Time to Leave the Trees: From Syntactic to Conceptual Querying of XML*

Bertram Ludäscher Ilkay Altintas Amarnath Gupta

San Diego Supercomputer Center
University of California, San Diego
{ludaesch,altintas,gupta}@sdsc.edu

Abstract. Current XML query languages operate on XML instances only but ignore valuable conceptual level information that is "buried" inside complex XML Schema documents. For example, XPath queries are evaluated against XML documents based on *element names* (tags) and their *syntactic nesting structure*, ignoring the *element types* and other conceptual level information that is declared in separate XML schemas. We propose an extension to XML query languages for *conceptual querying* of XML, based on an underlying abstract model of XML Schema (MXS). We show that this approach offers the user new and often more adequate high-level query capabilities beyond the traditional purely syntactic approach of querying XML trees.

1 Introduction

XML Schema is replacing XML Document Type Definitions (DTDs) for modeling the structure and contents of XML databases and is emerging as the standard XML schema language. With its roots in hypertext documents and SGML, the modeling capabilities of XML DTDs were very limited from a database perspective. Essentially, a DTD can be viewed as an extended context free grammar [Via01] or a (local) regular tree grammar [MLM01b] that simply specifies a set of *element names* (*i.e.*, tags for typing the tree nodes) together with their possible *nesting structure* (via regular expressions). In particular, there are no built-in or user-defined data types, class or type hierarchies, etc.

In contrast, XML Schema [XML01] includes a large number of built-in datatypes, user-defined complex datatypes, subtyping through extension and restriction mechanisms, ordered and unordered collection types, and various integrity constraints, *e.g.*, for specifying multiplicity constraints. At the same time, XML Schema retains the versatility of XML DTDs for representing *semistructured data* via flexible content models. Indeed XML Schema can be viewed as a (poor man's) object-oriented data model[1] that allows a database designer to employ

* Research supported by DOE SciDAC/SDM DE-FC02-01ER25486, NIH BIRN-CC 3 P41 RR08605-08S1, NSF NPACI ACI-9619020 (NARA suppl.), NSF NSDL/UCAR Subaward S02-36645, NSF I2T EIA9983510

[1] with a "rich man's" flexibility for defining the slot structure of objects using DTDs

object-oriented modeling techniques to express additional semantics beyond the purely syntactic information contained in XML database instances.

The information contained in a (relational, object-oriented, or XML) database schema can be used in many ways, *e.g.*, as a conceptual model during database design, to help formulate meaningful queries over the logical database schema, to validate instances, to determine and optimize the storage model and layout, or to facilitate semantic query optimization. However, current XML query languages such as XPath [XPa99] and XQuery [XQu01] operate only on the *syntactic level* of the given XML instance, *i.e.*, the "DTD part" of XML Schema that deals with tag names, element order and nesting only, and – without extensions like the ones proposed in this paper – these languages cannot make use of valuable *conceptual level* information given through an XML schema.

Example 1 (Syntactic vs. Conceptual Querying)
Consider a conceptual schema comprising classes `book` and `textbook`, where the latter is a subclass of the former. For example, `textbook` objects may have an additional attribute `@courses` to represent the courses for which a textbook is recommended. Using XML Schema we can declare the *type* `textBookT` to be a subtype of *type* `bookT` and declare that `<book>` and `<textBook>` *elements* are of the respective types.[2] In order to even adequately formulate queries such as

> *How many books are there in the database? What is their average price? List all books that cost between \$50 and \$100, ...*

XML Schema information has to be taken into account: *e.g.*, simply retrieving and counting `<book>` elements and computing their average price will result in an incorrect answer! Here, abiding to the art of good modeling and schema design, the database engineer may have deliberately specified in the XML Schema that `<textBook>` objects are a special kind of `<book>` objects with certain additional features. However, this information is *absent* from the database instance, which only contains element names, but not type names. Therefore, purely syntactic query languages working on XML instances only, cannot "know" that `<textbook>` elements are also `<book>` elements. Therefore, the programmer has to consult (and understand!) a separate XML Schema document like the one in Appendix A *and* carry the burden of coming up with a standard syntactic XML query Q_s that returns the desired result. Instead, the programmer should be able to define an equivalent conceptual query Q_c over an abstract conceptual model of the XML Schema information.[3] □

We argue that it is desirable for the user to be able to query both the syntactic element structure of XML instances as well as the accompanying *latent class structure* (induced by type declarations) and other conceptual level information hidden in XML Schemas. The programmer should be relieved from the burden of rewriting "type-enabled" conceptual queries into syntactic ones, say

[2] For readability, we use the convention that user-defined *type names* end with "`T`", say `bookT`, while *element names* (tags) are enclosed in pointy brackets: *e.g.*, `<book>`.

[3] See Section 3.3 for concrete examples of Q_s and Q_c.

in XPath, that achieve the desired effect. To this end, we propose to make the conceptual level information buried in XML schemas accessible to the user in an intuitive and seamless way, similar to the way she can now use conventional path expressions against XML documents without schema information. Note that while XML Schema documents are themselves XML documents and thus can *in principle* be queried directly using XPath and XQuery, doing so does *not* provide the user with a reasonable mechanism for conceptual level queries. To see why, consider the abstract XML Schema information depicted in Figure 1; its concrete XML Schema representation is shown in Appendix A: It is very hard to see the forest for the trees! For example, try to specify an XML query that will find all possible element names and types that can be associated directly or indirectly with `<publication>`, *i.e.*, via element/type associations, subtype relationships (extension or restriction), or substitution groups. In contrast, this query is easily formulated and evaluated over the conceptual level information in Figure 1.

The organization and main contributions of the paper are as follows. The basic idea of our approach is to first distill adequate conceptual level information of an XML Schema and represent it in an abstract model. To this end, we develop the abstract *Model of XML Schema* (MXS) in Section 2. Based on this model, Section 3 develops the framework for conceptual querying of XML: a standard XML instance model (Section 3.1) is augmented with schema information (Section 3.2). In Section 3.3, we use illustrative examples to show that conceptual XML queries are often preferable to syntactic ones. Section 3.4 defines an extension to XPath called *XPathT* (XPath with types) and shows how an executable logic specification for XPathT can be obtained. Finally, Section 4 summarizes and concludes.

Related Work

In terms of XML *query formulation and evaluation*, XML Schema information has been lying fallow until now and – to the best of our knowledge – this is the first approach that employs XML Schema information for conceptual level querying of XML.

The need for a formal and intuitive abstraction of the complex XML Schema standard [XML01] has been identified and addressed in [BFRW01a,BFRW01b]. Our prototypical implementation of XPathT employs such a formal abstraction called MXS (Section 2) as the basis of the conceptual querying framework. Our basic approach can also be used with other XML schema languages; see [Cov02] for a list of the many competing approaches, [MLM01b] for a comparison based on formal language theory, and [MLM01a] for a study of data modeling capabilities of XML schema languages including a comparison with traditional (E)ER models. XML DTDs are quite impoverished in terms of data modeling capabilities, *e.g.*, there is no subtyping mechanism [LPVV99,PV00]. Another area where XML schemas are relevant for XML querying is in type checking and type inference of queries: [Via01] provides an overview on the problems in type checking XML queries. A typical type checking problem is to compute, as accurately as possible, the output type of a query, given the query expression and the type of the input document. XML Schema information can also be used for type checking XQuery expressions when translating operator trees to XQuery Core expressions [FW01, p.91]. Clearly, these approaches are very different from ours, since they do not aim at incorporating XML schema information for query formulation and the actual query processing at runtime. While we propose to use abstract XML schemas as first-class queryable objects, these works use schemas for compile-time analysis of queries.

2 Abstract Model for XML Schema (MXS)

The extension from syntactic to conceptual queries is based on an underlying abstract *Model for XML Schema* (MXS). The XML Schema standard [XML01] has been criticized for its complexity, and attempts at a simpler and more formal description are underway [BFRW01b,BFRW01a]. As it turns out, much of the complexity is due to intricacies of how to *declare* schema information and *not* caused by the actual underlying schema metamodel (*i.e.*, the object-oriented modeling constructs which the schema designer can use) which is rather simple and straightforward. In the sequel, we present MXS, which is our formal model and approximation of the XML Schema standard. We do not aim at a complete formalization of all details of XML Schema, but rather at capturing its essential modeling features as required for conceptual querying of XML. Note that our approach of "type-aware" querying is not limited to XML Schema but could also be based on other XML schema languages such as RELAX NG [CM01].

Definition 1 (Names) An MXS schema comprises pairwise disjoint sets $\mathbb{T}$, $\mathbb{E}$, $\mathbb{A}$ of *type names*, *element names*, and *attribute names*, respectively. □

We often simply say *type*, *element*, and *attribute* when referring to $\mathbb{T}$, $\mathbb{E}$, and $\mathbb{A}$. For example, for the MXS schemas in Figures 1–3 we have types $\mathbb{T} = \{\texttt{bookT}, \texttt{USauthorT}, \ldots\}$, elements $\mathbb{E} = \{\texttt{<book>}, \texttt{<USauthor>}, \ldots\}$, and attributes $\mathbb{A} = \{\texttt{@coverStyle}, \texttt{@countryOfBirth}\}$.

Definition 2 (Kinds of Types)
Each type $T \in \mathbb{T}$ is either *simple* or *complex*, and either *abstract* or *concrete* leading to two partitions: $\mathbb{T} = \mathbb{T}_s \,\dot\cup\, \mathbb{T}_c$ and $\mathbb{T} = \mathbb{T}_a \,\dot\cup\, \mathbb{T}_{na}$.[4] □

For example, `nnDecimalT` (Figure 1) is a simple and concrete type, hence in $\mathbb{T}_s \cap \mathbb{T}_{na}$; on the other hand, `publicationT` $\in \mathbb{T}_c \cap \mathbb{T}_a$, so this is a complex and abstract type.

Definition 3 (Type Hierarchy) *Restriction* is a binary subtyping relation

- $\mathcal{R} \;\subseteq\; (\mathbb{T}_s \times \mathbb{T}_s) \cup (\mathbb{T}_c \times \mathbb{T}_c)$, $\quad (\mathcal{R})$

[4] $\mathbb{T}_{na}$ stands for non-abstract, *i.e.*, concrete types.

while *extension* is a binary subtyping relation

- $\mathcal{E} \subseteq (\mathbb{T}_s \cup \mathbb{T}_c) \times \mathbb{T}_c$. $(\mathcal{E})$

Given $\mathcal{R}$ and $\mathcal{E}$, the *subtyping relation* is

- $\mathcal{S} = \mathcal{R} \cup \mathcal{E}$. $(\mathcal{S})$

We require that $\mathcal{S}$ is *acyclic*, and that each subtype is derived from a single parent type only[5]. The induced forest is called the *type hierarchy* (or *class hierarchy*) over the given schema. The transitive closure of $\mathcal{S}$ is denoted by $\mathcal{S}^*$. □

Let T be a type and T' a subtype of T. In Section 3, we use predicates $\mathsf{restrict}(T, T')$, $\mathsf{extend}(T, T')$, and $\mathsf{subtype}(T, T')$ to denote the subtyping relations $\mathcal{R}$, $\mathcal{E}$, and $\mathcal{S}$, respectively. Restriction of a simple or complex type yields the same kind of type ($\mathcal{R}$), whereas extension of a simple or complex type always yields a complex type ($\mathcal{E}$). The reason is that in XML Schema, *extending* a type means to *add* "slots" (elements or attributes) to the XML objects, which always results in a complex type. In contrast, *restricting* a type keeps the given slot structure, but adds constraints on the existing slots. Conceptually, both restriction and extension are subtyping mechanisms ($\mathcal{S}$). XML Schema does not allow the user to use both features in a single step, however:

Example 2 (Indirect Subtypes) Assume we want to define a *direct* subtype of `bookT` called `expTextBookT` for expensive textbooks. We could derive the subtype from its supertype by (i) requiring an additional field `recommended_for` that lists the target audience of a textbook, and (ii) constraining the price to be more than \$100. Since in XML Schema the subtyping mechanisms restriction and extension cannot be used together in a single step, one must work around this limitation and introduce intermediate types, *e.g.*, extend(`bookT`, `textBookT`) and restrict(`textBookT`, `expTextBookT`) as shown in Figure 1. The corresponding MXS declarations are shown in Figure 2. □

XML Schema also lacks *multiple inheritance* known from object-oriented modeling:

Example 3 (Multiple Inheritance) We cannot define the type "19^{th} century textbook" as the common subtype of both `textbookT` and `c19bookT` (19^{th} century book). Instead we can either restrict the former by constraining the `<pubYear>` element to be of type `c19gYearT` (a restriction of XML Schema's year type `xsd:gYear` to the 19^{th} century), or by extending `c19bookT` with a `<recommended_for>` element. This results in the two *structurally equivalent* (but not name equivalent) types `textc19BookT` and `c19textBookT` shown in Figure 1 and Figure 2. □

From a user's modeling perspective, an XML Schema type hierarchy can be viewed as a class hierarchy that can be employed for simplifying syntactic

[5] this requirement can be lifted, but mirrors the fact that XML Schema does not have multiple inheritance, *i.e.*, two different types cannot have a common subtype

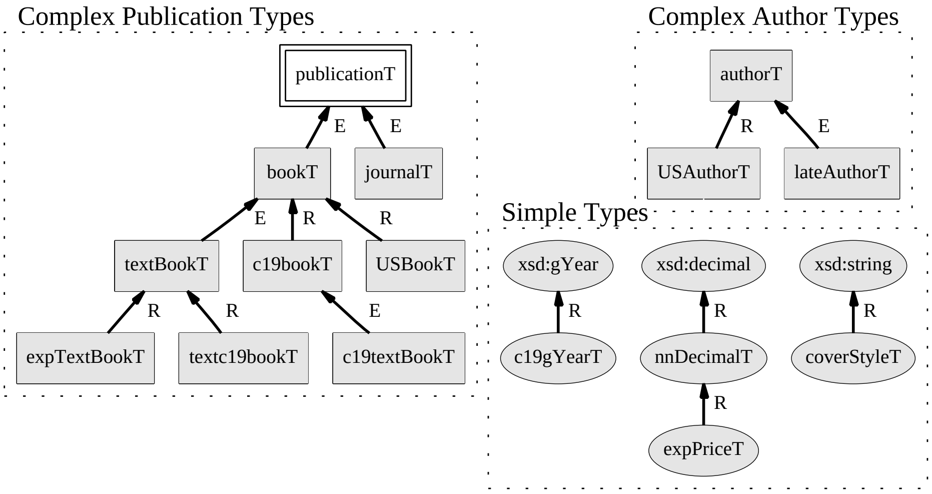

Fig. 1. Type hierarchies with *concrete types* (filled) and an *abstract type* (double border); *simple* (*complex*) *types* are depicted as ellipses (boxes); E (R) denotes *extension* (*restriction*).

queries and for formulating more adequate conceptual level queries. In the MXS framework, this includes navigations in the (transitive) class hierarchy $\mathcal{S}^*$, which abstracts from the limitations of how types have to be declared syntactically.

Definition 4 (Binding Elements and Attributes to Types) A *binding* of elements and attributes to types is a binary relation

- $\mathcal{B} \subseteq (\mathbb{E} \times (\mathbb{T}_s \cup \mathbb{T}_c)) \cup (\mathbb{A} \times \mathbb{T}_s)$, $\qquad (\mathcal{B})$

that maps element names to (simple or complex) types and attribute names to simple types. $\mathcal{B}$ is required to be functional, *i.e.*, each element/attribute maps to only one type. □

For example, in Figure 2, `<book>` is bound to the type `bookT` and `<tbook>` to type `textBookT` (see `/*ELEMENT DECLARATIONS*/`). In XML Schema these are called *global* bindings. In contrast, there are several *local* bindings of `<author>` to an author type, *e.g.*, within the complex type definition of `USBookT`, the `<author>` element is bound to the type `USauthorT`. Thus, when syntactically querying an XML instance for `<author>` elements (say using "`//author`"), we may retrieve different types of authors (`authorT`, `USauthorT`, `lateAuthorT`) without being able to directly query the types (classes) to which the different local `<author>` elements belong. We may still be able to "mine" this information from the instance by first analyzing the XML Schema (in Appendix A, or better its abstract MXS version in Figure 2), and then coming up with syntactic queries that allow us to identify *US authors* (check that `@countryOfBirth="USA"`), *late authors* (check for

```
/*========================
  COMPLEX TYPES DEFINITIONS
  ========================*/
publicationsT =
  <pubsA> :: pubsAT,
  <pubsB> :: pubsBT

pubsAT =
  <publication> :: publicationT[0..]

pubsBT =
  <books> :: booksT,
  <journal> :: journalT[0..]

publicationT[ABSTRACT] = /* abstract type */
  <title>   :: xsd:string,
  <pubYear> :: xsd:gYear,

booksT = /* abstract (sub)element <aBook> */
  <aBook>[ABSTRACT] :: bookT[0..]

journalT =
  EXTEND publicationT BY
    <editor> :: xsd:string[1..],

bookT =
  EXTEND publicationT BY
    <author> :: authorT[1..],
    <price>   :: xsd:nnDecimalT
  ; @coverStyle :: coverStyleT

textBookT =
  EXTEND bookT BY
    <recommended_for> :: xsd:string

expTextBookT =
  RESTRICT textBookT AS
    <title>   :: xsd:string,
    <pubYear> :: xsd:gYear,
    <author>  :: authorT[1..],
    <price>   :: expPriceT,
    <recommended_for> :: xsd:string
  ; @coverStyle :: coverStyleT

c19bookT =
  RESTRICT bookT AS
    <title>   :: xsd:string,
    <pubYear> :: c19gYearT,
    <author>  :: authorT[1..],
    <price>   :: nnDecimalT
  ; @coverStyle :: coverStyleT(
                   default="hardcover")

textc19BookT =
  RESTRICT textBookT AS
    <title>   :: xsd:string,
    <pubYear> :: c19gYearT,
    <author>  :: authorT[1..],
    <price>   :: nnDecimalT,
    <recommended_for> :: xsd:string
  ; @coverStyle :: coverStyleT
```

```
c19textBookT =
  EXTEND c19bookT BY
    <recommended_for> :: xsd:string

USBookT =
  RESTRICT bookt AS
    <title>   :: xsd:string,
    <pubYear> :: xsd:gYear,
    <author>  :: USAuthorT[1..],
    <price>   :: nnDecimalT,
  ; @coverStyle :: coverStyleT

authorT =
  <name> :: xsd:string
  ; @countryOfBirth :: xsd:string

USAuthorT =
  RESTRICT authorT AS
    <name> :: xsd:string
  ; @countryOfBirth :: xsd:string(
                       fixed = "USA")

lateAuthorT =
  EXTEND authorT BY
    <deathYear> :: xsd:gYear

/*======================
  SIMPLE TYPE DEFINITIONS
  ======================*/
nnDecimalT =  /* non-negative decimal */
  RESTRICT xsd:decimal
  AS {MININCLUSIVE = 0.0})

expPriceT =
  RESTRICT nnDecimalT AS
    {MINEXCLUSIVE = 100.0}

coverStyleT =
  RESTRICT xsd:string AS
    {ENUMERATION = {"hardcover",
                    "paperback"}}
c19gYearT =
  RESTRICT xsd:gYear AS
    {MININCLUSIVE = 1800,
     MAXEXCLUSIVE = 1900}

/*===================
  ELEMENT DECLARATIONS
  ===================*/
<publications> :: publicationsT
<book>    :: bookT
<expbook> :: expTextBookT
<tbook>   :: textBookT
<cbook>   :: c19bookT
<ctbook>  :: c19textBookT
<tcbook>  :: textc19BookT
<USbook>  :: USBookT

/*==================
  SUBSTITUTION-GROUPS
  ==================*/
<aBook> <==>
  { <book>, <expbook>, <tbook>, <cbook>,
    <ctbook>, <tcbook>, <USbook> }
```

Fig. 2. XML Schema example (in MXS notation)

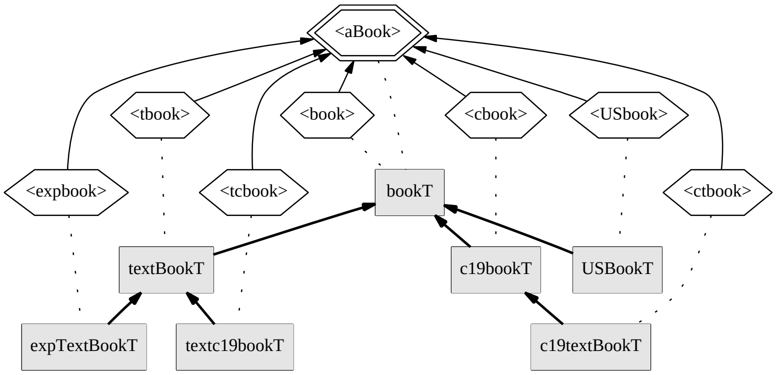

Fig. 3. Elements of a *substitution group* (hexagons) and associated complex types (boxes)

`<deathYear>`), and all *other authors*. With schema-aware conceptual querying of XML, the user is relieved from this burden.

Abstract Types. Figure 1 shows the abstract type `publicationT` and its concrete subtypes. As usual in object-oriented modeling, abstract types may not have direct instances, but their concrete subtypes may. For example, if an element, say `<publication>`, is bound to `publicationT` (Figure 2), then instances of that element will only be indirectly of type `publicationT`. The direct type has to be one of the concrete subtypes of `publicationT`, *e.g.*, `bookT` or `journalT`. In the XML instance, this is indicated by introducing a special attribute `@xsi:type` whose value is any concrete subtype of the abstract type (cf. the `<publication>` subelements of `<pubsA>` in the XML instance in Appendix A). The use of the `@xsi:type` attribute to discern which of the concrete subtype of an abstract type is being used, is an exception to the rule that XML Schema information is not part of XML instances.

Abstract Elements and Substitution Groups. An element name such as `<publication>` which is bound to an abstract type like `publicationT` can occur in an XML instance, but has to be subtyped to a concrete type, *e.g.*, `bookT`. XML Schema also introduces the notion of *abstract elements* such as `<aBook>` in Figure 2. An abstract element may *not* occur in the instance. Instead, a *substitution group* with concrete elements is defined, and any of the members of this group can be used in place of the abstract element. For example, Figure 3 depicts the substitution group of the abstract `<aBook>` element declared in Figure 2. Since the concrete elements of a substitution group can replace their corresponding abstract element in an instance, the members of the substitution group should have a subtype of the abstract element's type as shown in Figure 3. As an example, consider the children of `<pubsB>` in Appendix B: in place of the abstract element `<aBook>`, the instance contains elements of the substitution group. In contrast to the previous situation with abstract *types*, now the *element name* indicates how an abstract element is instantiated. Nevertheless, the instance loses conceptual level information, since the types of elements (grey boxes in Figure 3) are not known in the instance. Again the user has to consult the schema to discover the types of elements and how elements (or types) can be substituted for one another.

Content Models. The last important modeling feature of XML Schema is the definition of *complex content* or – in object-oriented parlance – the "slot structure" of a class. Essentially, content definitions can be seen as the (well-known) "DTD part" of XML Schema, so we omit the details here. The main difference to DTD content models is that XML content models also define the type of their content.

Most content models in Figure 2 include *sequence*, denoted simply by ",". For example, the content of `publicationsT` is the sequence "`<pubsA> , <pubsB>`". Another frequent construct is *collection*. For example, "`<author> :: authorT[1..]`" specifies a sequence of 1 or more `<author>` elements, each of type `authorT`. This corresponds to a typed version of the "`author+`" declaration in DTDs. There are a few other features of XML Schema such as *all groups* (unordered "all-or-none" groups of subelements) whose addition to MXS is straightforward but beyond the scope of this paper.

3 A Framework for Conceptual Querying of XML

In the sequel, we develop a framework for mixing conventional syntactic XML queries with *conceptual XML queries*, *i.e.*, which refer to the abstract model MXS of a given XML Schema. First, we present a simple relational representation of the (schemaless) XML instance model (Section 3.1). Next, a "schema-aware" extension, called *conceptual instance*, is proposed which adds to the former separate conceptual level information from the MXS model (Section 3.2). This conceptual information provides the user with new means to query XML instances: Instead of relying only on the syntactic tree structure to formulate XML queries, the user can employ the *class structures* induced by the type definitions in the MXS model. In this way, the user can define access and retrieval of XML information in new, often more adequate and convenient ways (Section 3.3).

3.1 XML Instance Model

For uniquely describing XML instances, we consider the following sets: *element names* $\mathbb{E}$, *attribute names* $\mathbb{A}$, *node identifiers* (or *nodes*) $\mathbb{N}$, and *data values* (character strings) $\mathbb{D}$. Based on these, an XML document can be completely characterized using the following functions:

- $\mathsf{root} : \emptyset \to \mathbb{N}$ returns the unique *root* node of the document,

- $\mathsf{children}: \ \mathbb{N} \to [\mathbb{N}]$ returns the ordered list of *children* of a node, or the empty list [ ] in case of a leaf node,
- $\mathsf{attr}: \ \mathbb{N} \to (\mathbb{A} \to \mathbb{D})$ returns the set of *attribute-value pairs* of a node,[6]
- $\mathsf{tag}: \ \mathbb{N} \to \mathbb{E}$ returns the unique *tag* (a.k.a. *element name*) of a node, and finally
- $\mathsf{data}: \ \mathbb{N} \to \mathbb{D}$ returns the *data value* of a leaf node, or $\perp$ (undefined) for non-leaf nodes.

Clearly, a complete traversal and exploration of an XML document can be performed using these functions only, starting with root, successively applying $\mathsf{children}$ for interior nodes, data for leaf nodes, and attr and tag along the way to markup nodes with element names and attribute-value pairs, respectively. Additional navigation functions can be derived easily, *e.g.*, to move from children nodes back to the *parent*, to jump to *descendants* or *ancestors* across several levels, or to move among *siblings*. The various navigation functions are at the core of evaluating path queries and can be represented using simple relational structures.[7]

Let Int denote the positive integers. The $\mathsf{children}$ function, *e.g.*, can be modeled as a relation $\mathsf{child} \subseteq (\mathbb{N} \times \mathsf{Int} \times \mathbb{N})$ where $\mathsf{child}(n, i, n')$ holds iff the i^{th} child of node n is n':

Definition 5 (XML Instance) The *syntactic instance* of an XML document is a relational structure $\mathcal{D}$ involving the following predicates with their obvious semantics:

- $\mathsf{root}(\mathbb{N}), \quad \mathsf{child}(\mathbb{N}, \mathsf{Int}, \mathbb{N}), \quad \mathsf{attr}(\mathbb{N}, \mathbb{A}, \mathbb{D}), \quad \mathsf{tag}(\mathbb{N}, \mathbb{E}), \quad \mathsf{data}(\mathbb{N}, \mathbb{D})$ □

3.2 Schema-Aware XML Instance Model

In order to query the conceptual information contained in an XML Schema, we need a representation of its MXS model described in Section 2:

Definition 6 (Conceptual XML Instance) Let $\mathcal{D}$ be the syntactic instance of an XML document. The *conceptual instance* $\mathcal{D}^+$ is obtained by adding to $\mathcal{D}$ the following relational representations of the associated XML Schema:

- $\mathsf{restrict}(\mathbb{T}, \mathbb{T}), \quad \mathsf{extend}(\mathbb{T}, \mathbb{T}), \quad \mathsf{subtype}(\mathbb{T}, \mathbb{T}), \quad \mathsf{bind}(\mathbb{E} \cup \mathbb{A}, \mathbb{T})$

where these relations satisfy the requirements for $\mathcal{R}$, $\mathcal{E}$, $\mathcal{S}$, and $\mathcal{B}$, respectively (Section 2). □

The conceptual instance $\mathcal{D}^+$ is *virtual* in the sense that it keeps the relations pertaining to the XML Schema ($\mathsf{restrict}$, extend, ...) distinct from the relations of the actual XML instance (child, data, ...) which are typically stored in a separate XML database. Nevertheless, $\mathcal{D}^+$ can be completely typed using the following rules:

[6] a set of attribute-value pairs is a function $\mathbb{A} \to \mathbb{D}$ mapping attribute names to attribute values

[7] for efficient implementation, additional data structures such as hash indexes may be useful

$$\mathsf{elementType(N,E,T)} \leftarrow \mathsf{tag(N,E)},\ \mathsf{bind(E,T)}.$$
$$\mathsf{attributeType(N,A,T)} \leftarrow \mathsf{attr(N,A,T)},\ \mathsf{bind(A,T)}.$$

The first rule defines when an element node N with element name (tag) E has type T using the relations from $\mathcal{D}^+$; similarly for the second rule, which is used to type the attributes A of a node N.

XML Schema requires the subtyping relation $\mathcal{S}$ to be acyclic, and that each type has at most one direct parent, thereby ruling out multiple inheritance. When considering other underlying schema languages for conceptual XML queries, we may relax these and other requirements without changing our framework: Consider Example 3 and its type hierarchy in Figure 1. We can remedy the problem of the structurally equivalent but not name equivalent types `textc19BookT` and `c19textBookT` by (i) removing one of the types, say `c19textBookT`, and (ii) adding `subtype(c19bookT,textc19BookT)`, so that elements bound to `textc19BookT` are known to be subtypes of both `textBookT` *and* `c19bookT`. While this will change $\mathcal{D}^+$ according to the relaxed schema model, our basic approach discussed below remains unchanged.

3.3 From Syntactic to Conceptual Querying of XML

Types as Classes. The abstract model of an XML Schema (MXS, cf. Figure 2) constitutes a conceptual model whose type hierarchy $\mathcal{S}$ (cf. Figure 1) is defined by the type declarations of the given XML Schema. XML Schema distinguishes between subtyping via restriction ($\mathcal{R}$) and subtyping via extension ($\mathcal{E}$), since both mechanisms refine the slot structure in different ways. From a conceptual modeling perspective, their union $\mathcal{S}$ ($=\mathcal{R} \cup \mathcal{E}$) corresponds to the overall type hierarchy. As illustrated below, we argue that the type hierarchy of an XML Schema can be effectively used as a *class hierarchy*, thereby facilitating conceptual queries against XML instances. Thus, we often use the terms *type hierarchy* and *class hierarchy* synonymously. By viewing types as classes, a user may deliberately choose to ignore (completely or partially) the "DTD part" of an XML Schema, *i.e.*, the syntactic tree structure imposed by the nesting of elements, and employ the class structure as an alternative and – from a modeling perspective – often more adequate access structure.

Example 4 Assume the user wants to issue the following query on books

- *Find all books whose price is below $80.* (Q)

against instances that conform to the class hierarchy and MXS model in Figure 1 and Figure 2. □

In the following, we discuss how the user can formulate this query, in the presence or absence of certain schema information.

Exploiting DTD Information Only. It is virtually impossible for the user to formulate the query (Q) by only looking at XML instances (see Appendix B). Hence let us first assume that the user is given an XML DTD. For example, the DTD part can be obtained from an abstract MXS model by relacing the right-hand sides of element declarations with the corresponding content models of complex type definitions, while ignoring all remaining type information. For example, for `<tbook>` in Figure 2, we extract the following content model (in simplified DTD-like syntax):

```
<tbook @coverStyle> ::=
    <title>,<pubYear>,<author>+,<price>,<recommended_for>
```

Other pieces of the DTD include the content models for `<book>`, `<publication>`, `<cbook>` etc. This DTD – whether given directly, or "mined" from an XML instance – does *not* provide information how elements are conceptually related. Thus, we can express (Q) only if the user has *additional information*, say as part of the DTD documentation, or by (correctly) speculating that `<cbook>`, `<tbook>`, etc. all refer to a conceptual class "`book`". Under the assumption that the user somehow understands all the unstated (and thus for querying purposes inaccessible) class information, she may end up with the following (almost correct) XPath query:

```
//*[book OR tbook OR expbook OR cbook OR...OR USbook][price<80]
```
(Q_s)

The query (Q_s) returns all subelements (`//*`) that satisfy the subsequent conditions: the element name has to be one of {`book`, `tbook`, ...}, and the `price` subelement has to have a value less than 80.[8]

Clearly, coming up with a query like (Q_s) is a difficult and error prone task. For example, the fact that certain `<publication>` elements are also `book` objects is far from obvious when looking at the DTD and instance information. Indeed, the large disjunction over all possible tags for book objects in (Q_s) must include one more case "`publication/@xsi:type=bookT`" since those `<publication>` elements which have an attribute-value pair `xsi:type="bookT"` have to be taken into account as well.

Exploiting Conceptual Information. The previous example illustrates that the current state-of-the-art of purely syntactic querying of XML leaves ample room for improvements. In particular, the conceptual level information that is usually buried in XML schemas like the one in Appendix A can be put to good use for query formulation and evaluation: With the conceptual instance model (Definition 6) and the query translation described below in place, (Q) can be simply expressed as follows:

```
//*[ts(bookT)][price<80]
```
(Q_c)

[8] See [Wad01] for an excellent introduction to XPath queries, and the subtleties and pitfalls of their semantics.

Here, the new construct "`ts(bookT)`" refers to the type hierarchy $\mathcal{S}^*$ given by the relation $\mathsf{subtype}(\mathbb{T}, \mathbb{T})$ in the conceptual instance: the qualifier "`[ts(bookT)]`" requires that any subelement of the document ("`//*`") makes it to the subsequent query step (here: the qualifier "`[price<80]`") if and only if it has direct type `bookT` *or one of its subtypes* (see below).

Semantic Query Optimization. Another shortcoming that went unnoticed in (Q_s) and that can be exploited for (Q_c) concerns a value restriction associated with the complex type `expTextBookT` to which the `<expbook>` element has been bound:

Example 5 From the MXS model we know that the `<price>` subelement of `<expbook>` elements is of type `expPriceT`, which has been restricted to values above \$100. Given this information, we can perform a *semantic query optimization* that deliberately excludes `<expbook>` elements and rewrites (Q_c) as follows:

```
//*[ts(bookT)][NOT ts(expTextBookT)][price<80]
```
$(\mathrm{Q'}_c)$

Observe that the (left-associative) sequence of qualifiers

```
[ts(bookT)][NOT ts(expTextBookT)][price<80]
```

corresponds to a logical conjunction. A further optimization could "compile" the conjunction of the first two qualifiers `[ts(bookT)` $\wedge \neg$ `ts(expTextBookT)]` into an additional index structure that avoids iterating over expensive textbooks. □

3.4 XPathT: Adding Conceptual Information to XML Path Queries

In order to implement the conceptual style of querying XML, we first have to extend a given XML query language such as XPath or XQuery to allow the user to refer to the information in the conceptual instance $\mathcal{D}^+$. To this end, we extend a version of XPath queries as follows:

XPath queries involve *patterns* and *qualifier* expressions [Wad01]. Oversimplifying slightly, the abstract syntax of XPath includes the constructs shown in Figure 4: Intuitively, patterns p are (usually) used to *select* a set of nodes, reachable from the current *context node*, while qualifiers q are used to *filter* a given set of nodes.

For example, "$p\,[q]$" first applies the pattern p resulting in a set of selected nodes, then keeps only those nodes which satisfy the qualifier q. Note that a pattern can also be used as a qualifier: p (when used as a qualifier) is satisfied, iff it returns a non-empty result (when used as a pattern).

XPathT Syntax. We obtain *XPathT*, a "typeful" extension of XPath for conceptual querying of XML, by adding additional patterns:

- *pattern* $p ::= p\,[q] \;\mid\; /p \;\mid\; //p \;\mid \cdots \mid\; \mathtt{r}(t) \mid \mathtt{e}(t) \mid \mathtt{s}(t) \mid \mathtt{tr}(t) \mid \mathtt{te}(t) \mid \mathtt{ts}(t) \mid$ ...

- *pattern* p ::=
  $p\,[q]$ | $/p$ | $//p$ | $(p_1|p_2)$ | n | $*$ | $@n$ | $@*$ | $\texttt{parent}(p)$ | $\texttt{ancestor}(p)$ | $\ldots$
- *qualifier* q ::=
  p | q_1 AND q_2 | q_1 OR q_2 | NOT q | $\cdots$

Fig. 4. XPath patterns and qualifiers: $n \in \mathbb{E}$ and $@n \in \mathbb{A}$ denote element and attribute names, respectively.

where $t \in \mathbb{T}$ is a type name.

When evaluating a pattern, one "moves" from previously selected nodes to the newly selected nodes. Thus, patterns can be viewed as binary predicates over an underlying domain. In XPath, only the syntactic instance $\mathcal{D}$ (Definition 5), most notably the child relation is used, *i.e.*, navigation involves nodes $\mathbb{N}$, element and attribute names $\mathbb{E}$, $\mathbb{A}$, and data $\mathbb{D}$. In contrast, in XPathT, we can also employ the underlying type domain $\mathbb{T}$ of the conceptual instance $\mathcal{D}^+$ to navigate along the class hierarchy via the subtype predicate.

Figure 5 shows a set of Datalog rules, which serve as an executable specification of the semantics of XPath(T) patterns.[9] The given base predicates of the conceptual instance $\mathcal{D}^+$ only occur in the body of rules (*rhs*), while the semantics of patterns is specified via the predicates in the head of the rules (*lhs*). For example, the rule for (proper) `sibling` states that two nodes X and Y are siblings iff they are children of the same parent P (and $X \neq Y$). Similarly, the semantics of a pattern like `ts(bookT)` used above as part of the qualifier `[ts(bookT)]` can be understood from the specification of the corresponding rules in Figure 5. For example, the rules specify the binary predicate $\texttt{ts}(T_1,T_2)$ as the reflexive, transitive closure subtype* of the subtype relation, which itself is defined as the union of restrict and extend.

XPathT Semantics: Query Translation. XPathT queries can be translated into query plans in a similar way as XPath queries. Consider, for example, the query (Q'$_c$):

```
root//*[ts(bookT)][NOT ts(expTextBookT)][price<80]
```
(Q'$_c$)

Here, we have added the special pattern root which selects the unique root node of the given document. The translation of (Q'$_c$) results in a relational query plan whose predicates are specified by rules like the ones in Figure 5: starting with the root node RN, all descendant nodes DN (including the root node RN) are selected by `root//`. The "*" matches any element name $E \in \mathbb{E}$. The semantics of the qualifier `[ts(bookT)]` is to keep only those nodes from the currently selected ones, whose element name E is bound to a type T which is a direct or indirect subtype of `bookT`. Additionally, we require that T is not a

[9] A similar translation of regular path expressions over semistructured databases into executable logic rules has been given in [LHL$^+$98].

```
% descendant(ℕ,ℕ) =
descendant(X,Y) ← child(X,_,Y).
descendant(X,Y) ← child(X,_,Z),descendant(Z,Y).

% descendant_or_self(ℕ,ℕ) =
descendant_or_self(X,X).
descendant_or_self(X,Y) ← descendant(X,Y).

% ancestor(ℕ,ℕ) =
ancestor(X,Y) ← descendant(Y,X).

% sibling(ℕ,ℕ) =
sibling(X,Y) ← child(P,_,X), child(P,_,Y), not X=Y.

% nextSibling(ℕ,ℕ) =
nextSibling(X,Y) ← child(P,I,X), child(P,I+1,Y).

% firstChild(ℕ,ℕ) =
firstChild(X,Y) ← child(X,1,Y)
% lastChild(ℕ,ℕ) =
lastChild(X,Y) ← child(X,_,Y), not nextSibling(Y,_)
    ⋮
```

```
% restrict(𝕋,𝕋) =
r(BaseT,RestrictedT) ← restrict(BaseT,RestrictedT).

% extend(𝕋,𝕋) =
e(BaseT,ExtendedT) ← extend(BaseT,ExtendedT).

% subtype(𝕋,𝕋) =
s(SuperT,SubT) ← restrict(SuperT,SubT).
s(SuperT,SubT) ← extend(SuperT,SubT).

% restrict+(𝕋,𝕋) =
tr(SuperT,SubT) ← r(SuperT,SubT).
tr(SuperT,SubT) ← r(SuperT,MidT), tr(MidT, SubT).

% extend+(𝕋,𝕋) =
te(SuperT,SubT) ← e(SuperT,SubT).
te(SuperT,SubT) ← e(SuperT,MidT), te(MidT, SubT).

% subtype*(𝕋,𝕋) =
ts(SuperT,SubT) ← SuperT = SubT.
ts(SuperT,SubT) ← s(SuperT,SubT).
ts(SuperT,SubT) ← s(SuperT,MidT), ts(MidT, SubT).
    ⋮
```

Fig. 5. Executable specification of pattern semantics: XPath (left) and XPathT (right)

subtype of `expTextBookT`. The final qualifier navigates along the so-called *child axis*, retrieving nodes with element name `price` and disqualifying the selected parent node if a price value is not < 80:

Query Plan	***Patterns[Qualifiers]***
answer(DN) ←	
root(RN),	root
descendant_or_self(RN,DN),	//
tag(DN,E),	*
bind(E,T), ts(bookT,T),	[ts(bookT)]
not ts(expTextBookT,T),	[NOT ts(expTextBookT)]
child(DN,_,CN), tag(CN,price),	[price ...
data(CN,PD), $PD < 80$.	... <80]

4 Discussion and Conclusion

We have proposed a new framework for conceptual querying of XML that extends the current practice of querying purely syntactic XML instances (and thereby ignoring valuable conceptual information contained in separate XML schema documents). By means of illustrative examples we have shown that conceptual queries can often capture the desired user query more adequately and concisely than an equivalent syntactic query.

Beyond the still comparatively simple examples used in this paper, real world *data integration scenarios* can also benefit from the use of conceptual level information. Our experiences in various collaborations with domain scientists [BIR01,SDM,MPL+01,MGW+02] led to the development of a *model-based mediator* architecture that employs conceptual level information and knowledge representation techniques in order to relate data sources whose schemas are largely or completely disjoint at the syntactic level [LGM00,LGM01]. The framework and techniques presented in this paper are a step towards bridging the gap between these declarative, logic-based data integration approaches, and the syntactic approaches widely used in the databases and XML communities. XML Schema can be seen as a simple object-oriented extension of semistructured data (XML), as well as a conservative "bottom-up" approach towards a more "Semantic Web". Making the best out of the XML Schema standard includes streamlining its underlying conceptual model into an intuitive abstract model, and *using* the information contained in the schema for *query formulation* and *evaluation*.

Acknowledgements. In a related joint work, Shriram Bharath is implementing a query translation from a logic language to XPath. The authors thank him for his contributions, which will be another step towards leaving the trees.

References

[BFRW01a] A. Brown, M. Fuchs, J. Robie, and P. Wadler. MSL: A model for W3C XML Schema. In *WWW10*, Hong Kong, 2001.

[BFRW01b] A. Brown, M. Fuchs, J. Robie, and P. Wadler), editors. *XML Schema: Formal Description, W3C Working Draft*, September 2001. `www.w3.org/TR/xmlschema-formal/`.

[BIR01] Biomedical Informatics Research Network (BIRN), National Center for Research Resources, NIH. `http://birn.ncrr.nih.gov/`, 2001.

[CM01] J. Clark and M. Makoto, editors. *RELAX NG Specification*, December 2001. `www.oasis-open.org/committees/relax-ng/spec-20011203.html`.

[Cov02] R. Cover. The XML Cover Pages: XML Schemas. `xml.coverpages.org/schemas.html`, January 2002.

[FW01] P. Fankhauser and P. Wadler. XQuery Tutorial. XML 2001, Orlando, December 2001. `www.research.avayalabs.com/user/wadler/papers/xquery-tutorial/xquery-tutorial.pdf`.

[LGM00] B. Ludäscher, A. Gupta, and M. E. Martone. Model-Based Information Integration in a Neuroscience Mediator System. In *26th Intl. Conf. on Very Large Data Bases (VLDB)*, pp. 639–642, Cairo, Egypt, 2000.

[LGM01] B. Ludäscher, A. Gupta, and M. E. Martone. Model-Based Mediation with Domain Maps. In *17th Intl. Conf. on Data Engineering (ICDE)*, Heidelberg, Germany, 2001. IEEE Computer Society.

[LHL+98] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing Semistructured Data with FLORID: A Deductive Object-Oriented Perspective. *Information Systems*, 23(8):589–613, 1998. Elsevier/Pergamon.

[LPVV99] B. Ludäscher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. View Definition and DTD Inference for XML. In *Post-ICDT Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, Jerusalem, 1999. `www-rodin.inria.fr/external/ssd99/workshop.html`.

[MGW+02] M. E. Martone, A. Gupta, M. Wong, X. Qian, G. Sosinsky, S. Lamont, B. Ludäscher, and M. H. Ellisman. A Cell-Centered Database for Electron Tomographic Data. *Journal of Structural Biology*, 2002. to appear.

[MLM01a] M. Mani, D. Lee, and R. R. Muntz. Semantic Data Modeling using XML Schemas. In *Proc. 20th Intl. Conf. on Conceptual Modeling (ER)*, Yokohama, Japan, 2001.

[MLM01b] M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.

[MPL+01] M. E. Martone, S. Peltier, S. Lamont, S. K. A. Gupta, B. Ludäscher, T. Molina, and M. H. Ellisman. Increasing Access to Tomographic Resources: Web-Based Telemicroscopy and Database. In *Proceedings of the Microscopy Society of America*, 2001.

[PV00] Y. Papakonstantinou and V. Vianu. DTD Inference for Views of XML Data. In *ACM Symposium on Principles of Database Systems (PODS)*, 2000.

[SDM] Scientific Data Management Center (SDM). `http://sdm.lbl.gov/sdmcenter/`, see also `http://www.npaci.edu/online/v5.17/scidac.html`.

[Via01] V. Vianu. A Web Odyssey: from Codd to XML. In *ACM Symposium on Principles of Database Systems (PODS)*, 2001.

[Wad01] P. Wadler. A Formal Semantics of Patterns in XSLT. *Markup Languages: Theory & Practice*, June 2001. cf. `www.research.avayalabs.com/user/wadler/papers/`.

[XML01] XML Schema, W3C Recommendation. `www.w3.org/TR/xmlschema-{1,2}`, May 2001.

[XPa99] XML Path Language (XPath), Version 1.0. W3C Recommendation, `www.w3.org/TR/xpath`, 1999.

[XQu01] XQuery 1.0: An XML Query Language. W3C Working Draft, `www.w3.org/TR/xquery/`, 2001.

A Example XML Schema in Concrete Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <xsd:complexType name="publicationsT">
                <xsd:sequence>
                        <xsd:element name="pubsA" type="pubsAT"/>
                        <xsd:element name="pubsB" type="pubsBT"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="pubsAT">
                <xsd:sequence>
                        <xsd:element name="publication" type="publicationT" minOccurs="0"
                                     maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="pubsBT">
                <xsd:sequence>
                        <xsd:element name="books" type="booksT"/>
                        <xsd:element name="journal" type="journalT" minOccurs="0"
                                     maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
        <xsd:complexType name="booksT">
                <xsd:sequence>
                        <xsd:element ref="aBook" minOccurs="0" maxOccurs="unbounded"/>
```

```
        </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="publicationT" abstract="true">
        <xsd:sequence>
                <xsd:element name="title" type="xsd:string"/>
                <xsd:element name="pubYear" type="xsd:gYear"/>
        </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="journalT">
        <xsd:complexContent>
                <xsd:extension base="publicationT">
                        <xsd:sequence>
                                <xsd:element name="editor" type="xsd:string"
                                             maxOccurs="unbounded"/>
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="bookT">
        <xsd:complexContent>
                <xsd:extension base="publicationT">
                        <xsd:sequence>
                                <xsd:element name="author" type="authorT"
                                     maxOccurs="unbounded"/>
                                <xsd:element name="price" type="nnDecimalT"/>
                        </xsd:sequence>
                        <xsd:attribute name="coverStyle" type="coverStyleT"/>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="textBookT">
        <xsd:complexContent>
                <xsd:extension base="bookT">
                        <xsd:sequence>
                                <xsd:element name="recommended_for"
                                             type="xsd:string"/>
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="c19bookT">
        <xsd:complexContent>
                <xsd:restriction base="bookT">
                        <xsd:sequence>
                                <xsd:element name="title" type="xsd:string"/>
                                <xsd:element name="author" type="authorT"
                                             maxOccurs="unbounded"/>
                                <xsd:element name="pubYear" type="c19gYearT"/>
                                <xsd:element name="price" type="nnDecimalT"/>
                        </xsd:sequence>
                        <xsd:attribute name="coverStyle" type="coverStyleT"
                                       default="hardcover"/>
                </xsd:restriction>
        </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="USBookT">
        <xsd:complexContent>
                <xsd:restriction base="bookT">
                        <xsd:sequence>
                                <xsd:element name="title" type="xsd:string"/>
                                <xsd:element name="author" type="USAuthorT"
                                             maxOccurs="unbounded"/>
                                <xsd:element name="pubYear" type="xsd:gYear"/>
                                <xsd:element name="price" type="nnDecimalT"/>
                        </xsd:sequence>
                        <xsd:attribute name="coverStyle" type="coverStyleT"
                                       fixed="hardcover"/>
                </xsd:restriction>
        </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="expTextBookT">
        <xsd:complexContent>
                <xsd:restriction base="textBookT">
                        <xsd:sequence>
                                <xsd:element name="title" type="xsd:string"/>
```

```
                                <xsd:element name="author" type="authorT"
                                             maxOccurs="unbounded"/>
                                <xsd:element name="pubYear" type="xsd:gYear"/>
                                <xsd:element name="price" type="expPriceT"/>
                                <xsd:element name="recommended_for" type="xsd:string"/>
                        </xsd:sequence>
                        <xsd:attribute name="coverStyle" type="coverStyleT"/>
                </xsd:restriction>
        </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="textc19BookT">
        <xsd:complexContent>
                <xsd:restriction base="textBookT">
                        <xsd:sequence>
                                <xsd:element name="title" type="xsd:string"/>
                                <xsd:element name="author" type="authorT"
                                             maxOccurs="unbounded"/>
                                <xsd:element name="pubYear" type="c19gYearT"/>
                                <xsd:element name="price" type="nnDecimalT"/>
                                <xsd:element name="recommended_for" type="xsd:string"/>
                        </xsd:sequence>
                        <xsd:attribute name="coverStyle" type="coverStyleT"/>
                </xsd:restriction>
        </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="c19textBookT">
        <xsd:complexContent>
                <xsd:extension base="c19bookT">
                        <xsd:sequence>
                                <xsd:element name="recommended_for"
                                     type="xsd:string"/>
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="authorT">
        <xsd:sequence>
                <xsd:element name="name" type="xsd:string"/>
        </xsd:sequence>
        <xsd:attribute name="countryOfBirth" type="xsd:string"/>
</xsd:complexType>
<xsd:complexType name="USAuthorT">
        <xsd:complexContent>
                <xsd:restriction base="authorT">
                        <xsd:sequence>
                                <xsd:element name="name" type="xsd:string"/>
                        </xsd:sequence>
                        <xsd:attribute name="countryOfBirth" type="xsd:string"
                             fixed="USA"/>
                </xsd:restriction>
        </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="lateAuthorT">
        <xsd:complexContent>
                <xsd:extension base="authorT">
                        <xsd:sequence>
                                <xsd:element name="deathYear" type="xsd:gYear"/>
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="nnDecimalT">
        <xsd:restriction base="xsd:decimal">
                <xsd:minInclusive value="0.0"/>
        </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="coverStyleT">
        <xsd:restriction base="xsd:string">
                <xsd:enumeration value="hardcover"/>
                <xsd:enumeration value="paperback"/>
        </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="expPriceT">
        <xsd:restriction base="nnDecimalT">
                <xsd:minExclusive value="100.0"/>
```

```
			</xsd:restriction>
		</xsd:simpleType>
		<xsd:simpleType name="c19gYearT">
			<xsd:restriction base="xsd:gYear">
				<xsd:minInclusive value="1800"/>
				<xsd:maxExclusive value="1900"/>
			</xsd:restriction>
		</xsd:simpleType>
		<xsd:element name="publications" type="publicationsT"/>
		<xsd:element name="aBook" type="bookT" abstract="true"/>
		<xsd:element name="book" type="bookT" substitutionGroup="aBook"/>
		<xsd:element name="expbook" type="expTextBookT" substitutionGroup="aBook"/>
		<xsd:element name="tbook" type="textBookT" substitutionGroup="aBook"/>
		<xsd:element name="cbook" type="c19bookT" substitutionGroup="aBook"/>
		<xsd:element name="ctbook" type="c19textBookT" substitutionGroup="aBook"/>
		<xsd:element name="tcbook" type="textc19BookT" substitutionGroup="aBook"/>
		<xsd:element name="USbook" type="USBookT" substitutionGroup="aBook"/>
</xsd:schema>
```

B Example XML Instance

```
<?xml version="1.0" encoding="UTF-8"?>
<publications xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
		xsi:noNamespaceSchemaLocation="S:\XMLDM\PaperSchema0119.xsd">
	<pubsA>
		<publication xsi:type="journalT">
			<title>ACM TODS</title>
			<pubYear>1999</pubYear>
			<editor>ACM</editor>
		</publication>
		<publication xsi:type="bookT" coverStyle="paperback">
			<title> The Unbearable Lightness of Being </title>
			<pubYear>1984</pubYear>
			<author countryOfBirth="Czech">
				<name>Milan Kundera</name>
			</author>
			<price>10.40</price>
		</publication>
	</pubsA>
	<pubsB>
		<books>
			<book coverStyle="paperback">
				<title> The Book of Laughter and Forgetting </title>
				<pubYear>1979</pubYear>
				<author countryOfBirth="Czech">
					<name>Milan Kundera</name>
				</author>
				<price>10.40</price>
			</book>
			<expbook coverStyle="hardcover">
				<title>The Dictionary of Art  </title>
				<author countryOfBirth="USA" xsi:type="USAuthorT">
					<name>Jane Shoaf Turner</name>
				</author>
				<pubYear>1996</pubYear>
				<price>8800.00</price>
				<recommended_for> Art students</recommended_for>
			</expbook>
			<tbook coverStyle="hardcover">
				<title> Computer Organization and Design:
					The Hardware/Software Interface </title>
				<pubYear>1997</pubYear>
				<author countryOfBirth="USA" xsi:type="USAuthorT">
					<name>David A. Patterson</name>
				</author>
				<author countryOfBirth="USA" xsi:type="USAuthorT">
					<name>John L. Hennessy</name>
				</author>
				<price>79.95</price>
				<recommended_for>Computer Science and Engineering
						Students</recommended_for>
			</tbook>
			<book coverStyle="paperback">
```

```
                <title> Faust </title>
                <pubYear>1808</pubYear>
                <author countryOfBirth="Germany" xsi:type="lateAuthorT">
                        <name>J.W. von Goethe</name>
                        <deathYear>1832</deathYear>
                </author>
                <price>15.16</price>
            </book>
            <cbook>
                <title>Vatan Yahut Silistre </title>
                <author countryOfBirth="Turkey" xsi:type="lateAuthorT">
                        <name>namik Kemal</name>
                        <deathYear>1916</deathYear>
                </author>
                <pubYear>1873</pubYear>
                <price>25.67</price>
            </cbook>
        </books>
        <journal>
            <title>ACM TODS</title>
            <pubYear>1999</pubYear>
            <editor>ACM</editor>
        </journal>
        <journal>
            <title>TKDE</title>
            <pubYear>2000</pubYear>
            <editor>IEEE</editor>
        </journal>
    </pubsB>
</publications>
```