MelodyContour` declaration does not permit mixed content, we have augmented the transition rule's complex content condition `c2` with a further unmixed content condition to obtain a more faithful mapping.

Definition 24 outlines a formal specification of the behaviour of `evaluate()` and `type()` methods for the class `UnmixedContentCondition` so that instances of this class can be used like in Figure 22 to prevent the occurrence of mixed content. According to the postcondition provided for the specification of `evaluate()`, the method first checks whether the passed element has text nodes among its child nodes. If it has, the unmixed content condition already evaluates to `false`. Otherwise, the result of the `evaluate()` method is identical to the result

of the augmented condition's `evaluate()` method. As an unmixed content condition behaves like the augmented condition for the construction of typed representations, the pseudocode given for the `type()` method of `UnmixedContentCondition` simply delegates all its calls to the `type()` method of the augmented condition.

Definition 24 (Unmixed content condition)

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

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

5.5.3 Complex type declarations

MPEG-7 DDL offers complex type declarations as a powerful and flexible construct for organizing the structure of schema definitions which, as we have already been able to observe by means of our sample `Melody` media description scheme, faces extensive use within MPEG-7 media description schemes. A complex type essentially constitutes a named complex content model which can be referenced within an element type declaration in order to define the con-

tents valid for the element type. Given their relevance for MPEG-7, it is clearly of interest to examine how complex type declarations are represented within typing automata.

Inspection of the mapping scheme we have used to translate the `Melody` media description scheme of Figure 1 to the set of transition rules depicted by Figure 17 reveals that there is no one-to-one correspondence between complex type declarations and transition rules. Instead, a complex type declaration is implicitly covered by the complex content conditions of all those transition rules that represent element type declarations where the complex type is used to define the element type's content model. E.g, the complex type `MeterType` is not translated to a dedicated transition rule. But it is used to create the complex content condition `c1` of the transition rule `tr1` which represents the declaration of the element type `Meter` because `Meter`'s content model is defined by means of `MeterType`.

This kind of mapping imposes no problems as long as complex types are not interrelated. However, MPEG-7 DDL allows to derive complex types from each other. A complex type may extend the content model of the complex type it is derived from with additional elements and attributes or it may restrict the content model to a valid subset. Complex types can thus be organized into a kind of specialization hierarchy. MPEG-7 makes heavy use of this feature for organizing its media description schemes. In our `Melody` media description scheme, for instance, every complex type is derived from the complex types `AudioDSType` or `AudioDType`, which in turn are ultimately derived from the predefined complex types `DSType` and `DType` which form the roots of MPEG-7's specialization hierarchy [34].

Once such a specialization hierarchy is established, MPEG-7 DDL supports a special form of polymorphism: whenever a complex type is used to define the content model of an element type, the content models defined by the complex types directly or indirectly derived from the complex type constitute perfectly valid content models for the element type as well. An element that instantiates the element type in an XML document according to the content model of a derived complex type only has to announce the name and namespace of that derived type employing the predefined attribute `xsi:type`.

The problem with the mapping scheme used so far for translating MPEG-7 DDL schema definitions to typing automata is that it completely ignores complex type polymorphism. Because of this – and as an observant reader might have already noticed – we were forced to fall

back on some black magic during the translation of the `Melody` media description scheme to the transition rules of Figure 17 so that the example media description of Figure 2 is valid with regard to the typing automaton. Although the content model of the element type `AudioDescriptionScheme` is defined by the complex type `AudioDSType`, we clairvoyantly knew that the example description would employ complex type polymorphism and fill the content of its root element according to the complex type `MelodyType`. Thus, we have constructed transition rule `tr7` as if the the content model of `AudioDescriptionScheme` had been defined by `MelodyType` right from the beginning.

To obtain support for complex type polymorphism, we therefore suggest an extended mapping scheme. In that scheme, every element type declaration is translated to a corresponding transition rule just as before. But whenever the content model within an element type declaration is defined by means of a complex type from which other complex types are derived in the schema definition, an additional transition is introduced for each of the directly or indirectly derived complex types. Such an additional transition bears the element type of the original element type declaration as its result state; its condition basically constitutes a complex content condition that represents the effective content model which is defined by the derived complex type. Hence, we effectively create a transition rule for every of the element type's content models that could be potentially instantiated by an element of that type via complex type polymorphism within a document.

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

document by means of an `xsi:type` attribute value.

Thus, we suggest the introduction of complex type polymorphism conditions as a further kind of secondary condition. Complex type polymorphism conditions are represented by the class `ComplexTypePolymorphismCondition` in the class diagram of Figure 21. A complex type polymorphism condition maintains the name and namespace of a given complex type as indicated by the attributes `name` and `namespace`. During the validation phase, a complex type polymorphism verifies whether the element for which the condition is evaluated features an `xsi:type` attribute value addressing the name and namespace of the complex type maintained by the condition. Only if this relatively simple check has been successfully passed, the condition augmented the complex content condition augments is also evaluated. Since complex type polymorphism conditions only serve to decide for one of the alternative transition rules representing a single element type declaration more quickly but do not otherwise influence the creation of corresponding typed representations once an appropriate transition rule has been chosen, they exactly behave like the augmented conditions during the typing phase.

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

Figure 23 provides an example that illustrates the extended mapping scheme and the application of complex type polymorphism conditions. The top of the figure shows an excerpt of the `Melody` media description scheme consisting of the complex types `MelodyContourType` and `MelodyType` as well as the element type `AudioDescriptionScheme` as we know them from Figure 1 already. In addition, the declaration of the complex type `AudioDSType` is provided that is used to define the content model of `AudioDescriptionScheme`. `AudioDSType` specifies a content model consisting of an arbitrarily repeatable sequence of elements of type `Header`.

The bottom of the figure depicts the mapping of the `AudioDescriptionScheme` element type declaration to the transition rules of a typing automaton according to the suggested extended mapping scheme. First of all, the element type declaration is translated to the transition rule `tr8` as usual. `tr8` employs the complex condition `c8` to restrict the

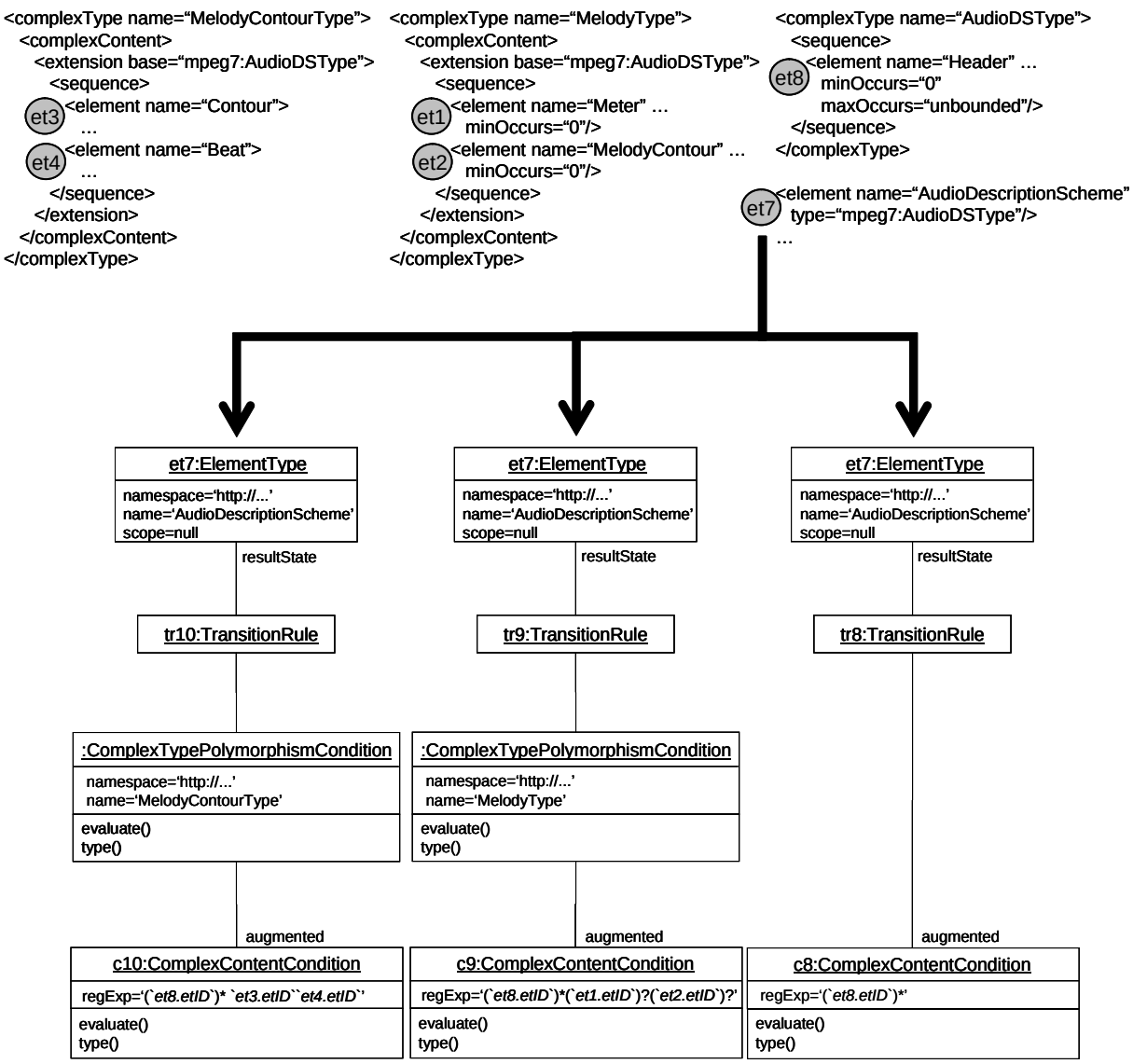


Figure 23: Example mapping of complex type derivation hierarchy (UML object diagram)

allowable contents of `AudioDescriptionScheme` elements to arbitrarily long sequences of `Header` elements as demanded by `AudioDSType`. In a second step, further transition rules with `AudioDescriptionScheme` as their result state are created for each complex type derived from `AudioDSType`. In this example, these are the complex types `MelodyType` and `MelodyContourType` which lead to the transition rules `tr9` and `tr10`. The conditions of `tr9` and `tr10` are made up of complex type polymorphism conditions which reference the names and namespaces of the respective complex types. Both conditions further augment complex content conditions that model the effective content models defined by the complex types. I.e.,

condition c9 restricts the permitted content of AudioDescriptionScheme elements to arbitrarily long sequences of Header elements followed by optional Meter and MelodyContour elements as defined by the complex type MelodyType; condition c10 restricts the permitted content to arbitrarily long sequences of Header elements followed by Contour and Beat elements as defined by the complex type MelodyContourType.

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

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

Definition 25 (Complex type polymorphism condition)

```
context ComplexTypePolymorphismCondition::evaluate(Element e,
            TransitionRule tr) :  Boolean
post:   result = e.attributeValue -> exists(av |
            av.attName = "type" and
            av.attNamespace = "http://www.w3.org/2001/XMLSchema-instance" and
            av.content.name = self.name and
```

```
            av.content.namespace = self.namespace) and
        self.augmented.evaluate(e, tr)

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

5.5.4 Attribute declarations

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

Regarding this support for strong typing and given the fact that MPEG-7 media description schemes considerably make use of that support to define attributes that carry non-textual data, it is no surprise that TDOM's notion of typed representation not only encompasses elements

but also attribute values. In order to be considered an adequate intermediary representation of an MPEG-7 DDL schema definition, typing automata should consequently provide support for the representation of attribute declarations that facilitate the validation of attribute values occurring in a document and the production of corresponding typed representations of these values. So far, however, our specification of typing automata has intentionally ignored attribute values for simplicity.

In the following, we therefore want to outline an extension of typing automata for the representation of attribute declarations. Just like the other extensions suggested so far, we again perform this extension by introducing a dedicated kind of secondary condition, namely attribute conditions which are represented by the class `AttributeCondition` in the diagram of Figure 21. An attribute condition basically consists of a collection of attribute declarations. If an attribute condition is used to augment a transition rule's condition, not only the augmented condition is evaluated during the validation phase whenever the transition rule is applied to an element; it is also verified whether the attribute values of the element conform to the attribute declarations collected by the attribute condition. During the typing phase, the attribute condition behaves like the augmented condition but additionally transforms the attribute values of the element to corresponding typed representations in a way that suits the attribute declarations to which the attibute values comply.

The class diagram of Figure 24 provides further details on the structure we suggest for attribute conditions and attribute declarations. According to the diagram, an attribute condition collects at least one attribute declaration all of which are subsumed by the abstract base class `AttributeDeclaration`. Attribute declarations are categorized into single attribute declarations and any attribute declarations as modeled by the subclasses `SingleAttributeDeclaration` and `AnyAttributeDeclaration`. The abstract notion of single attribute declarations subsumes basic attribute declarations that essentially permit a single value of the attribute (addressed by the association to the class `Attribute`) to appear within an element, if the content of the attribute value originates from the domain of the simple type (addressed by the association to the class `SimpleType`). In the diagram, the single attribute declarations considered are fixed, optional, required, and prohibited attribute declarations represented by corresponding subclasses. These represent the usual basic kinds of attribute decla-

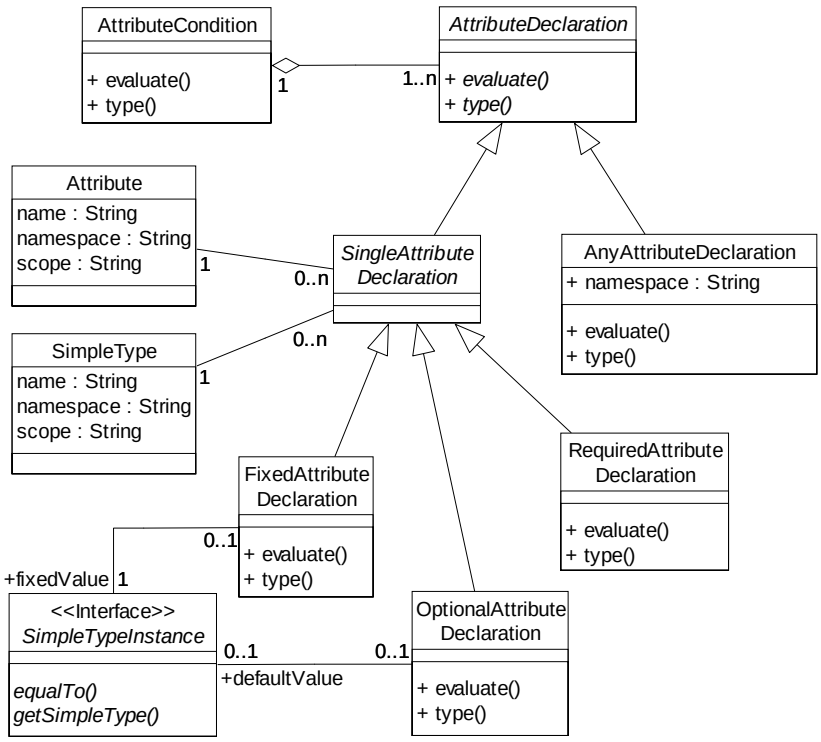


Figure 24: Attribute condition structure (UML class diagram)

rations that are supported by most XML schema definition languages: required attribute declarations enforce the instantiation of a given attribute within an element. Optional attribute declarations allow the instantiation of a given attribute within an element but do not enforce it. A default value in form of a simple type instance can be specified for the case that an optional attribute is not instantiated. Fixed attribute declarations behave like optional attribute declarations but rigidly restrict the allowable contents of attribute values to the simple type instance provided as the fixed value. Finally, prohibited attribute declarations forbid the instantiation of an attribute within an element. Support for further kinds of attribute declarations could be integrated into this structure by additional subclasses of `SingleAttributeDeclaration` if necessary.

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

automaton.

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

Constraint 11 (Restrictions on attribute conditions)

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

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

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

Figure 25 provides an example how attribute conditions can be used to represent attribute declarations occurring in an MPEG-7 media description scheme. The right part of the figure shows the declaration of the complex type `AudioDSType` which has been enhanced by an attribute declaration permitting the optional use of an attribute `id` of type `ID` and by a declaration on the basis of the `<anyAttribute>` construct further allowing the use of arbitrary attributes as long as they originate from the namespace `http://www.mpeg7.org/`.

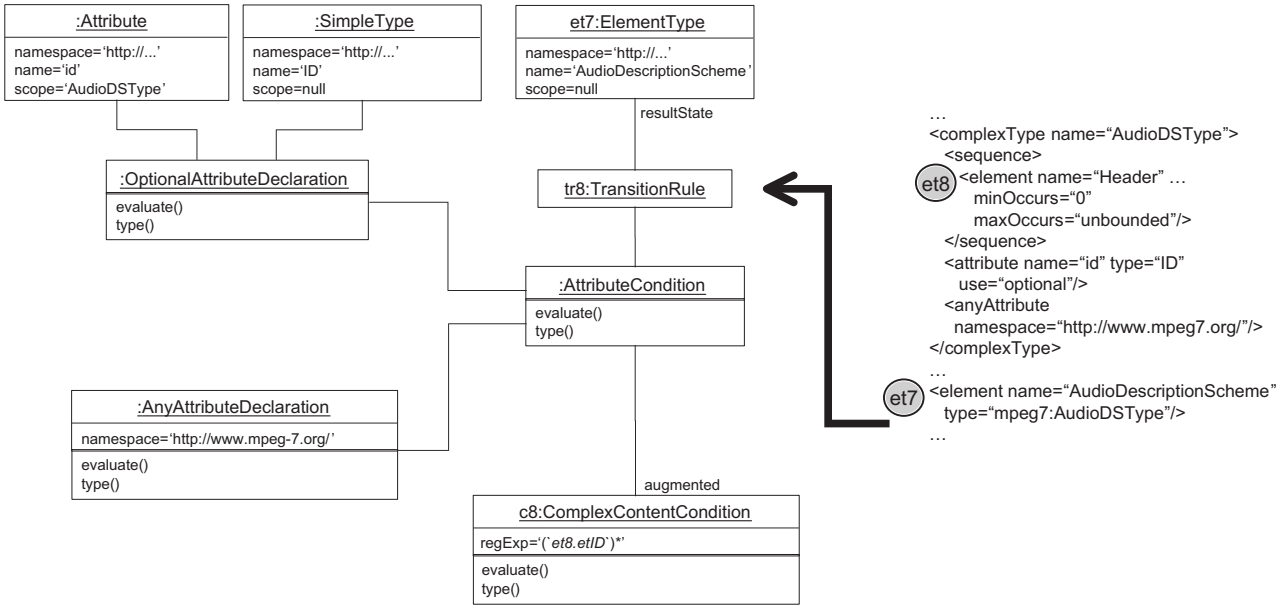


Figure 25: Example mapping of attribute declarations (UML object diagram)

The left part of the figure shows the representation of the declaration of the element type `AudioDescriptionScheme`, whose content model is defined via `AudioDSType`, by means of a transition rule. For the figure, complex type polymorphism is neglected for the sake of clarity. As one can observe, the attribute declaration is straightforwardly mapped to an attribute condition within the transition rule. The attribute condition augments a complex content condition that restricts allowable element contents to arbitrarily long sequences of `Header` elements in accordance to `AudioDSType`. The attribute condition models the declaration of the `id` attribute with an optional attribute declaration which references an `Attribute` object representing `id` and a `SimpleType` object modeling the simple type `ID`. The `<anyAttribute>` construct is mapped to an any attribute declaration whose `namespace` property is set to `http://www.mpeg7.org/`.

Definition 26 details the behaviour of attribute conditions during the validation and typing phases of a typing automaton by outlining a specification of the `evaluate()` and `type()` methods of the class `AttributeCondition`. Since an attribute condition essentially constitutes a container for various kinds of attribute declarations, the specification of both methods relies on a fixed set of functionality that has to be offered by an attribute declaration. As already indicated by the abstract methods of the abstract base class `AttributeDeclaration` in the

class diagram of Figure 24, any concrete subclass of `AttributeDeclaration` that represents a particular kind of attribute declaration must provide appropriate implementations for the methods `evaluate()` and `type()` similar to a condition's methods of the same name: `evaluate()` expects an attribute value as its parameter and returns `true` if the attribute value constitutes a valid instantiation of the current attribute declaration; `type()` takes an attribute value in untyped representation as its parameter and transforms the attribute value to a corresponding typed representation in a manner that is appropriate for the current attribute declaration.

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

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

Definition 26 (Attribute condition)

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

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

```
foreach av in e.attributeValue -> select(untyped) do
    iad := self.attributeDeclaration -> any(ad | ad.evaluate(av))
    iad.type(av)
endforeach
```

We conclude our treatment of attribute conditions by showing how implementations of the `evaluate()` and `type()` methods of attribute declarations could look like. We restrict ourselves to optional attribute declarations and any attribute declarations that we have used for our example of Figure 25. The implementations of both methods for optional attribute declarations are covered by Definition 27. `evaluate()` delivers `true` for an attribute value if (a) the attribute name and namespace of the attribute value match the name and namespace of the attribute addressed by the current optional attribute declaration and if (b) an instance of the simple type addressed by the optional attribute declaration can be constructed from the textual representation of the attribute value's content. `type()` uses the attribute and simple type associated with the optional attribute declaration to straightforwardly the attribute value that is passed passed to the method to typed representation.

Definition 27 (Optional attribute condition)

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

context OptionalAttributeCondition::type(AttributeValue av)
pre:    not(av.typed)
```

pre: `self.evaluate(av)`

post: `av.CUR(av@pre)`

pseudocode:

```
-- Bring the attribute value to an appropriate typed
-- representation.
av.typed := true
av.name := null
av.namespace := null
av.attribute := av.attribute -> including(self.attribute)
-- Use the simple type instance factory associated
-- with the simple type of the optional attribute condition
-- to produce an appropriate simple type instance for use as
-- typed attribute value content
stif := self.simpleType.simpleTypeInstanceFactory
sti := stif.fromString(av.simpleContent)
-- Set simple type instance as simple content of element
av.simpleContent := null
av.typedSimpleContent := av.typedSimpleContent
    -> including(sti)
```

Definition 28 treats the `evaluate()` and `type()` methods of any attribute declarations. `evaluate()` returns `true` if the attribute namespace of the attribute value matches the namespace referred to by the current any attribute declaration. Because an any attribute declaration does not carry sufficient information for the construction of a corresponding typed representation of the attribute value that is passed to `type()`, the method attempts to delegate its call to a single attribute declaration that is also validly instantiated by the attribute value. Should no such single attribute declaration exists, the attribute value remains in untyped representation.

Definition 28 (Any attribute condition)

context `AnyAttributeCondition::evaluate(AttributeValue av) : Boolean`

post: `result = self.namespace <> null` **implies**

`av.attNamespace = self.namespace`

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

6 Conclusion

Starting out with essential requirements for the management of MPEG-7 media descriptions, we have analyzed current XML database solutions for their suitability for use in the context of MPEG-7. Facing the deficiencies of these solutions with respect to these requirements, we have realized the need for more adequate database solutions for MPEG-7 media descriptions. As a foundation of such a solution, we have introduced the Typed Document Object Model, a data model for XML documents specifically designed with the requirements for the management of MPEG-7 media descriptions in mind. We have highlighted TDOM's key features and given a

thorough definition of the model. We have introduced and formally defined typing automata as an executable intermediary representation for media description schemes that is capable of validating MPEG-7 media descriptions and inferring typed representations their contents. We have proposed optimizations for the core functionality of typing automata that promise to substantially reduce the effort necessary for document validation and typing, especially after document updates. We have further shown that the mechanism of typing automata is flexible enough to be extended up to the expressiveness of MPEG-7 DDL.

We have fully implemented TDOM with Java on the basis of the object-oriented DBMS ObjectStore. Our implementation comes with a schema catalog that manages media description schemes on the basis of typing automata and uses these automata for the validation of MPEG-7 media descriptions and the automatic construction of typed representations of the contents of the descriptions. Furthermore, our implementation includes an indexing component supporting a variety of secondary access methods for indexing the basic contents of a media description including Hashtables, B-Trees, and R-Trees. We are currently providing a processor for XPath expressions [9] that exploits schema information and indexes available for an optimized query evaluation. The XPath processor forms the heart of optimizing processors for XQuery [4] and XSLT [7] that we plan to implement in future.

References

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

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

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

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

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

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

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

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

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

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

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

[14] A. Davidson, M. Fuchs, M. Hedin, et al. Schema for Object-Oriented XML. W3C Note Version 2.0, World Wide Web Consortium (W3C), July 1999.

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

[16] 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.

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

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

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

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

[21] 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.

[22] 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.

[23] S. Higgins, O. Alonso, S. Banerjee, et al. Oracle 9i Application Developer's Guide – XML. Product Documentation Release 1 (9.0.1), Oracle Corp., June 2001.

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

[25] 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.

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

[27] IEEE P1484.12 Learning Object Metadata Working Group. Draft Standard for Learning Object Metadata. IEEE Draft Standard P1484.12/D6.1, Institute of Electrical and Electronics Engineers, Inc. (IEEE), April 2001.

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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

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

[37] 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.

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

[39] 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.

[40] 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.

[41] A. Le Hors, P. Le Hégaret, L. Wood, et al. Document Object Model (DOM) Level 3 Core Specification. W3C Working Draft Version 1.0, World Wide Web Consortium (W3C), April 2002.

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

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

[44] 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.

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

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

[47] 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.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

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

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

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

[57] 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.

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

[59] X-Hive Corp. X-Hive/DB 2.0 – Manual. System Documentation Release 2.0.2, X-Hive Corp., May 2002.

[60] xCBL.org. XML Common Business Library (xCBL). Structure Reference Version 3.5, Commerce One, Inc., November 2001.

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