Available online at www.sciencedirect.com

ScienceDirect

Data & Knowledge Engineering 64 (2008) 134–169

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Filtering unsatisfiable XPath queries

Jinghua Groppe, Sven Groppe *

University of Lübeck, Institute of Information Systems (IFIS), Ratzeburger Allee 160, D-23538 Lübeck, Germany

Received 22 June 2007; accepted 22 June 2007
Available online 22 August 2007

Abstract

The satisfiability test checks, whether or not the evaluation of a query returns the empty set for any input document, and can be used in query optimization for avoiding the submission and the computation of unsatisfiable queries. Thus, applying the satisfiability test before executing a query can save processing time and query costs. We focus on the satisfiability problem for queries formulated in the XML query language *XPath*, and propose a schema-based approach to the satisfiability test of XPath queries, which checks whether or not an XPath query conforms to the constraints in a given schema. If an XPath query does not conform to the constraints given in the schema, the evaluation of the query will return an empty result for any valid XML document. Thus, the XPath query is unsatisfiable. We present a complexity analysis of our approach, which proves that our approach is efficient for typical cases. We present an experimental analysis of our developed prototype, which shows the optimization potential of avoiding the evaluation of unsatisfiable queries.

Keywords: Queries; XML; XPath; Satisfiability tester; Query optimization

1. Introduction

XPath (see [29,30]) is either a standalone XML query language or is embedded in other XML languages (e.g. XSLT, XQuery, XLink and XPointer) for specifying node sets in XML documents. An important issue in XPath evaluation is the satisfiability problem of XPath queries. An XPath query `Q` is unsatisfiable, if the evaluation of `Q` on any XML document returns every time an empty result. Therefore, the satisfiability test of XPath queries plays a critical role in query optimization. The application of the satisfiability test can avoid the submission and the unnecessary evaluation of unsatisfiable queries, and thus saving processing time and query cost. As well as for query optimization, the XPath satisfiablity test is also important for consistency problems, e.g. XML access control [6] and type-checking of transformations [19]. Therefore, many research efforts focus on the satisfiability test of XPath queries with or without respect to schemas, e.g. [2,10,11,14,17,18].

In the absence of schemas, the satisfiability test can detect that the structure properties of XPath queries are inconsistent with the XML data model (e.g. [14]). For example, the XPath query `Q1 = /parent::a` is unsatisfiable, because the document node has no parent node according to the XML data model. The query

* Corresponding author.
E-mail addresses: Jinghua.Groppe@ifis.uni-luebeck.de (J. Groppe), groppe@ifis.uni-luebeck.de (S. Groppe).

0169-023X/$ - see front matter
doi:10.1016/j.datak.2007.06.018

`Q2 = //regions/america` is tested as a satisfiable XPath query without respect to a schema. However, according to a given schema, e.g. the schema given in [8], the element `regions` can have children, which are called `namerica` and `samerica`, but does not have children with name `america`. Therefore, `Q2` is unsatisfiable with respect to the given schema. Thus, we can detect more errors in XPath queries if we additionally consider schema information. Therefore, we focus on the satisfiability test of XPath in the presence of schemas.

The most widely used schema languages are XML Schema (see [31,32]) and DTD (see [28]). In this paper, we focus on XML Schema for the definition of schemas. As well as imposing the constraints of the structure and semantics on XML documents as DTDs do, the XML Schema language provides powerful capabilities for specifying data types on elements and attributes, most of which are not expressible in DTDs. The XML Schema language provides a large number of built in simple types and allows deriving new types for the values of elements and attributes, which are only specified to be character data in DTDs. Thus, if the types of values of elements or attributes in an XPath query do not conform to constraints specified in the XML Schema definition, the XPath query selects an empty set of nodes for any valid XML document. For example, the query `meeting[@date = '01-05-06']` does not retrieve anything if the type of the attribute `date` is declared to have the format `DD-MM-`*`YYYY`*. Therefore, the powerful data-typing facilities supported by XML Schema provide another dimension for the satisfiability test of XPath queries. Since XML Schema can express more restrictions than a DTD, a DTD can be easily transformed into an XML Schema representation, but in general, an XML Schema definition cannot be transformed into a DTD without loosing information. To the best of our knowledge, existing work only deals with DTDs except our previous contributions (see [10,11]).

Our schema-based approach checks whether or not an XPath query `Q` conforms to the structure, semantics, data type and occurrence constraints in a given XML schema definition `S` by evaluating `Q` on `S` rather than the instance documents of `S`. If `Q` does not conform to the constraints of `S`, `Q` cannot be evaluated completely on `S`, and thus `Q` is unsatisfiable. For schemas, our approach supports recursive as well as non-recursive schemas, considers a significant part of the XML Schema language and allows arbitrary nesting and references of model groups. For XPath, our approach allows all XPath axes and negation operations in predicates. The satisfiability test for the XPath subset supported by our approach in the presence of the schemas supported by our approach is undecidable (see [2]). Therefore, we present an incomplete, but fast satisfiability tester, i.e. if our tester returns *unsatisfiable*, then we are sure that the XPath query is unsatisfiable, but if our tester returns *maybe satisfiable*, then the XPath query may be satisfiable or may be unsatisfiable. Note that we do not loose correctness in the proposed application scenarios of our satisfiability tester when using an incomplete tester.

This paper is an extended version of [10,11]. We extend the contributions of [10,11] by significantly extending the supported subset (see Section 2.2) of the XML Schema language, allowing various content models of elements and arbitrary nesting of model groups; by supporting the type-checking of values of elements and attributes (see Section 4.5) and the checking of occurrence constraints (see Section 4.6); by integrating all new contributions into the prototype of [11] and by additional experiments (see Section 6).

The rest of the paper is organized as follows: Section 2 describes the supported subsets of XPath and XML Schema. Section 3 develops a data model for XML Schema. This data model for XML Schema is the basis for our XPath–XSchema evaluator (see Section 4), which evaluates XPath queries on XML Schema definitions in order to compute the schema paths of the queries. Section 4 also includes a complexity analysis of the approach. Section 5 discusses the satisfiability test of XPath based on the schema paths. We present a comprehensive performance analysis in Section 6. Section 7 deals with further related work. We end up with the summary and conclusions in Section 8.

2. XPath and XML Schema

In this section, we present the subset of the XPath language and the subset of XML Schema language supported in this work.

2.1. XPath

XPath (see [29,30]) is a query language for XML data. In this paper, we consider the basic properties of the XPath language, and the abstract syntax of the supported XPath subset is defined in EBNF as follows:

```
Pattern     e ::= e|e|/e|e/e|e[q]|a::n.
Predicate   q ::= e|e = C | e = e|q and q|q or q|not(q)|(q).
Axis        a ::= child|attribute|descendant|self|following|
                  preceding|parent|ancestor|DoS|AoS|FS|PS.
Nodetest    n ::= label| * |node()|text().
```

where `label` is an element or attribute name and `C` is a literal, i.e. a string or a number. Furthermore, we write `DoS` for `descendant-or-self`, `AoS` for `ancestor-or-self`, `FS` for `following-sibling` and `PS` for `preceding-sibling`.

The semantics of each pattern is defined in terms of the semantics of its sub-patterns. The smallest pattern is called a *location step* `a::n[`q_1`]...[`q_i`]`, which consists of an *axis* `a` and a *nodetest* `n` with or without *predicates* $q_1, \ldots, q_i$, e.g. `child::title` and `descendant::section[child::*]`. Axis and nodetest of a location step select a set of XML nodes relevant to a *context* node, which is further filtered by the predicates. Location steps are separated by the token '/', and the nodes selected by a location step are the context nodes of the next location step.

The XPath language also defines several abbreviations, e.g. `/child::a` is abbreviated to `/a`, and // represents `/descendant-or-self::node()/`. Whenever possible, we will use the abbreviated syntax in this paper as more compact representation.

2.2. XML Schema

XML Schema (see [31,32]) is a language for defining a class of XML documents, called *instance documents* of the schema. We call a schema, which is formulated in the XML Schema language, an *XML Schema definition* (or *XSchema* as short name), which is itself an XML document. An XSchema defines the structure of the instance documents, the vocabulary (e.g. the element and attribute names used), and the data types of elements and attributes. In this paper, we support a significant subset of the XML Schema language, where a given XSchema must conform to the following EBNF rules.

```
XSchema ::= <schema> (simpleTypeD | complexTypeD | groupD | attributeGroupD | elementD | attributeD)* </schema>.
simpleTypeD ::= <simpleType (name=NCName)?> restrictionSimpleTypeD </simpleType>.
restrictionSimpleTypeD ::= <restriction base=QName> facet* </restriction>.
facet ::=  <minExclusive  value=Value /> | <minInclusive value=Value /> |
           <maxExlusive value=Value /> | <maxInclusive value=Value /> |
           <totalDigits value=Value /> | <fractionDigits value=Value /> |
           <length value=Value /> | <minLength value=Value /> |
           <maxLength value=Value /> | <enumeration value=Value /> |
           <whiteSpace value=Value /> | <pattern value=Value />.
complexTypeD ::= <complexType (mixed=Boolean)? (name=NCName)?> (simpleContentD |  complexContentD |
        ((groupD | allD | choiceD | sequenceD)? (attributeD | attributeGroupD)*)) </complexType>.
simpleContentD ::= <simpleContent> (restrictionSimpleContent | extensionSimpleContent) </simpleContent>.
complexContentD ::= <complexContent (mixed= Boolean)?>
        (restrictionComplexContent | extensionComplexContent) </complexContent>.
restrictionSimpleContent ::= <restriction base=QName> facet* (attributeD | attributeGroupD)* </restriction>.
extensionSimpleContent ::= <extension base=QName> (attributeD | attributeGroupD)* </extension>
restrictionComplexContent ::= <restriction base= 'anyType'> (attributeD | attributeGroupD)* </restriction>.
extensionComplexContent ::= <extension base=QName> ((groupD | allD | choiceD | sequenceD)?
        (attributeD | attributeGroupD)*) </restriction>
groupD ::= <group (maxOccurs=(nonNegativeInteger | 'unbounded'))? (minOccurs=nonNegativeInteger)?
        (name=NCName | ref=QName)?> (allD | choiceD | sequenceD)? </group>.
attributeGroupD ::= <attributeGroup (name=NCName | ref=QName)?> (attributeD | attributeGroupD)* </attributeGroup>
allD ::= <all maxOccurs= '1' minOccurs=('0' | '1')> elementD* </all>
choiceD ::= <choice (maxOccurs=(nonNegativeInteger | 'unbounded'))?
        (minOccurs=nonNegativeInteger)?> (elementD | groupD | choiceD | sequenceD)* </choice>.
sequenceD ::= <sequence (maxOccurs=(nonNegativeInteger | 'unbounded'))? (minOccurs=nonNegativeInteger)?>
        (elementD | groupD | choiceD | sequenceD)* </sequence>.
elementD ::= <element (fixed=string)? (maxOccurs=(nonNegativeInteger | 'unbounded'))?
        (minOccurs=nonNegativeInteger)? (Name=NCName | ref=QName)? (type=QName)?>
        (simpleTypeD | complexTypeD)? </element>
attributeD ::= <attribute (fixed=string)? (name=NCName | ref=QName)? (type=QName)?
        (use=('optional' | 'prohibited' | 'required'))?>  simpleTypeD? </attribute>
```

```
(D1)   <schema>
(D2)     <group name='journalArticle'>
(D3)       <sequence>
(D4)         <element name='article' minOccurs='1' maxOccurs='1'>
(D5)           <complexType>
(D6)             <sequence>
(D7)               <element name='title' minOccurs='0' maxOccurs='1' type='string'/>
(D8)               <element name='year' minOccurs='0' maxOccurs='1' type='string'/>
(D9)               <element name='journal' minOccurs='0' maxOccurs='1' type='string'/>
(D10)              <element name='refs' minOccurs='0' maxOccurs='1'>
(D11)                <complexType>
(D12)                  <group ref='journalArticle'  minOccurs='0' maxOccurs='unbounded'/>
                     </complexType>
                   </element>
                 </sequence>
               </complexType>
             </element>
           </sequence>
         </group>

(D13)    <element name='bib'>
(D14)      <complexType>
(D15)        <group ref='journalArticle' minOccurs='0' maxOccurs='unbounded'/>
           </complexType>
         </element>
       </schema>
```

Fig. 1. An XML Schema definition bib.xsd.

```
<bib>
   <article>
         <title> My second article </title>
         <year> 2007 </year>
         <journal> Well-Known Journal (WKJ) </journal>
         <refs>
              <article>
                   <title> My first article </title>
                   <year> 2006 </year>
                   <journal> Well-Known Journal (WKJ) </journal>
                   <refs/>
              </article>
         </refs>
   </article>
</bib>
```

Fig. 2. Example XML document conforming to the XML Schema definition of Fig. 1.

where QName (see [32]) is an XML qualified name, NCName (see [32]) is an XML non-colonized name, Boolean is a boolean value, i.e. true or false, nonNegativeInteger is a non-negative integer, string is a character string, and Value is a value, e.g. a number or a string.

Example 1. Fig. 1 presents an example of an XML Schema definition bib.xsd, which describes a schema for XML documents containing information about journal articles. Fig. 2 contains an example XML document, which conforms to the XML Schema definition of Fig. 1.

3. Data model for the XML Schema language

Based-on the data model for the XML language given by Waddler [24], we develop a data model for XML Schema for identifying the navigation paths of XPath queries on an XML Schema definition.

3.1. Notations

The following notations on sets, relationships and sequences are used to model the XML Schema definition, and are also used to model the schema path (see Section 4). `Set(T)` (or `Sequence(T)`, respectively) indicates the type of a set (or of a sequence, respectively) the entries of which are of type `T`. We write $\emptyset$ for the empty set, $\in$ for membership and $\cup$ for the union of sets. We express the signature of a function *f* by $f{:}\mathtt{T}_1 \rightarrow \mathtt{T}_2$, where $\mathtt{T}_1$ is the type of the domain and $\mathtt{T}_2$ is the type of the co-domain. Note that a type `T` can be a simple type, e.g. an XSchema node (`Node`), an XPath expression (`XPath`) or a node test (`NodeTest`). Furthermore, `T` can be the type of a set the entries of which are of a type $\mathtt{T}_1$, i.e. $\mathtt{Set}(\mathtt{T}_1)$, the type of a sequence the entries of which are of a type $\mathtt{T}_1$, i.e. $\mathtt{Sequence}(\mathtt{T}_1)$, or the cross-product of two or more types, e.g. $\mathtt{T}_1 \times \mathtt{T}_2$. The transitive closure f^+ and reflexive transitive closure f^* of a function $f{:}\mathtt{T} \rightarrow \mathtt{Set(T)}$ are defined as follows:

$$f^{\mathrm{n}}(\mathrm{x}) = \{\mathrm{z} \mid \mathrm{y} \in f^{\mathrm{n}-1}(x) \wedge \mathrm{z} \in f(\mathrm{y})\}, \quad \text{where } f^0(\mathrm{x}) = \{\mathrm{x}\} \quad \text{and} \quad f^1(x) = f(x)$$
$$f^+(\mathrm{x}) = \cup_{\mathrm{n}=1}^{\infty} f^{\mathrm{n}}(\mathrm{x})$$
$$f^*(\mathrm{x}) = \cup_{\mathrm{n}=0}^{\infty} f^{n}(\mathrm{x})$$

We write $(\mathrm{x}_1, \ldots, \mathrm{x}_{\mathrm{m}})$ for a sequence of entries $\mathrm{x}_1, \ldots, \mathrm{x}_{\mathrm{m}}$. We use the operator + to concatenate two sequences, e.g. $(\mathrm{x}_1, \ldots, \mathrm{x}_{\mathrm{m}}) + (\mathrm{y}_1, \ldots, \mathrm{y}_{\mathrm{n}}) = (\mathrm{x}_1, \ldots, \mathrm{x}_{\mathrm{m}}, \mathrm{y}_1, \ldots, \mathrm{y}_{\mathrm{n}})$. Let `s` be a sequence, then we write `s[k]` for the *k*th entry of the sequence `s`, and write $|\mathrm{s}|$ for the length of `s`, i.e. the number of entries in `s`. Thus, `s[1]` indicates the first entry of `s` and `s[|s|]` indicates the last entry of `s`, `s[|s|-1]` indicates the pre-last entry of `s`, and so on. Furthermore, we also call a node in an XML Schema definition an *XSchema node*.

3.2. Concepts

An XML Schema definition is a set of nodes of type `Node`. There are three specific `Node` types in an XML Schema definition, which are associated with *instance element*, *instance attribute* and *instance text* nodes of the XML Schema definition: `iElement`, `iAttribute` and `iText`. Accordingly, we define three functions with signature `Node` → `Boolean` to test the type of a node: `isiElement`, `isiAttribute`, and `isiText`, which return true if the type of the given node is of type `iElement`, `iAttribute` or `iText`, respectively, otherwise false.

Definition 1 (*instance nodes*). The *instance nodes* of an XML Schema definition are

- ⟨`element name = N...`⟩ (which is an *instance element* node of type `iElement`),
- ⟨`attribute name = N...`⟩ (which is an *instance attribute* node of type `iAttribute`),
- attribute node `type = T` of nodes ⟨`element type = T...`⟩, which we denote as @⟨`type = T`⟩ (which is an *instance text* node of type `iText`, if `T` is a built-in simple type),
- ⟨`simpleType...`⟩ (which is an *instance text* node of type `iText`),
- ⟨`complexType mixed = 'true'...`⟩ (which is an *instance text* node of type `iText`),
- ⟨`simpleContent...`⟩ (which is an *instance text* node of type `iText`), and
- ⟨`complexContent mixed = 'true'...`⟩ (which is an *instance text* node of type `iText`).

Definition 2 (*instance child node*). Let `x` and `y` be two XSchema nodes of type *iElement*. If the element defined in `y` can appear in instance XML documents as a child of the element defined in `x`, then `y` is an instance child node of `x`.

Definition 3 (*instance text node*). Let `x` be an XSchema node of type *iElement*, and `y` be an XSchema node of type *iText*. If `y` is an attribute node of `x` or a node that is used to define the type of the element declared in `x`, then `y` is an instance text node of `x`.

Definition 4 (*instance attribute node*). Let `x` be an XSchema node of type *iElement*, and y be an XSchema node of type *iAttribute*. If the attribute defined in `y` can appear in instance XML documents as an attribute of the element defined in `x`, then `y` is an instance attribute node of `x`.

Definition 5 (*instance parent node*). Let x be an XSchema node of type *iElement*, and y be either an instance child node or an instance text node or an instance attribute node of x, then x is the instance parent node of y.

Definition 6 (*instance sibling*, *instance preceding sibling and instance following sibling node*). Let x be an XSchema node of type *iElement* or *iText*, and y be an XSchema node of type *iElement* or *iText*. If the element that is defined in x or the text whose data type is defined in x can appear in valid XML documents as a sibling, or a preceding sibling, or a following sibling, respectively, of the element that is defined in y or the text whose datatype is defined in y, then x is an instance sibling node, or an instance preceding sibling node, or an instance following sibling node, respectively, of y.

Definition 7 (*succeeding node*). A node N2 in an XML Schema definition is a *succeeding node* of a node N1 in the XML Schema definition if

- N2 is a child node of N1, or
- N1 = ⟨element type = N...⟩ and N2 = ⟨simpleType name = N...⟩ with the same N, or
- N1 = ⟨attribute type = N...⟩ and N2 = ⟨simpleType name = N...⟩ with the same N, or
- N1 = ⟨element type = N...⟩ and N2 = ⟨complexType name = N...⟩ with the same N, or
- N1 = ⟨element ref = N...⟩ and N2 = ⟨element name = N...⟩ with the same N, or
- N1 = ⟨attribute ref = N...⟩ and N2 = ⟨attribute name = N...⟩ with the same N, or
- N1 = ⟨group ref = N...⟩ and N2 = ⟨group name = N...⟩ with the same N, or
- N1 = ⟨attributeGroup ref = N⟩ and N2 = ⟨attributeGroup name = N⟩ with the same N, or
- N1 = ⟨restriction base = N⟩ and N2 = ⟨simpleType name = N...⟩ with the same N, or
- N1 = ⟨extension base = N⟩ and N2 = ⟨simpleType name = N...⟩ with the same N, or
- N1 = ⟨extension base = N⟩ and N2 = ⟨complexType name = N...⟩ with the same N.

Definition 8 (*preceding node*). Node N1 in an XML Schema definition is a *preceding node* of a node N2 in the XML Schema definition if N2 is a *succeeding node* of N1.

3.3. Functions

Fig. 3 defines the data model of the XML Schema language, which consists of a group of functions. These functions relate an XSchema node to a set of XSchema nodes or to a set of sequences of XSchema nodes, or relate a sequence of XSchema nodes to a set of sequences of XSchema nodes, represented in comprehension notation (see [24]).

The function child: Node → Set(Node) relates an XSchema node to all its child nodes; the function succeeding: Node → Set(Node) relates an XSchema node to all its *succeeding* nodes; the function preceding: Node → Set(Node) relates an XSchema node to all its *preceding* nodes.

iChild: Node → Set(Sequence(Node)), which is defined to find the instance child nodes of type *iElement* of an XScheme node N, relates the XSchema node N to a set of XSchema node sequences, i.e. if y ∈ iChild(N), then y[1] = N and y[|y|] is an instance child node of N. Other nodes in y are the intermediate nodes visited when searching for y[|y|] of y[1], i.e. ones that belong to both succeeding$^+$(y[1]) and preceding$^+$(y[|y|]). Some of them may be the declaration nodes of model groups, which control the occurrence of y[|y|], and the occurrence order of y[|y|] and its instance sibling nodes in an instance XML document. iAttributeChild: Node → Set(Sequence(Node)), which is defined to find the instance attribute nodes of an XSchema node N, relates the node N to a set of node sequences, i.e. if y ∈ iAttribueChild(N), then y[1] = N and y[|y|] is an instance attribute node of N. Other nodes in y are the intermediate nodes visited when searching for y[|y|] of y[1], i.e. ones that belong to both succeeding$^+$(y[1]) and preceding$^+$(y[|y|]). The auxiliary function iChild-helper: Node → Set(Sequence(Node)) helps iChild(N) and iAttributeChild(N) to find the corresponding nodes, and returns all the node sequences visited before the instance child nodes and instance attribute nodes of the XSchema node N.

- child(N) = { N1 | N1 is a child node of N }
- succeeding(N) = { N1 | N1 is a *succeeding* node of N }
- preceding(N) = { N1 | N1 is a *preceding* node of N }
- iChild-helper(N) = $\cup_{i=0}^{\infty} S_i$, where
  S_0 = { (N) },
  S_i = { y+(N1) | y∈ S_{i-1} ∧ N1∈ succeeding(y[|y|]) ∧
  ¬isiElement(N1) ∧ ¬isiAttribute(N1) }
- iChild(N) = { y+(N1) | y∈ iChild-helper(N) ∧ N1∈ succeeding(y[|y|]) ∧ isiElement(N1) }
- iAttributeChild(N) = { y+(N1) | y∈ iChild-helper(N) ∧ N1∈ succeeding(y[|y|]) ∧ isiAttribute(N1) }
- iText-helper(N) = $\cup_{i=0}^{\infty} R_i$, where
  R_0 = { (N) },
  R_i = { y+(N1) | y∈ R_{i-1} ∧ N'= y[|y|] ∧ ¬isiText(N') ∧ ¬isiAttribute(N') ∧ N'≠<complexType…> ∧
  (N'≠<element type=T…> ∨ (N'=<element type=T> ∧ ¬built-in(T))) ∧
  N1∈ succeeding(N') }
- iTextChild(N) = { y | (y∈ iText-helper(N) ∧ isiText(y[|y|])) ∨
  (y=z+(N1) ∧ z∈ iText-helper(N) ∧ N'= z[|z|] ∧ ¬isiText(N') ∧ isiText(N1) ∧ (
  (N'=<element type=T…> ∧ N1=attributeNode(N', type=T)) ∨
  (N'=<complexType…> ∧ N1∈ succeding(N'))))}
- iPS(x) = { y | (y∈ iChild(x[1]) ∨ y∈ iTextChild(x[1])) ∧
  y[|y|]≠@<type=T> ∧ y[y]≠<simpleType…> ∧ y[y]≠ <simpleContent…> ∧ (
  (y[|y|]=<complexType mixed='true'…> ∨ y[|y|]=<complexContent mixed='true'…>)
  ∨
  (x[|x|]=<complexType mixed='true'…> ∨ x[|x|]=<complexContent mixed='true'…>)
  ∨
  (x=y ∧ ∃i∈{2, 3, ..., |x|}: attribute(x[i], 'maxOccurs')>1)
  ∨
  (∀i∈{1, …, k}: x[i]=y[i] ∧ x[k+1]≠y[k+1] ∧ k<min(|x|, |y|) ∧ (
  x[k]=<all> ∨
  ∃i∈{2, 3, ..., k}: attribute(x[i], 'maxOccurs')>1 ∨
  (y[k+1]<<x[k+1] ∧ ∀i∈{2, 3, ..., k}: (
  x[i]=<sequence maxOccurs=1…> ∨
  x[i]=<choice maxOccurs=1…> ∨
  x[i]=<group maxOccurs=1…> ∨
  (x[i]≠<sequence…> ∧ x[i]≠<choice…> ∧ x[i]≠<group…> ∧ x[i]≠<all…>)) ∧
  x[k]≠<choice…>)))))}
- iFS(x) = { y | (y∈ iChild(x[1]) ∨ y∈ iTextChild(x[1])) ∧
  y[|y|]≠@<type=T> ∧ y[y]≠<simpleType…> ∧ y[y]≠ <simpleContent…> ∧ (
  (y[|y|]=<complexType mixed='true'…> ∨ y[|y|]=<complexContent mixed='true'…>)
  ∨
  (x[|x|]=<complexType mixed='true'…> ∨ x[|x|]=<complexContent mixed='true'…>)
  ∨
  (x=y ∧ ∃i∈{2, 3, ..., |x|}: attribute(x[i], 'maxOccurs')>1)
  ∨
  (∀i∈{1, …, k}: x[i]=y[i] ∧ x[k+1]≠y[k+1] ∧ k<min(|x|, |y|) ∧ (
  x[k]=<all> ∨
  ∃i∈{2, 3, ..., k}: attribute(x[i], 'maxOccurs')>1 ∨
  (x[k+1]<<y[k+1] ∧ ∀i∈{2, 3, ..., k}: (
  x[i]=<sequence maxOccurs=1…> ∨
  x[i]=<choice maxOccurs=1…> ∨
  x[i]=<group maxOccurs=1…> ∨
  (x[i]≠<sequence…> ∧ x[i]≠<choice…> ∧ x[i]≠<group…> ∧ x[i]≠<all…>)) ∧
  x[k]≠<choice…>)))))}

Fig. 3. A data model of the XML Schema language for evaluating XPath queries on XML Schema definitions.

`iTextChild: Node → Set(Sequence(Node))` is defined to find the instance text nodes of an XSchema node `N`, and relates the node `N` to a set of node sequences. Let `y ∈ iTextChild(N)`, then `y[1] = N` and `y[|y|]` is an instance text node of `N`. The nodes between `y[1]` and `y[|y|]` are the intermediate nodes visited when searching for `y[|y|]` of `y[1]`, i.e. the nodes that belong to both `succeeding`$^+$`(y[1])` and `preceding`$^+$`(y[|y|])`. The auxiliary function `attributeNode(N', type = T)` in `iTextChild(N)` returns the attribute node `type = T` of the node N'. The XML data model defines that an element of simple type must have and only has a text node, and that an element of complex type can either have one or more text nodes or have no text node at all. XML Schema specifies whether or not an element of complex type has text nodes, but does not specify the number of the text nodes. Therefore, we only need to take care whether or not an XSchema node has instance text nodes, and we only need to find one instance text node but not all the instance text nodes of an XSchema node. We achieve these goals by using the auxiliary function `iText-helper`: `Node` → `Set(Sequence(Node))`.

If N of `iText-helper(N)` declares an element of simple type, then N must have instance text nodes, which are either the attribute node `type = T` of N if T is a built-in simple type, or the nodes ⟨`simpleType`...⟩ in $\texttt{succeeding}^{+}$(N). If N declares an element e of complex type, then there must exist a node of a complex type declaration, i.e. D = ⟨`complexType`...⟩, which is used to define the type of the element e, i.e. D is a node in $\texttt{succeeding}^{+}$(N). If D contains the construct `mixed = 'true'`, then D is an instance text node of N. If D is not an instance text node of N, but D has a child node of ⟨`simpleContent`...⟩ or ⟨`complexContent mixed = 'true'`...⟩, then the child node of D is the instance text node of N. If D does not have such a child, then N does not have instance text nodes. Let y ∈ `iText-helper(N)`, then y[1] = N, and y[|y|] is either an instance text node, or the node ⟨`complexType`...⟩, or a node visited before an instance text node or before an instance attribute node or before the node ⟨`complexType`...⟩ of N. The auxiliary function `built-in(T)` in `iText-helper(N)` tests whether or not the type T is a built-in simple type.

Different from the XML data model, where a node has only a parent node, in XML Schema definitions, a node may have several instance parent nodes. Thus, the function `iPS`: `Sequence(Node)` → `Set(Sequence(Node))` for finding the instance preceding sibling nodes and the function `iFS`: `Sequence(Node)` → `Set(Sequence(Node))` for finding the instance following sibling nodes relate a sequence x of nodes to a set of sequences of nodes. The first node in x is the instance parent node of the last node of x. Let y be a node sequence in iPS(x), then y[1] = x[1], and y[|y|] is both an instance child node or an instance text node of y[1] and an instance preceding sibling node of x[|x|].

Since the XML Schema does not specify the position of the instance text nodes of a node N that defines an element e of complex type, we assume that a text child of the element e can appear before or after other children of the element e in any instance XML document. If y[|y|] = ⟨`complexType mixed = 'true'`...⟩ or y[|y|] = ⟨`complexContent mixed = 'true'`...⟩, then y[|y|] is an instance text node of y[1] that defines an element of complex type. Thus, a text child of the element can appear before or after other children of the element in any instance XML document. However, if N defines an element e of complex type, which has attributes and the text child but has no element children, then the text child is the only child of e. Thus, the instance text node ⟨`simpleContent`...⟩ of N has no instance sibling nodes. Similarly, the text child of an element of simple type is the only child of the element, so the instance text node of a node that defines an element of simple type has no instance sibling nodes. If y[|y|] = ⟨`simpleContent`...⟩ or y[|y|] = @⟨`type = T`⟩ or y[y] = ⟨`simpleType`...⟩, then y[|y|] is an instance text node of y[1], and thus y[|y|] has no instance preceding and following sibling nodes.

A node N2 = y[|y|] is an instance preceding sibling node of the instance node N1 = x[|x|], i.e. y is a node sequence in iPS(x), if N2 is an instance child node of N = x[1] in the case that N1 and N2 are contained in an *all* model group, or if N2 is an instance child node of N in the case that there is at least a model group, which either directly or recursively contains both N1 and N2, is declared with `maxOccurs` > 1, or if N2 is an instance child node of N, and N2 is visited before N1 in the XML Schema definition, in the case that all the model groups, which either directly or recursively contain both x and y, consist of only *sequence* and *choice* groups, which are declared with `maxOccurs` = 1. In the latter, N2 is not an instance sibling node of N1, if N1 and N2 are contained in a common *choice* group, and either N1 or N2 must be directly contained in the *choice* group.

x[|x|] and y[|y|] have some common ancestor nodes, some of which may be the model groups that either directly or recursively contain x[|x|] and y[|y|]. The common ancestor nodes are the nodes from x[1] to x[k] if ∀i ∈ {1, ..., k}: x[i] = y[i] ∧ x[k + 1] ≠ y[k + 1] ∧ k < min(|x|, |y|), where the function min(|x|, |y|) returns the minimum of |x| and |y|. Among these common ancestor nodes, x[1] is the instance parent node of x[|x|] and y[|y|], and thus the possible model group nodes in these common ancestor nodes are the nodes from x[2] to x[k]. If x[|x|] = y[|y|], then x = y. In this case, whether or not x[|x|] is a sibling node of itself relies on the occurrence constraints of x[|x|]. If x[|x|] can occur more than one time, i.e. ∃

$\mathtt{i} \in \{2, 3, \ldots, |\mathtt{x}|\}$: $\mathtt{attribute}(\mathtt{x}[\mathtt{i}], \text{'}\mathtt{maxOccurs}\text{'}) > 1$, then $\mathtt{x}[|\mathtt{x}|]$ is either a preceding sibling node or a following sibling node of itself.

XML Schema stipulates that an *all* group must appear as the sole child at the top of a content model, and the content model of an *all* group consists of element declarations, i.e. $\langle\mathtt{all}\ldots\rangle$ $\mathtt{elementD}*$ $\langle/\mathtt{all}\rangle$. Therefore, if $\mathtt{x}[\mathtt{k}] = \langle\mathtt{all}\rangle$, then $\mathtt{x}[|\mathtt{x}|]$ and $\mathtt{y}[|\mathtt{y}|]$ are contained in an *all* group, and thus the element declared in $\mathtt{x}[|\mathtt{x}|]$ can appear before or after the element declared in $\mathtt{y}[|\mathtt{y}|]$ in any valid XML document. If there is at least one node in $(\mathtt{x}[2], \ldots, \mathtt{x}[\mathtt{k}])$ defined with $\mathtt{maxOccurs} > 1$, i.e. $\exists \mathtt{i} \in \{2, 3, \ldots, \mathtt{k}\}$: $\mathtt{attribute}(\mathtt{x}[\mathtt{i}], \text{'}\mathtt{maxOccurs}\text{'}) > 1$, then the element declared in $\mathtt{x}[|\mathtt{x}|]$ can appear before or after the element declared in $\mathtt{y}[|\mathtt{y}|]$ in any valid XML document. If $(\mathtt{x}[2], \ldots, \mathtt{x}[\mathtt{k}])$ does not contain an *all* group and each model group in the sequence is defined with $\mathtt{maxOccurs} = 1$, then the element declared in $\mathtt{x}[|\mathtt{x}|]$ and the element declared in $\mathtt{y}[|\mathtt{y}|]$ appear in an XML instance document in the same order as the visited order of the node $\mathtt{x}[|\mathtt{x}|]$ and the node $\mathtt{y}[|\mathtt{y}|]$. The visited order is defined by the order in which $\mathtt{y}[\mathtt{k}+1]$ and $\mathtt{x}[\mathtt{k}+1]$ appear in the XML Schema definition. Let $\mathtt{N1}$ and $\mathtt{N2}$ be two nodes in an XML Schema definition, then $\mathtt{N1} \ll \mathtt{N2}$ indicates that $\mathtt{N1}$ appears before $\mathtt{N2}$ in the XML Schema definition. However, if $\mathtt{x}[\mathtt{k}]$ is the node $\langle\mathtt{choice}\ldots\rangle$, then the element defined in $\mathtt{x}[|\mathtt{x}|]$ and the element defined in $\mathtt{y}[|\mathtt{y}|]$ cannot appear simultaneously in any XML instance document. Therefore, $\mathtt{y}[|\mathtt{y}|]$ is not an instance sibling node of $\mathtt{x}[|\mathtt{x}|]$, and thus $\mathtt{y}$ is not a node sequence of $\mathtt{iPS(x)}$. The auxiliary function $\mathtt{attribute}(\mathtt{N}, \mathtt{attributeName})$ returns the value of the attribute $\mathtt{attributeName}$ in node $\mathtt{N}$, e.g. $\mathtt{attribute}(\mathtt{N}, \text{'}\mathtt{maxOccurs}\text{'})$ retrieves the value of the attribute with the name $\mathtt{maxOccurs}$ in node $\mathtt{N}$.

The function $\mathtt{NT}$: $\mathtt{Node} \times \mathtt{NodeTest} \to \mathtt{Boolean}$, which tests an instance XSchema node $\mathtt{N}$ against a node test of XPath, is defined as:

- $\mathtt{NT}(\mathtt{N}, *) = \mathtt{isiElement(N)} \vee \mathtt{isiAttribute(N)}$
- $\mathtt{NT}(\mathtt{N},\mathtt{label}) = (\mathtt{isiElement(N)} \wedge \mathtt{attribute}(\mathtt{N},\text{'}\mathtt{name}\text{'}) = \mathtt{label}) \vee (\mathtt{isiAttribute(N)} \wedge \mathtt{attribute}(\mathtt{N},\text{'}\mathtt{name}\text{'}) = \mathtt{label})$
- $\mathtt{NT}(\mathtt{N},\mathtt{text}()) = \mathtt{isiText(N)}$
- $\mathtt{NT}(\mathtt{N},\mathtt{node}()) = \mathtt{true}$

4. XPath–XSchema evaluator

A common XPath evaluator is typically constructed to evaluate XPath queries on XML instance documents. Our approach evaluates XPath queries on XML Schema definitions rather than on the instance documents of schemas in order to test the satisfiability of XPath with respect to schemas. Therefore, we name our XPath evaluator *XPath–XSchema* evaluator.

4.1. Schema paths

Instead of computing the node set of XML documents specified by an XPath query, our XPath–XSchema evaluator computes a set of schema paths to the possible resultant nodes, when the XPath query is evaluated by a common XPath evaluator on XML instance documents. If an XPath query cannot be evaluated completely, the schema paths for the XPath query are computed to an empty set of schema paths.

Definition 9 (*Schema paths*). A schema path the type of which we denote by $\mathtt{schema\_path}$ is a sequence of pointers to either the schema path records $\langle\mathtt{XP'}, \mathtt{S}, \mathtt{z}, \mathtt{lp}, \mathtt{f}\rangle$, or the schema path records $\langle\mathtt{o}, \mathtt{f}\rangle$, or schema path records $\langle\mathtt{e}\rangle$ where

- $\mathtt{XP'}$ is an XPath expression,
- $\mathtt{S}$ is a set of sequences of XSchema nodes,
- $\mathtt{z}$ is a set of pointers to schema path records,
- $\mathtt{lp}$ is a set of schema paths,
- $\mathtt{f}$ is a set of sets of schema paths,
- $\mathtt{e}$ is a predicate expression $\mathtt{self{::}node}() = \mathtt{C}$, where $\mathtt{C}$ is a literal, i.e. a number or a string, and
- $\mathtt{o}$ is a keyword and $\mathtt{o} \in \{=, \mathtt{or}, \mathtt{and}, \mathtt{not}\}$.

Let $\mathtt{Q}$ be an XPath query, which is the input of our XPath–XSchema evaluator, and $\mathtt{Q} = \mathtt{XP_e}/\mathtt{XP_c}/\mathtt{XP_r}$, where $\mathtt{XP_e}$ is the part, which has been evaluated, $\mathtt{XP_c}$ is the part, which is being evaluated, and $\mathtt{XP_r}$ is the part, which has not been evaluated so far by the XPath–XSchema evaluator. In a schema path record, $\mathtt{XP}' = \mathtt{XP_c}/\mathtt{XP_r}$. $\mathtt{XP}'$ is needed for the detection of loop schema paths. `S` is a set of sequences of XSchema nodes and the last node $\mathtt{N_l}$ in each sequence `s` of `S` is an instance node, which is visited by the XPath–XSchema evaluator when evaluating $\mathtt{XP_c}$, and which is also a context node to compute the following nodes. The first node $\mathtt{N_f}$ of `s` is an instance parent node of $\mathtt{N_l}$, and other nodes in `s` are ones that are visited when searching for $\mathtt{N_l}$ of $\mathtt{N_f}$, some of which may be the nodes of model groups and are useful for consistency checking of occurrence constraints and sequences. The field `z` in a schema record `R` is a set of pointers to the schema path records in which the last schema node of the node sequences is the instance parent node of the last schema node of the node sequences of the record `R`. Note whenever an instance XSchema node is the first node of a loop, the node has more than one possible instance parent node, and thus there are several sequences of nodes and pointers in a schema path record. `lp` represents loop schema paths; `f` represents the schema paths computed from the predicates that tests the last node of `S`, which is the context node of the predicates. The schema paths can consist of predicate expressions, i.e. $\{(\langle \mathtt{self::node()} = \mathtt{C}\rangle)\}$. `o` represents operators like =, `or`, `and` and `not`.

Example 2. Our XPath–XSchema evaluator evaluates the XPath query `Q` of Fig. 4 on the XML Schema definition of Fig. 1 and computes the schema paths presented in Fig. 5. Fig. 6 is the graphical representation of Fig. 5, in which we only present the last node of the node sequences in a schema path record rather than the entire record for simplicity of presentation and readability.

`Q` selects the parent node `refs` of the node `article`, which is a descendant node of the document node `bib`. The node `article` has two predicates. The first predicate qualifies that the node `article` must have children `year`. The second predicate qualifies that the node article cannot have children `editor`, or the node `article` may have children `editor`, but the children `editor` cannot have `bib` nodes as ancestor nodes.

Our XPath–XSchema evaluator first evaluates the very first part / of `Q`, and computes the first schema path record ⟨`Q`, {(/)}, -, -, -⟩. The first location step `bib` selects the instance child node `D13` of the document node `D1`. There are no other nodes visited after `D1` and before `D13`, such that the set of the node sequences is `{(D1, D13)}`. When evaluating //`artile`, the first selected instance child node of `D13` is `D4`, other nodes visited between `D13` and `D4` are `D14`, `D15`, `D2`, `D3` in this order. The instance child nodes of `D4` are `D7`, `D8`, `D9` and `D10`, and thus the following schema paths are computed:

$$\texttt{/}\overbrace{\texttt{bib}\underbrace{\texttt{//article[}\underbrace{\texttt{year}}_{\texttt{S3}}\texttt{ and not(self::node()[editor]/AoS::node()[self::node()='bib'])]/}\underbrace{\texttt{parent::refs}}_{\texttt{S4}}}_{\texttt{S2}}}^{\texttt{S1}}$$

Fig. 4. Example XPath query `Q` and its sub-expressions.

```
(R1)  { (<Q,  {(/)},   -,    -,   -> ,
(R2)     <S1, {(D1, D13)},  {R1},  -,  ->,
(R3)     <S2, {(D13, D14, D15, D2, D3, D4),(D10, D11, D12, D2, D3, D4)},  {R2, R4},
(R4)              { (<S2,  {(D4, D5, D6, D10)},  {R3}, -, -> ,
(R5)                 <S2, {(D10, D11, D12, D2, D3, D4)},  {R4}, -, ->) },
(R6)                            { {(<'and',
(R7)                                     { {(<-, {(D13, D14, D15, D2, D3, D4),
                                                (D10, D11, D12, D2, D3, D4)},  {R2, R4}, -, ->,
(R8)                                        <S3, {(D4, D5, D6, D8)},  {R7}, -, ->)},
(R9)                                      {(<'not',
(R10)                                     ∅>)} } >)} }>,
(R11)    <S4, {(D4, D5, D6, D10)},  {R3}, -, ->) }
```

Fig. 5. Schema paths of query `Q`.

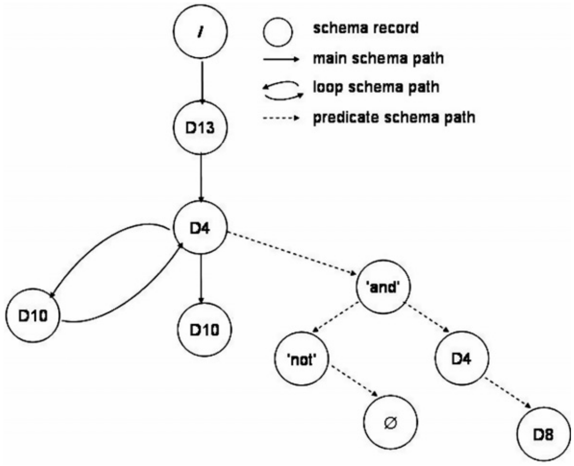

Fig. 6. Graphical representation of the schema paths of Fig. 5, where we only present the last node of the node sequences of the records of the schema paths.

{(R1, R2, R3, ⟨S2, {(D4, D5, D6, D7)}, {R3}, -, -⟩),
(R1, R2, R3, ⟨S2, {(D4, D5, D6, D8)}, {R3}, -, -⟩),
(R1, R2, R3, ⟨S2, {(D4, D5, D6, D9)}, {R3}, -, -⟩),
(R1, R2, R3, ⟨S2, {(D4, D5, D6, D10)}, {R3}, -, -⟩)}.

Since D7, D8 and D9 are not the resultant nodes of the location //artile and they do not have any descendant nodes either, the schema paths of these branches are computed to empty. The instance child node of D10 is D4 and the corresponding node sequence is D10, D11, D12, D2, D3, D4. The schema paths are now

(R1) {(⟨Q, {(/)}, -, -, -⟩,
(R2) ⟨S1, {(D1, D13)}, {R1}, -, -⟩,
(R3) ⟨S2, {(D13, D14, D15, D2, D3, D4)}, {R2}, -, -⟩
(R4) ⟨S2, {(D4, D5, D6, D10)}, {R3}, -, -⟩,
(R5) ⟨S2, {(D10, D11, D12, D2, D3, D4)}, {R4}, -, -⟩)}.

The resultant schema paths of //article are

(R1) {(⟨Q, {(/)}, -, -, -⟩,
(R2) ⟨S1, {(D1, D13)}, {R1}, -, -⟩,
(R3) ⟨S2, {(D13, D14, D15, D2, D3, D4)}, {R2}, -, -⟩
(R4) ⟨S2, {(D4, D5, D6, D10)}, {R3}, -, -⟩,
(R5) ⟨S2, {(D10, D11, D12, D2, D3, D4)}, {R4}, -, -⟩),
(R6) ⟨S2, {(D4, D5, D6, D10)}, {R5}, -, -⟩, ...)}

A loop occurs when evaluating //article, i.e. D10 is an instance child node of D4 and D4 is an instance child node of D10. When a loop is detected, the loop part is placed to the field of the loop schema paths in the record, where the last schema node of the node sequences is the initial node of the loop. Therefore, the schema paths are modified as follows:

(R1) {(⟨Q, {(/)}, -, -, -⟩,
(R2) ⟨S1, {(D1, D13)}, {R1}, -, -⟩,
(R3) ⟨S2, {(D13, D14, D15, D2, D3, D4), (D10, D11, D12, D2, D3, D4)}, {R2, R4},
(R4) {(⟨S2, {(D4, D5, D6, D10)}, {R3}, -, -⟩,
(R5) ⟨S2, {(D10, D11, D12, D2, D3, D4)}, {R4}, -, -⟩)}, -⟩)}

We present the detection of loops and the constructions of loop schema paths in Section 4.3.

The location step //`article` has a predicate that is and-combined with of two predicate expressions, so the schema paths of the predicate consists of the schema path record ⟨`o`,{`f1`,`f2`}⟩, where `o` = '`and`' and the two schema paths `f1` and `f2` are computed from two predicate expressions, respectively. The first record of the schema paths of the predicate expression `year` is the record, the last schema node of which is the context node of `year`, the purpose of the record is setting the context node of the predicate expression. Furthermore, since the nodes `article` selected by `self`::`node`() do not have a child `editor`, the schema paths of the predicate [`editor`] are computed to empty, and thus the evaluation of (`self`::`node`()[`editor`]/`AoS`::`node`()[`self`::`node`() = '`bib`']) is aborted after the evaluation of [`editor`]. Therefore, the schema paths of this part are computed to empty (see (R10) in Fig. 5). We present the method to evaluate predicates in Section 4.4.

4.2. Evaluating XPath expressions

We use the semantics technique to describe our XPath–XSchema evaluator, and define the following notations. Let `z` be a pointer in a schema path and `d` is a field of a schema path record, we write `z.d` to refer to the field `d` of the record to which the pointer `z` points. Let `p` be a schema path and |`p`| be the size of the schema path `p`, i.e. the number of pointers (or schema path records) in `p`, then `p[k]` indicates the *k*th pointer (or the record to which the *k*th pointer points) of the schema path `p`, and thus `p`[|`p`|].`XP`' refers to the field `XP`' of the last schema record of `p`. For readability, we often write that `p[k]` is the *k*th schema path record of schema path `p`, instead of that `p[k]` is the *k*th pointer of `p`, which points to a schema path record. Let `S` be a set of sequences of XSchema nodes, then `S(1)` indicates an arbitrary sequence of nodes in `S`. We use the operator / to express the concatenation of two XPath expressions, e.g. `XP1/XP2`.

The semantics of the XPath–XSchema evaluator is specified by a function `L` (see Fig. 7). The function `L`: `XPath x schema_path x XPath` → `Set(schema_path)` takes two XPath expressions and a schema path as the arguments and yields a set of new schema paths. The first XPath expression is one that is evaluated on a given XML Schema definition in this function, and the second XPath expression is the part `XP2` of the given XPath query `Q`, which has not been evaluated so far when the function is called. `XP2` is assigned to the `XP`' field of a schema path record, and this field is needed for the detection of loop schema paths. The schema path in this function signature is one of the schema paths of the part `XP1` of the given XPath query `Q`, which has been evaluated when calling this function. Thus, `Q` = `XP1/XP2`. `L(XPath, schema_path, XPath)` is defined recursively on the structure of XPath expressions (see Fig. 7).

4.3. Evaluating axis and node-test

For evaluating each location step of an XPath expression, our XPath–XSchema evaluator first computes the axis and the node-test `a::n` of the location step by iteratively taking the last schema node from a node sequence of the last schema path record (note that the last node of all the node sequences in a schema path record are the same) from each schema path `p` in the path set as the context node (see Fig. 8). The path set is computed from the part of the XPath query, which has been evaluated by the XPath–XSchema evaluator. For each resultant node `r` selected by `a::n`, `L` first computes a node sequence `s` based-on the data model of the XML Schema. `s[1]` is the instance parent node of `r`, `s`[|`s`|] = `r` and other nodes in `s` are intermediate ones visited when searching for `r` of `s[1]`. The function `L` then constructs a pointer `e` to a new schema path record, i.e. `e` → ⟨`xp`', {`s`}, `z`, -, -⟩ and extends `p` to `p`' by adding the pointer `e` at the end of the given schema path `p`, denoted by `p`' = `p` + `e`. In Example 2, the new schema path record `e` → ⟨`S4`, {(`D4`, `D5`, `D6`, `D10`)}, {`R3`}, -, -⟩ is generated when evaluating the part `parent::refs` of the query `Q`, and is added at the

- L(e1|e2, p, e1|e2) = L(e1, p, e1) ∪ L(e2, p, e2)
- L(/e, p, /e) = L(e, p1, e), where p1=(</e, {(/)}, -, -, - >)
- L(e1/e2, p, e1/e2) = { p2 | p2∈ L(e2, p1, e2) ∧ p1∈ L(e1, p, e1/e2) }

Fig. 7. The function `L`: `XPath x schema_path x XPath` → `Set(schema_path)` is defined recursively on the structure of XPath expressions.

- $L(\mathrm{self}::n, p, xp') = \{ p + \langle xp', S, p[|p|].z, \text{-}, \text{-} \rangle \mid S=p[|p|].S \wedge NT(S(1)[|S(1)|], n) \}$
- $$\begin{array}{l} L(\mathrm{child}::n, p, xp') = \{ p + \langle xp', \{s\}, p[|p|], \text{-}, \text{-} \rangle \mid NT(s[|s|], n) \wedge isiElement((S(1)[|S(1)|]) \wedge S=p[|p|].S \wedge (\\ \quad (s\in iChild(S(1)[|S(1)|]) \wedge n\neq \mathrm{text}()) \vee \\ \quad (s\in iTextChild(S(1)[|S(1)|] \wedge (n=\mathrm{text}() \vee n=\mathrm{node}()))) \} \end{array}$$
- $L^r(\mathrm{self}::n, p, xp') = \{ p \mid NT(S(1)[|S(1)|], n) \wedge S=p[|p|].S \}$
- $$\begin{array}{l} L(\mathrm{descendant}::n, p, xp') = \{ p' \mid p'\in \cup_{i=1}^{\infty} L^r(\mathrm{self}::n, p'_i, xp') \wedge (\\ \quad (p_i'=p_i \wedge p_i\in L(\mathrm{child}::\mathrm{node}(), p_{i-1}, xp') \wedge \\ \qquad \forall k\in\{1, \ldots, |p_i|-1\}: (\\ \qquad\qquad p_i[k].XP'\neq p_i[|p_i|].XP' \vee \\ \qquad\qquad (S_1(1)[|S_1(1)|]\neq S_2(1)[|S_2(1)|] \wedge S_1=p_i[k].S \wedge S_2=p_i[|p_i|].S)) \wedge \\ \qquad p_{i-1}\in L(\mathrm{child}::\mathrm{node}(), p_{i-2}, xp') \wedge \ldots \wedge p_1\in L(\mathrm{child}::\mathrm{node}(), p, xp')) \\ \quad \vee \\ \quad (p'_i=\perp \wedge (p_i[k]\rightarrow\langle p_i[k].XP', p_i[k].S\cup p_i[|p_i|].S, p_i[k].z\cup p_i[|p_i|].z, \\ \qquad\qquad p_i[k].lp\cup\{(p_i[k+1], \ldots, p_i[|p_i|-1], p_i[|p_i|])\}, p_i[k].f\rangle) \wedge \\ \qquad \exists k\in\{1, \ldots, |p_i|-1\}: (\\ \qquad\qquad p_i[k].XP'=p_i[|p_i|]XP' \wedge \\ \qquad\qquad S_1(1)[|S_1(1)|]=S_2(1)[|S_2(1)|] \wedge S_1=p_i[k].S \wedge S_2=p_i[|p_i|].S) \wedge \\ \qquad p_i\in L(\mathrm{child}::\mathrm{node}(), p_{i-1}, xp') \wedge p_{i-1}\in L(\mathrm{child}::\mathrm{node}(), p_{i-2}, xp') \wedge \\ \qquad\quad \ldots \wedge p_1\in L(\mathrm{child}::\mathrm{node}(), p, xp')))\} \end{array}$$
- $L(\mathrm{parent}::n, p, xp') = \{ p + \langle xp', S, Z1.z, \text{-}, \text{-} \rangle \mid S=Z1.S \wedge Z1\in p[|p|].z \wedge NT(S(1)[|S(1)|], n) \}$
- $$\begin{array}{l} L(\mathrm{ancestor}::n, p, xp') = \{ p' \mid p'\in \cup_{i=1}^{\infty} L^r(\mathrm{self}::n, p'_i, xp') \wedge (\\ \quad (p_i'=p_i \wedge p_i\in L(\mathrm{parent}::*, p_{i-1}, xp') \wedge \\ \qquad \forall k\in\{1, \ldots, |p_i|-1\}: (\\ \qquad\qquad p_i[k].XP'\neq p_i[|p_i|].XP' \vee \\ \qquad\qquad (S_1(1)[|S_1(1)|]\neq S_2(1)[|S_2(1)|] \wedge S_1=p_i[k].S \wedge S_2=p_i[|p_i|].S)) \wedge \\ \qquad p_{i-1}\in L(\mathrm{parent}::*, p_{i-2}, xp') \wedge \ldots \wedge p_1\in L(\mathrm{parent}::*, p, xp')) \\ \quad \vee \\ \quad (p'_i=\perp \wedge (p_i[k]\rightarrow\langle p_i[k].XP', p_i[k].S\cup p_i[|p_i|].S, p_i[k].z\cup p_i[|p_i|].z, \\ \qquad\qquad p_i[k].lp\cup\{(p_i[k+1], \ldots, p_i[|p_i|-1], p_i[|p_i|])\}, p_i[k].f\rangle) \wedge \\ \qquad \exists k\in\{1, \ldots, |p_i|-1\}: (\\ \qquad\qquad p_i[k].XP'=p_i[|p_i|]XP' \wedge \\ \qquad\qquad S_1(1)[|S_1(1)|]=S_2(1)[|S_2(1)|] \wedge S_1=p_i[k].S \wedge S_2=p_i[|p_i|].S) \wedge \\ \qquad p_i\in L(\mathrm{parent}::*, p_{i-1}, xp') \wedge p_{i-1}\in L(\mathrm{parent}::*, p_{i-2}, xp') \wedge \\ \qquad\quad \ldots \wedge p_1\in L(\mathrm{parent}::*, p, xp')))\} \end{array}$$
- $L(\mathrm{DoS}::n, p, xp') = L(\mathrm{self}::n, p, xp') \cup L(\mathrm{descendant}::n, p, xp')$
- $L(\mathrm{AoS}::n, p, xp') = L(\mathrm{self}::n, p, xp') \cup L(\mathrm{ancestor}::n, p, xp')$
- $L(\mathrm{FS}::n, p, xp') = \{ p + \langle xp', \{s\}, p[|p|].z, \text{-}, \text{-} \rangle \mid s\in iFS(s1) \wedge NT(s[|s|], n) \wedge s1\in p[|p|].S \}$
- $L(\mathrm{following}::n, p, xp') = L(\mathrm{AoS}::\mathrm{node}()/\mathrm{FS}::\mathrm{node}()/\mathrm{DoS}::n, p, xp')$
- $L(\mathrm{PS}::n, p, xp') = \{ p + \langle xp', \{s\}, p[|p|].z, \text{-}, \text{-} \rangle \mid s\in iPS(s1) \wedge NT(s[|s|], n) \wedge s1\in p[|p|].S \}$
- $L(\mathrm{preceding}::n, p, xp') = L(\mathrm{AoS}::\mathrm{node}()/\mathrm{PS}::\mathrm{node}()/\mathrm{DoS}::n, p, xp')$
- $L(\mathrm{attribute}::n, p, xp') = \{ p + \langle xp', \{s\}, p[|p|], \text{-}, \text{-} \rangle \mid s\in iAttribute(S(1)[|S(1)|]) \wedge NT(s[|s|], n) \wedge S=p[|p|].S \}$

Fig. 8. The function `L`: `XPath x schema_path x XPath` → `Set(schema_path)` for evaluating axis and node test.

end of `p` (see (R11) in Fig. 5) by `L(parent::refs, p, parent::refs)`. If no node is selected by the current location step, the function `L` computes an empty set of schema paths. For example, the part `[editor]` of `Q` in Example 2 is computed to empty by `L(editor, p, editor)` since no node is selected by the current location step `editor` and this causes that the corresponding main schema paths are computed to empty (see (R10) in Fig. 5).

In the case of recursive schemas, a loop is identified whenever the XPath–XSchema evaluator revisits a node `N` of the XML Schema definition without any progress in the processing of the query. In order to avoid an infinite evaluation, we do not continue the evaluation after the node `N`, once a loop has been detected. We detect loops in the following way: let $e = \langle xp', \{s\}, z, \text{-}, \text{-} \rangle$ be a new schema path record generated when computing

$L(a::n, p, xp')$. If there exists a record $p[k]$ in p such that $S(1)[|S(1)|] = s[|s|] \wedge S = p[k].S \wedge p[k].XP' = xp'$, a loop is detected and the loop path segment is $lp = (p[k+1], \ldots, p[|p|], e)$. lp is added to the field of the loop schema paths in the schema path record $p[k]$, where the loop occurs (e.g. (R4) and (R5) in Fig. 5). A loop might occur when an XPath query contains the axis descendant, ancestor, preceding or following, which are boiled down to the recursive evaluation of the axis child or parent, respectively. For computing $L(descendant::n, p, xp')$, we first compute p_i, where $p_i \in L(child::node(), p_{i-1}, xp') \wedge p_{i-1} \in L(child::node(), p_{i-2}, xp') \wedge \ldots \wedge p_1 = L(child::node(), p, xp')$. If no loop is detected in the path p_i, i.e. $\forall k \in \{1, \ldots, |p_i| - 1\}: p_i[k].XP' \neq p_i[|p_i|].XP' \vee (S_1(1)[|S_1(1)|] \neq S_2(1)[|S_2(1)|] \wedge S_1 = p_i[k].S \wedge S_2 = p_i[|p_i|].S)$, then let $p'_i = p_i$ and $L^r(self::n, p'_i, xp')$ is computed in order to construct a possible new path from p_i. If a loop path segment $(p_i[k+1], \ldots, p_i[|p_i| - 1], p_i[|p_i|])$ is detected in the path p_i, i.e. $\exists k \in \{1, \ldots, |p_i| - 1\}$: $p_i[k].XP' = p_i[|p_i|].XP' \wedge S_1(1)[|S_1(1)|] = S_2(1)[|S_2(1)|] \wedge S_1 = p_i[k].S \wedge S_2 = p_i[|p_i|].S$, then the schema path record $p_i[k]$, from which the loop starts, is modified by integrating the new detected loop schema path, the new sequence of nodes and the new parent pointer, i.e. $\langle p_i[k].XP', p_i[k].S \cup p_i[|p_i|].S, p_i[k].z \cup p_i[|p_i|].z, p_i[k].lp \cup \{(p_i[k+1], \ldots, p_i[|p_i| - 1], p_i[|p_i|])\}, p_i[k].f\rangle$. Note that all the schema paths, which contain the pointer to the schema path record, are also aware of this modification. When a loop is detected, instead of setting $p'_i = p_i$, p'_i is set to empty, i.e. if a loop is detected in p_i, p_i will not contribute to the further computation of schema paths anymore.

4.4. Evaluating predicates

The schema paths $L(q, fp, q)$ of a predicate q are added into the field of the predicate schema paths in the record, where the last node of the field of the node sequences is the context node of the predicate, e.g. $L(e[q], p, xp') = \{(p'[1], p'[2], \ldots, p'[|p'| - 1]) + \langle p'[|p'|].XP', p'[|p'|].S, p'[|p'|].z, p'[|p'|].lp, p'[|p'|].f \cup L(q, fp, q)\rangle | p' \in L(e, p, xp') \wedge L(q, fp, q) \neq \emptyset \wedge fp = (\langle -, p'[|p'|].S, p'[|p'|].z, -, - \rangle)\}$ (see Fig. 9). fp logs the context node of the predicate such that we compute the schema paths of the predicate from fp. When $L(q, fp, q)$ is computed to empty, the main schema paths are computed to an empty set of schema paths, i.e. $L(e[q], p, xp') = \emptyset$ if $L(q, fp, q) = \emptyset$. When $q = (q_1 \text{ or } q_2)$, $L(q_1 \text{ or } q_2, fp, q_1 \text{ or } q_2)$ computes a schema path with only one record for the predicate expression q_1 or q_2, i.e. $\{(\langle \text{'or'}, L(q_1, fp, q_1) \cup L(q_2, fp, q_2)\rangle)\}$ that consists of a keyword or and two sets of schema paths computed from q_1 and q_2 respectively. The schema path is added to the field of predicate schema paths of the record, where the last node in the field of the node sequences is the context node of $[q_1 \text{ or } q_2]$. If both $L(q_1, fp, q_1)$ and $L(q_2, fp, q_2)$ are computed to empty, the schema paths of the predicate q_1 or q_2 are computed to the empty set, i.e. $L(q_1 \text{ or } q_2, fp, q_1 \text{ or } q_2) = \emptyset$ if $L(q_1, fp, q_1) = \emptyset \wedge L(q_2, fp, q_2) = \emptyset$.

4.5. Integrating data type checking

The XML Schema language defines 44 built in simple types, and allows users to define new simple types. If the value of an element or an attribute in an XPath query does not conform to the type of the value of the element or the attribute specified in the given XML Schema definition, the XPath query selects an empty

- $L(e[q], p, xp') = \{ (p'[1], p'[2], \ldots, p'[|p'|-1]) + \langle p'[|p'|].XP', p'[|p'|].S, p'[|p'|].z, p'[|p'|].lp,$ $p'[|p'|].f \cup L(q, fp, q)\rangle \mid p' \in L(e, p, xp') \wedge L(q, fp, q) \neq \emptyset \wedge fp = (\langle -, p'[|p'|].S, p'[|p'|].z, -, -\rangle) \}$
- $L(e[q_1]\ldots[q_n], p, xp') = \{ (p'[1], p'[2], \ldots, p'[|p'|-1]) +$ $\langle p'[|p'|].XP', p'[|p'|].S, p'[|p'|].z, p'[|p'|].lp, p'[|p'|].f \cup L(q_1, fp, q_1) \cup \ldots \cup L(q_n, fp, q_n)\rangle \mid$ $p' = L(e, p, xp') \wedge L(q_1, fp, q_1) \neq \emptyset \wedge \ldots \wedge L(q_n, fp, q_n) \neq \emptyset \wedge fp = (\langle -, p'[|p'|].S, p'[|p'|].z, -, -\rangle) \}$
- $L(q_1 \text{ and } q_2, fp, q_1 \text{ and } q_2) = \{ (\langle \text{'and'}, L(q_1, fp, q_1) \cup L(q_2, fp, q_2)\rangle) \mid L(q_1, fp, q_1) \neq \emptyset \wedge L(q_2, fp, q_2) \neq \emptyset \}$
- $L(q_1 \text{ or } q_2, fp, q_1 \text{ or } q_2) = \{ (\langle \text{'or'}, L(q_1, fp, q_1) \cup L(q_2, fp, q_2)\rangle) \mid L(q_1, fp, q_1) \neq \emptyset \vee L(q_2, fp, q_2) \neq \emptyset \}$
- $L(q_1 = q_2, fp, q_1 = q_2) = \{ (\langle \text{'='}, L(q_1, fp, q_1) \cup L(q_2, fp, q_2)\rangle) \mid L(q_1, fp, q_1) \neq \emptyset \wedge L(q_2, fp, q_2) \neq \emptyset \}$
- $L(not(q), fp, not(q)) = \{ (\langle \text{'not'}, L(q, fp, q)\rangle) \}$
- $L(q{=}C, fp, q{=}C) = L(q[self::node(){=}C], fp, q[self::node(){=}C])$, where $q \neq self::node()$
- $L(self::node(){=}C, fp, self::node(){=}C) = \{ (\langle self::node(){=}C\rangle) \}$

Fig. 9. The function L: XPath x schema_path x XPath → Set(schema_path) for evaluating predicates.

set of nodes for any XML document, which is valid according to the given XML Schema definition. Therefore, integration of data type checking, when evaluating XPath queries on an XML Schema definition, can detect more unsatisfiable queries.

The data type checking is involved in the computation of the schema paths of the predicate expression `self::node() = C`, and thus we modify the function `L(self::node() = C, p, self::node() = C)` in order to integrate type-checking (see Fig. 10). In the XPath language, the value of an element is the text node of the element, and thus, e.g. two predicate expressions `child::mark = 1.0` and `child::mark/child::text() = 1.0` are semantically equal. Therefore, if the node selected by `self::node()` is an element node, we evaluate `child::text()/self::node() = C` rather than `self::node() = C` in order to make the node selected by `self::node()` be a text node. If the constant `C` of the predicate expression `self::node() = C` conforms to the type of the value of the node specified by `self::node()`, the predicate expression itself as the schema paths is added to the field of the predicate schema paths of the record, the last node of the node sequences of which is the context node of the predicate expression. If `C` does not conform to the type constraints, the predicate expression `self::node() = C` is computed to the empty set of schema paths, i.e. `L(self::node() = C, p, self::node() = C)` $= \emptyset$, and thus the corresponding main schema paths are computed to the empty set of schema paths. The auxiliary function `typeChecking(type, C)` validates whether or not the constant `C` conforms to the given type; the auxiliary function `valueType(N)` returns the type of the value of the element or the attribute declared in the node `N` and the restricting facets of the value.

Whenever an element contains elements and text nodes for its value, i.e. declared as ⟨`complexType mixed = 'true'...`⟩ or ⟨`complexContent mixed = 'true'...`⟩, XML Schema does not impose any specific data type for the value of the element. Therefore, the value is considered as character string, and there is no restricting facet either, i.e. we do not check the data type of values in this case.

XML Schema specifies a specific data-type for the value of elements if the elements are of a simple type, i.e. the elements consist of a text node with or without attributes. The attributes are always of simple types. The types of values of elements and attributes can be either the built-in simple types of the XML Schema language

```
• L(self::node()=C, p, self::node()=C) = { p1 | (
      ( p1∈L(child::text()/self::node()=C, p, child::text()/self::node()=C) ∧
        ¬isiText(N) ∧ ¬isiAttribute(N) ∧ N=S(1)[|S(1)|] ∧ S=p[|p|].S )
      ∨
      ( p1=(<self::node()=C>) ∧ (isiText(N) ∨ isiAttribute(N)) ∧
        N=S(1)[|S(1)|] ∧ S=p[|p|].S ∧ typeChecking(valueType(N), C) ∧ (
            ( valueType(N)=(T, -) ∧ (
                    N=@<type=T> ∨
                    ( N=<attribute type=T...> ∧ built-in(T) )))
            ∨
            ( valueType(N)=computeType(N1, facets) ∧ (
                    ( N=<attribute type=T...> ∧ ¬built-in(T) ∧
                      N1∈succeeding(N) ∧ N1=<simpleType name=T...> )
                    ∨
                    ( N=<attribute...> ∧ attributeNode(N, type=T)=⊥ ∧
                      N1∈child(N) ∧ N1=<simpleType...> )) ∧
              |facets|=12 ∧ facets[1]=null ∧ ... ∧ facets[12]=null )
            ∨
            ( valueType(N)=('string', -) ∧ (
                    N=<complexType mixed= 'true'...> ∨
                    N=<complexContent mixed= 'true'...> ))
            ∨
            ( valueType(N)=computeType(N, facets) ∧ (
                    N=<simpleType...> ∨ N=<simpleContent...> ) ∧
              |facets|=12 ∧ facets[1]=null ∧ ... ∧ facets[12]=null ))))}
```

Fig. 10. The function `L: XPath x schema_path x XPath → Set(schema_path)` for integrating data type checking.

or user-defined simple types. New simple types are derived from existing simple types, which are called the base types of the derived types, by restricting the range of base types. XML Schema applies one or more *facets* to restrict the legal values of base types. Thus, a new simple type is a particular combination of a base type and the facets. Base types can be built-in or derived, and thus in order to know what a new simple type is, one must find the source of the derivation, i.e. the built-in simple type, and all the restrictions imposed by the sequence of the derivations. The function `computeType(N, facets)` computes the type of value of an element or an attribute, if the element or the attribute is not of a built-in simple type, where `N` is the instance text node of the node that declares the element or `N` is the succeeding node ⟨`simpleType`...⟩ of the instance attribute node that declares the attribute.

Whenever an instance text node is the attribute node `type` = `T` of an element declaration node `N`, then `T` must be a built-in simple type. In this case, the value type of the element defined in `N` is `T` without restricting facets, i.e. `valueType(N)` = (`T`, -). Since an attribute is always of simple type, the attribute node can be declared with a built-in simple type, or with a user-defined simple type, or with an anonymously new simple type. Therefore, if `N` = ⟨`attribute type` = `T`...⟩ and `built-in(T)`, i.e. `T` is a built-in simple type, the value type of the attribute is the built-in simple type without restricting facets, i.e. `valueType(N)` = (`T`, -). If `N` = ⟨`attribute type` = `T`...⟩ and ¬`built-in(T)`, then `T` is defined by a node `N1` = ⟨`simpleType name` = `T`...⟩ that is a succeeding node of `N`, and the value type of the attribute defined in `N` is computed by the function `computeType(N1, facets)`. If an instance attribute node `N` does not contain a named type, i.e. `attributeNode(N, type = T)` = ⊥, the instance attribute node has an anonymous type that is defined in a child node `N1` = ⟨`simpleType`...⟩ of `N`, the value type of the attribute declared in the node `N` is computed by the function `computeType(N1, facets)`.

Algorithm 1 (`computeType(N, facets)`) describes how to retrieve the type of values of attributes and elements according to the syntax for `simpleTypeD` and `simpleContentD` (see Section 2.2). XML Schema identifies 12 restricting facets, and thus the argument `facets` is an array variable containing 12 string data. We use the name of facets specified in [32] as the index of the array to which the value of the facet is assigned.

Algorithm 1 (`computeType(N, facets)`)

```
N1 ∈child(N);
if (N1 = ⟨extension...⟩) {
   base = attribute(N1, 'base');
   if (built-in(base)) return (base, facets);
   else {
      N2 ∈ succeeding(N1), where N2 = ⟨simpleType...⟩;
      return computeType(N2, facets);
   }
}
if (N1 = ⟨restriction...⟩) {
   base = attribute(N1, 'base');
   if (∃s ∈ succeeding(N1): s = ⟨simpleType...⟩) (base, facets) = computeType(s, facets);
   ∀s ∈ succeeding(N1) {
      if (s = ⟨length value = V/⟩) facets[length] = V;
      if (s = ⟨minLength value = V/⟩) facets[minLength] = V;
      if (s = ⟨maxLength value = V/⟩) facets[maxLength] = V;
      if (s = ⟨pattern value = V/⟩) facets[pattern] = V;
      if (s = ⟨enumeration value = V/⟨) facets[enumeration] = V;
      if (s = ⟨whiteSpace value = V/⟩) facets[whiteSpace] = V;
      if (s = ⟨maxInclusive value = V/⟩) facets[maxInclusive] = V;
      if (s = ⟨maxExclusive value = V/⟩) facets[maxExclusive] = V;
      if (s = ⟨minInclusive value = V/⟩) facets[minInclusive] = V;
      if (s = ⟨minExclusive value = V/⟩) facets[minExclusive] = V;
      if (s = ⟨totalDigits value = V/⟩) facets[totalDigits] = V;
      if (s = ⟨fractionDigits value = V/⟩) facets[fractionDigits] = V;
   }
   return (base, facets);
}
```

In Algorithm 1, node N1 = ⟨extension base = *QName*⟩ is a child node of ⟨simpleContent...⟩; node N2 = ⟨restriction base = *QName*⟩ is a child node of ⟨simpleType...⟩. Both nodes indicate the base type of the derivation, which may be either a built-in or a derived simple type. If the base type is not a built-in simple type, there is a node ⟨simpleType name = *QName*⟩ with the same *QName*, which defines the base type of the derived type, and which is a succeeding node of N1 or N2. Thus, a new simple type might be derived recursively from a sequence of existing simple types, until the base is a built-in simple type. The facets that restrict the range of value of the base type are identified by several child nodes of ⟨restriction...⟩. Furthermore, the restrictions imposed by a derived type override the restrictions from its base type. If ⟨restriction...⟩ does not have a child ⟨simpleType...⟩, the attribute base of the node ⟨restriction...⟩ must be a built-in simple type. This means that we find the source of derivation and all restricting facets, i.e. we compute the type of value of the element or the attribute.

4.6. Integrating occurrence constraints checking

XML Schema specifies some constraints that control the occurrence of elements and attributes and their values. When an element is declared with maxOccurs = 0 (and minOccurs = 0, because it is an error if minOccurs ≠ 0) or a model group of the element is declared with maxOccurs = 0, or when an attribute is declared with use = 'prohibited', the element and the attribute must not appear in any instance document. When an element or an attribute is declared to have a fixed value, e.g. fixed = '100', the value of the element or the attribute in all instance documents must be 100.

In order to integrate the occurrence constraints checking, we modify the data model of XML Schema, especially, the functions iChild(x) and iAttribute(x) in Fig. 3, as in Fig. 11.

The function iChild(N) first computes a set S of node sequences using the auxiliary function iChild-helper(N). Each sequence y ∈ S consists of N and the nodes visited after N but before an instance child node of N. If the succeeding nodes N1 of y[|y|] are not the instance element nodes, then no node sequence is computed from y. In the case of a succeeding node N1 of y[|y|] being an instance element node, iChild(N) returns the node sequence y + (N1), only when each model group of N1 is declared with maxOccurs > 0, i.e. if u is a node in y, then u is either a node of a model group with maxOccurs > 0, or is a node rather than the node of a model group. If y[|y|] = ⟨element ref = E maxOccurs = D⟩, then N1 is an instance child node of N only when D > 0. Note that we do not check the attribute maxOccurs of the instance parent node y[1] of N1, because we assume that the elements defined in instance ancestor nodes of N1 are allowed to appear in instance XML documents.

The constraints on fixed values are closely related with type-checking, and thus the function L(self::node() = C, p, self::node() = C) is modified as in Fig. 12.

In L(self::node() = C, p, self::node() = C), if N = ⟨attribute...⟩ is the node selected by self::node(), N can carry the attribute *fixed*. If N contains the attribute fixed, i.e. N = ⟨attribute Ûxed = V...⟩,

- iChild(N) = { y+(N1) | y∈iChild-helper(N) ∧ N1∈succeeding(y[|y|]) ∧ isiElement(N1) ∧
  ∀i∈{2, 3, ..., |y|}: (
  ((y[i]=<group maxOccurs=D...> ∨
  y[i]=<sequence maxOccurs=D...> ∨
  y[i]=<choice maxOccurs=D...> ∨
  y[i]=<all maxOccurs=D....>) ∧
  D>0)
  ∨
  (y[i]≠<group...> ∧ y[i]≠<sequence...> ∧ y[i]≠<choice...> ∧ y[i]≠<all...>)) ∧
  ((y[|y|]=<element ref=E maxOccurs=D...> ∧ D>0) ∨ y[|y|]≠<element ref=E...>) ∧
  attribute(N1, maxOccurs)>0 }

- iAttribute(N) = { y+(N1) | y∈iChild-helper(N) ∧ N1∈succeeding(y[|y|]) ∧ isiAttribute(N1) ∧ (
  (y[|y|]=<attribute ref=A> ∧ attribute(y[|y|], 'use')≠'prohibited') ∨
  (y[|y|]≠<attribute ref=A> ∧ attribute(N1, 'use')≠'prohibited'))}

Fig. 11. The function L: XPath x schema_path x XPath → Set(schema_path) for integrating occurrence constraints checking.

```
• L(self::node()=C, p, self::node()=C) = { p1 | (
      ( p1∈L(child::text()/self::node()=C, p, child::text()/self::node()=C) ∧
        ¬isiText(N) ∧ ¬isiAttribute(N) ∧ N=S(1)[|S(1)|] ∧ S=p[|p|].S )
      ∨
      ( p1=(<self::node()=C>) ∧ (isiText(N) ∨ isiAttribute(N)) ∧
        N=S(1)[|S(1)|] ∧ S=p[|p|].S ∧ (
              ( C=V ∧ N=<attribute fixed=V…> )
              ∨
              ( C=V ∧ N1=<element fixed=V…> ∧ N1=s[1] ∧ s∈p[|p|].S ∧ (
                      N=@<type=T> ∨
                      N=<simpleType…> ∨
                      N=<simpleContent…> ∨
                      N=<complexType mixed= 'true'…> ∨
                      N=<complexContent mixed= 'true'…> ))))
      ∨
      ( p1=(<self::node()=C>) ∧ (isiText(N) ∨ isiAttribute(N)) ∧
        N=S(1)[|S(1)|] ∧ S=p[|p|].S ∧ typeChecking(valueType(N), C) ∧ (
              ( valueType(N)=(T, -) ∧ N=@<type=T> ∧
                attributeNode(N1, fixed=V)=⊥ ∧ N1=s[1] ∧ s∈p[|p|].S )
              ∨
              ( valueType(N)=(T, -) ∧ N=<attribute type=T…> ∧ built-in(T) ∧
                attributeNode(N, fixed=V)=⊥ )
              ∨
              ( valueType(N)=computeType(N1, facets) ∧
                attributeNode(N, fixed=V)=⊥ ∧ (
                      ( N=<attribute type=T…> ∧ ¬built-in(T) ∧ N1∈succeeding(N) ∧
                        N1=<simpleType name=T...> )
                      ∨
                      ( N=<attribute…> ∧ attributeNode(N, type=T)=⊥ ∧
                        N1=child(N) ∧ N1=<simpleType…> )) ∧
                |facets|=12 ∧ facets[1]=null ∧ … ∧ facets[12]=null )
              ∨
              ( valueType(N)=('string', -) ∧ attributeNode(N1, fixed=V)=⊥ ∧
                N1=s[1] ∧ s∈p[|p|].S   ∧ (
                      N=<complexType mixed= 'true'…>  ∨
                      N=<complexContent mixed= 'true'…> ))
              ∨
              ( valueType(N)=computeType(N, facets) ∧
                attributeNode(N1, fixed=V)=⊥ ∧ N1=s[1] ∧ s∈p[|p|].S ∧ (
                      N=<simpleType…> ∨ N=<simplexContent…> ) ∧
                |facets|=12 ∧ facets[1]=null ∧ … ∧ facets[12]=null )))))}
```

Fig. 12. The function L: XPath x schema_path x XPath → Set(schema_path) for integrating fixed value checking.

the schema paths of the predicate self::node() = C is $\{(\langle$self::node() = C$\rangle)\}$ if and only if C=V; the schema paths of the predicate self::node() = C is computed to the empty set if C $\neq$ V, and thus the corresponding main paths are computed to the empty set. If N does not contain the attribute *fixed*, i.e. attributeNode(N, fixed = V) = $\perp$, C must conform to the type of value of the attribute defined in N, in order to compute $\{(\langle$self::node( )=C$\rangle)\}$ from the predicate self::node() = C; if C does not conform to the type constraint, then L(self::node() = C, p, self::node() = C) = $\emptyset$. When the node selected by self::node() is an attribute node @$\langle$type = T$\rangle$ or a node $\langle$simpleType…$\rangle$ or a node $\langle$simpleContent…$\rangle$ or a node $\langle$complexType…$\rangle$ or a node $\langle$complexContent…$\rangle$, these nodes do not contain the attribute *fixed*, which can be contained by the instance parent node of these nodes, i.e. N1 = $\langle$element…$\rangle$, which is the first node in the corresponding node sequences.

4.7. Complexity analysis

We first analyze the complexity of our approach in the worst case. Different from instance XML documents the topology of which is a tree, an XML Schema definition is a directed graph. In the directed graph leading to the worst-case complexity, each node has directed edges to all nodes. Therefore, we assume that in an XML Schema definition $\mathtt{S}$ in the worst case, each node in $\mathtt{S}$ is an instance node and each node is a succeeding node of all the nodes. In an XPath query $\mathtt{Q}$ in the worst case, each location step in $\mathtt{Q}$ selects all the instance nodes in $\mathtt{S}$.

Let $\mathtt{a}$ be the number of location steps in the query $\mathtt{Q}$. Let $\mathtt{N}$ be the number of nodes and $\mathtt{i}$ be the number of the instance nodes in a given XML Schema definition $\mathtt{S}$, where $\mathtt{i} \leqslant \mathtt{N}$. In the worst case, from each schema path $\mathtt{p}$, the length of which is $\mathtt{s}$, of the result of the previous location step, firstly $\mathtt{N}$ nodes, which are directly reachable from the context node, are visited and selected as the resultant nodes, and thus $\mathtt{N}$ new schema path records are created and $\mathtt{N}$ schema paths with the length $\mathtt{s}+1$ are computed. From each of $\mathtt{N}$ visited nodes, $\mathtt{N}$ succeeding nodes are visited and selected as the resultant nodes, one of which is revisited. No new schema paths are computed from the revisited nodes, and they do not contribute to the further computation of schema paths either, but the revisited nodes indicate the occurrence of a loop. Therefore, $\mathtt{N}-1$ new schema path records are created, one existing schema path record is modified by integrating the new loop schema path, and $\mathtt{N}-1$ new schema paths with length of $\mathtt{s}+2$ are computed. Therefore, there are $\mathtt{N}+\mathtt{N}*\mathtt{N}$ nodes visited and $\mathtt{N}+\mathtt{N}*(\mathtt{N}-1)$ schema paths with length from $\mathtt{s}+1$ to $\mathtt{s}+2$ computed so far. From each of $\mathtt{N}*(\mathtt{N}-1)$ nodes, $\mathtt{N}$ succeeding nodes are visited and selected as the resultant nodes, two of which are revisited. Therefore, $\mathtt{N}-2$ new schema path records are created, two existing schema path records are modified by integrating the new loop schema path, and $\mathtt{N}-2$ new schema paths with length $\mathtt{s}+3$ are computed. $\mathtt{N}+\mathtt{N}*\mathtt{N}+\mathtt{N}*(\mathtt{N}-1)*\mathtt{N}$ nodes are visited and $\mathtt{N}+\mathtt{N}*(\mathtt{N}-1)+\mathtt{N}*(\mathtt{N}-1)*(\mathtt{N}-2)$ schema paths with length from $\mathtt{s}+1$ to $\mathtt{s}+3$ are computed so far. After a location step is evaluated, from each schema path $\mathtt{p}$ of the result of the previous location step, $\mathtt{N}+\mathtt{N}*\mathtt{N}+\mathtt{N}*(\mathtt{N}-1)*\mathtt{N}+\cdots+\mathtt{N}*(\mathtt{N}-1)*(\mathtt{N}-2)*\ldots*2*\mathtt{N}=\mathtt{N}*\sum_{\mathtt{k}=0}^{\mathtt{N}-1}\mathtt{N}!/(\mathtt{N}-\mathtt{k})!$ nodes are visited and $\mathtt{N}+\mathtt{N}*(\mathtt{N}-1)+\mathtt{N}*(\mathtt{N}-1)*(\mathtt{N}-2)+\cdots+\mathtt{N}*(\mathtt{N}-1)*(\mathtt{N}-2)*\cdots*2*1=\sum_{\mathtt{k}=1}^{\mathtt{N}}\mathtt{N}!/(\mathtt{N}-\mathtt{k})!$ schema paths are computed with length from $\mathtt{s}+1$ to $\mathtt{s}+\mathtt{N}$.

Let $\mathtt{X}=\mathtt{N}*\sum_{\mathtt{k}=0}^{\mathtt{N}-1}\mathtt{N}!/(\mathtt{N}-\mathtt{k})!$ and $\mathtt{P}=\sum_{\mathtt{k}=1}^{\mathtt{N}}\mathtt{N}!/(\mathtt{N}-\mathtt{k})!$. In the worst case, having evaluated the first location step, $\mathtt{X}$ nodes are visited and $\mathtt{P}$ schema paths are created with length from 1 to $\mathtt{N}$; having evaluated the first two location steps, $\mathtt{X}+\mathtt{P}*\mathtt{X}$ nodes are visited and $\mathtt{P}^2$ schema paths are created with length from 2 to $\mathtt{N}+\mathtt{N}$; having evaluated $\mathtt{Q}$, $\mathtt{X}+\mathtt{P}*\mathtt{X}+\mathtt{P}^2*\mathtt{X}+\cdots+\mathtt{P}^{\mathtt{a}-1}*\mathtt{X}=\mathtt{X}*\sum_{\mathtt{j}=0}^{\mathtt{a}-1}\mathtt{P}^{\mathtt{j}}$ nodes are visited and $\mathtt{P}^{\mathtt{a}}$ schema paths are created with length from $\mathtt{a}$ to $\mathtt{a}*\mathtt{N}$. Since $\Sigma_{\mathtt{k}=1}^{\mathtt{N}}(\mathtt{N}!/(\mathtt{N}-\mathtt{k})!)<\mathtt{N}!*3$ and $\Sigma_{\mathtt{k}=0}^{\mathtt{N}-1}(\mathtt{N}!/(\mathtt{N}-\mathtt{k})!)<\mathtt{N}!*2$, thus $\mathtt{X}*\sum_{\mathtt{j}=0}^{\mathtt{a}-1}\mathtt{P}^{\mathtt{j}}=\mathtt{N}*\sum_{\mathtt{k}=0}^{\mathtt{N}-1}\mathtt{N}!/(\mathtt{N}-\mathtt{k})!*\sum_{\mathtt{j}=}0^{\mathtt{a}-1}(\sum_{\mathtt{k}=1}^{\mathtt{N}}\mathtt{N}!/(\mathtt{N}-\mathtt{k})!)^{\mathtt{j}}<\mathtt{N}*\mathtt{N}!*2*\sum_{\mathtt{j}=0}^{\mathtt{a}-1}(\mathtt{N}!*3)^{\mathtt{j}}<\mathtt{N}*\mathtt{N}!*2*\mathtt{a}*(\mathtt{N}!*3)^{\mathtt{a}-1}$. Therefore, the XPath–XSchema evaluator visits at most $\mathtt{O}(\mathtt{N}*\mathtt{N}!*\mathtt{a}*(\mathtt{N}!*3)^{\mathtt{a}-1})$ nodes, and creates at most $\mathtt{O}((\mathtt{N}!*3)^{\mathtt{a}})$ different schema paths, each of which contains at most $\mathtt{O}(\mathtt{a}*\mathtt{N})$ pointers, and thus $\mathtt{O}((\mathtt{N}!*3)^{\mathtt{a}})$ schema paths contains at most $\mathtt{O}(\mathtt{a}*\mathtt{N}*(\mathtt{N}!*3)^{\mathtt{a}})$ pointers to at most $\mathtt{O}(\mathtt{N}*\mathtt{N}!*\mathtt{a}*(\mathtt{N}!*3)^{\mathtt{a}-1})$ schema path records.

Therefore, the worst case complexity of our approach in terms of runtime and space is $\mathtt{O}(\mathtt{a}*\mathtt{N}*(\mathtt{N}!*3)^{\mathtt{a}})$.

The XML Schema definitions of the worst case, where each node has all the nodes as succeeding nodes and each node is an instance node, are rare. A query that selects all the nodes of a given XML instance document is `/descendant-or-self::node()`. Other queries with multiple location steps each of which selects up to all nodes are typically not used. Therefore, it makes sense to investigate the complexity of our approach in typical cases.

According to real-world schemas and queries, we assume that the typical cases are characterized as follows: each node in an XML Schema definition $\mathtt{S}$ has only a small number of succeeding nodes compared with the number $\mathtt{N}$ of nodes in $\mathtt{S}$; for each location step of $\mathtt{Q}$, the number of nodes visited is in average less than a constant $\mathtt{C}$, and thus less than $\mathtt{C}$ schema paths are created for each location step. Therefore, after $\mathtt{Q}$ is evaluated for the typical case, $\mathtt{a}*\mathtt{C}$ nodes are visited and $\mathtt{C}$ schema paths are created, the length of each of which is at most $\mathtt{a}*\mathtt{N}$.

Therefore, the complexity of runtime and space of our approach is $\mathtt{O}(\mathtt{a}*\mathtt{N}*\mathtt{C})$ for typical cases. When the number of the nodes visited is in average less then $\mathtt{N}$ for each location step and this is quite typical based on real-word schemas and queries, the complexity of our approach in terms of runtime and space is $\mathtt{O}(\mathtt{a}*\mathtt{N}*\mathtt{N})$ for typical cases.

5. Satisfiability tester

Definition 10 (*Satisfiability of XPath queries*). A given XPath query `Q` is satisfiable according to a given XML Schema definition `XSD`, if there exists an XML document `D`, which is valid according to `XSD`, and the evaluation of `Q` on `D` returns a non-empty result. Otherwise, `Q` is unsatisfiable according to `XSD`.

Proposition 1 (*Unsatisfiable XPath queries*). If the evaluation of an XPath query `Q` on a given XML Schema definition `XSD` by the XPath–XSchema evaluator generates an empty set of schema paths, then `Q` is unsatisfiable according to `XSD`.

Proof. The XPath–XSchema evaluator is constructed in such a way that the XPath–XSchema evaluator returns an empty set of schema paths, if the constraints given in `Q` and the constraints given in `XSD` exclude the constraints of the other, i.e. the navigation paths described by `Q` cannot be mapped to the paths in `XSD`, or the values of attributes or elements given in `Q` do not conform to the types of the values of elements or attributes specified in `XSD`, or the attributes and elements specified in `Q'` are prohibited by `XSD` to appear in instance XML documents. Thus, there does not exist a valid XML document according to `XSD`, where the application of `Q` returns a non-empty result. □

If an XPath query is computed to a non-empty set of schema paths by our XPath–XSchema evaluator on an XML Schema definition, the XPath query is only *maybe* satisfiable, since the satisfiability test of XPath queries formulated in the supported subset of XPath is undecidable [2] and our satisfiability tester is incomplete. Our approach checks whether or not each location step in an XPath query `Q` conforms to the constraints given in the XML Schema definition, but our approach does not check whether or not two or more location steps in `Q` contradict each other. For example, the query `Q1` = `a[b/c][b/d]` and `Q2` = `a[not(b)]/*` are tested as satisfiable queries by our approach. However, `Q1` is unsatisfiable if the schema specifies that `b` can occur only one time and `c` and `d` cannot appear in any valid XML document simultaneously; the query `Q2` is unsatisfiable if the schema specifies that `b` is the only children of `a`.

6. Performance analysis

We have implemented a prototype of our approach in order to verify the correctness of our approach and to demonstrate the optimization potential for avoiding the evaluation of unsatisfiable XPath queries. The performance analysis focuses on the detection of unsatisfiable XPath queries by our approach and the evaluation of these unsatisfiable queries by common XPath evaluators. We also study the overhead of evaluating satisfiable XPath queries by our approach, where we compare the time of evaluating the satisfiable queries by our approach with the time of evaluating unsatisfiable queries by our approach and with the time of evaluating these satisfiable queries by common XPath evaluators, in order to prove the usability of our approach.

6.1. Test system and data

The test system for all experiments is an Intel Pentium 4 processor 2.4 GHz with 512 MB RAM, Windows XP as operating system and Java VM build version 1.4.2. We use the XQuery evaluators Saxon version 8.0 (see [16]) and Qizx version 0.4pl (see [7]) in order to evaluate the XPath queries. We use the XPathMark benchmark (see [8]) as the source of our experimental data, and generate data from 0.116 MB to 11.597 MB by using the data generator of [8]. An XML Schema definition `benchmark.xsd` (see Appendix A) is manually adapted according to the DTD `benchmark.dtd` of the XPathMark benchmark (see [8]) and the instance documents in order to integrate as many constructs of XML Schema as possible and to specify more specific data types for values of elements and attributes, which are all declared as `#PCDATA` in `benchmark.dtd`. We design two groups of unsatisfiable queries and two groups of satisfiable queries. The first group of unsatisfiable queries `Q1`–`Q11` (see Table 1) is modified from some of the XPathMark benchmark queries (see [8]) to contain erroneous semantics and structure; the second group of unsatisfiable queries `Q12`–`Q26` (see Table 3) does not conform to value-types or occurrence constraints. We correct the errors of the semantics and structure in the first group of unsatisfiable queries Q1–Q11 and get a group of satisfiable queries Q1'–Q11' (see Table 2); we modify the second group of unsatisfiable queries Q12–Q26 and obtain another group of satisfiable queries Q12'–Q26' (see Table 4), which conform to the value-types and occurrence constraints. Furthermore, the queries in these groups are also designed to contain as many constructs of the XPath language as possible in order to test how the different constructs of the XPath language influence the processing performance. We present the average results of 10 executions of these queries.

6.2. Filtering queries with incorrect semantics or structure

Fig. 13 presents the time of evaluating the queries Q1–Q11 on benchmark.xsd by our XPath–XSchema evaluator, when returning an empty set of schema paths. Our evaluator can evaluate XPath queries Q1–Q4 without recursive axes very fast, less than 0.02 s; evaluating queries Q5–Q7 with one recursive location step is on average 2.6 slower than evaluating queries Q1–Q4 without recursive location steps; evaluating queries with two descendant recursive location steps (Q8) doubles the time of evaluating queries with one descendant axis (Q5–Q7); evaluating queries (Q10 and Q11) with one descendant location step and one ancestor location step is 1.4 times slower than evaluating queries (Q8) with two descendant location steps; evaluating Q9, which has a location step //* that selects all the nodes in an XML document, is slowest, i.e. three times slower than Q5–Q7 with one recursive location step, which consists of a label nodetest. Figs. 14 and 15 present the time of evaluating these queries using the Saxon and the Qizx evaluator, respectively, when an empty result is returned. Figs. 16 and 17 present the speed-up factors achieved by our approach over the Saxon evaluator and the Qizx evaluator, respectively, when evaluating Q1–Q11. The experimental results show that our approach can check the

Table 1
Queries with incorrect semantics or structure

Queries		Reasons for unsatisfiability
Q1	/site/closed_auctions/closed_auction/annotation/description/parlist/text	parlist has no child text
Q2	/site/regions/*/item[parent::america]	item has no parent america
Q3	/site/open_auctions/open_auction[bidder//title]	bidder has no descendant title
Q4	/site/people/person[age or gender]	person has neither child age nor child gender
Q5	//person[age or gender]	person has neither child age nor child gender
Q6	//keyword[italic][bold]	keyword has no child italic
Q7	/descendant-or-self::persons	persons does not exist
Q8	//open_auction[bidder//title]	bidder has no descendant title
Q9	//*/person[age or gender]	person has neither child age nor child gender
Q10	//keyword/ancestor-or-self::mail[@title]	mail has no attribute title
Q11	//keyword/ancestor::listitem/type	listitem has no child type

Table 2
Queries with correct semantics and structure

Satisfiable Queries	
Q1'	/site/closed_auctions/closed_auction/annotation/descrition/parlist
Q2'	/site/regions/*/item[parent::namerica]
Q3'	/site/open_auctions/open_auction[bidder]
Q4'	/site/people/person
Q5'	//person
Q6'	//keyword[bold]
Q7'	/descendant-or-self::person
Q8'	//open_auction[bidder]
Q9'	//*/person
Q10'	//keyword/ancestor-or-self::mail
Q11'	//keyword/ancestor::listitem

Table 3
Queries not conforming to data-types or occurrence constraints

Queries		Reasons for unsatisfiability
Q12	/site/people/person/race	race is a prohibited element, i.e. maxOccurs = '0'.
Q13	//person/race	race is a prohibited element
Q14	/site[@owner = 'A']	owner is a prohibited attribute, i.e. use = 'prohibited'
Q15	//site[@owner = 'A']	owner is a prohibited attribute
Q16	/site/people/person/watches/watch/@expression	expression is a prohibited attribute
Q17	//watch/@expression	expression is a prohibited attribute
Q18	//*/@expression	expression is a prohibited attribute
Q19	/site/people/person[creditcard = '1234 4567 890a 1234']	creditcard is of pattern \d{4}\s*\d{4}\s*\d{4}\s*\d{4}
Q20	//creditcard[self::node() = '1234 7890 1234']	creditcard is of pattern \d{4}\s*\d{4}\s*\d{4}\s*\d{4}
Q21	//*[creditcard = '1234 456 7890 1234']	creditcard is of pattern \d{4}\s*\d{4}\s*\d{4}\s*\d{4}
Q22	//happiness[self::node() = 11]	happiness has maxInclusive = '10'
Q23	/site/people/person/profile[gender = 'M']	gender has enumeration male, female
Q24	//gender[self::node() = 'f']	gender has enumeration male, female
Q25	/site/catgraph/edge[self::node() = 'edge']	edge has no value
Q26	//edge[self::node = 123.45]	edge has no value

Table 4
Queries conforming to data-types and occurrence constraints

Satisfiable Queries	
Q12'	/site/people/person
Q13'	//person
Q14'	/site
Q15'	//site
Q16'	/site/people/person/watches/watch
Q17'	//watch
Q18'	//*
Q19'	/site/people/person[creditcard = '1234 4567 8900 1234']
Q20'	//creditcard[self::node() = '1234 7890 1234 7890']
Q21'	//*[creditcard = '1234 4567 7890 1234']
Q22'	//happiness[self::node() = 9]
Q23'	/site/people/person/profile[gender = 'male']
Q24'	//gender[self::node() = 'female']
Q25'	/site/catgraph/edge
Q26'	//edge

satisfiability of XPath queries effectively. Our approach is 488 times (and 128 times, respectively) faster on average when evaluating the queries without recursive axis, and 129 times (and 32 times, respectively) faster on average when evaluating the queries with recursive axes at 12 MB in comparison with the evaluation of the unsatisfiable queries when using the Saxon evaluator (and the Qizx evaluator, respectively).

6.3. Filtering queries not conforming to data-types or occurrence constraints

Fig. 18 presents the time of evaluating the XPath queries Q12–Q26 on benchmark.xsd by our XPath–XSchema evaluator, when it returns an empty set of schema paths. Fig. 18 shows similar results for the influence of different XPath constructs on the processing performance. Figs. 19 and 20 present the time of evaluating these queries using the Saxon and the Qizx evaluator, respectively, when an empty result is returned. Figs. 21 and 22 present the speed-up factors achieved by our approach over the Saxon evaluator and the Qizx evaluator, respectively, when evaluating these queries. Likewise, the experimental results show that our approach can check the satisfiability of XPath queries effectively. Our approach is 543 times (and 167 times, respectively) faster on average when evaluating the queries without recursive axis, and 91 times (and 36 times,

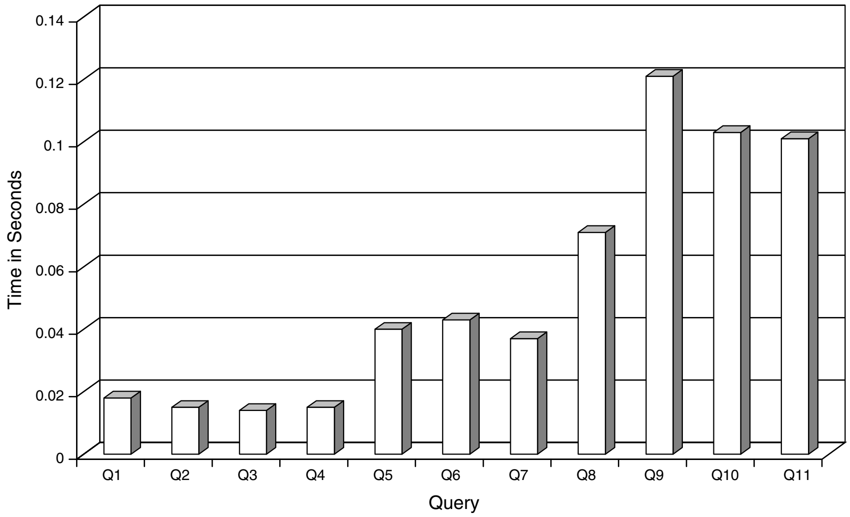

Fig. 13. Filtering Q1–Q11 by our approach.

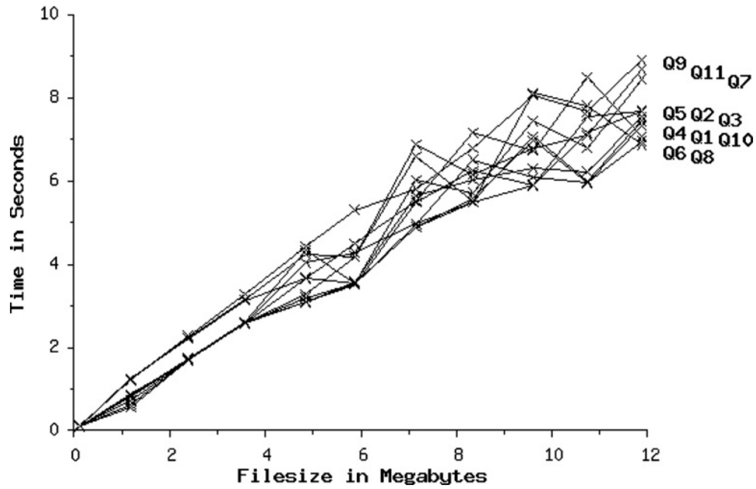

Fig. 14. Evaluating Q1–Q11 using the Saxon evaluator.

respectively) faster on average when evaluating the queries with recursive axes than Saxon (and Qizx, respectively) at 12 MB in comparison with the evaluation of the unsatisfiable queries.

6.4. Measuring the overhead of evaluating satisfiable queries

Fig. 23 presents the time of evaluating the satisfiable XPath queries Q1'–Q11' on benchmark.xsd by our XPath–XSchema evaluator, when it returns an un-empty set of schema paths, and the time of evaluating the unsatisfiable XPath queries Q1-Q11 by our evaluator for ease of comparison. Fig. 24 presents the time of evaluating the satisfiable XPath queries Q12'–Q26' on benchmark.xsd by our XPath–XSchema evaluator, when it returns an un-empty set of schema paths, and the time of evaluating the unsatisfiable XPath queries Q12-Q26 by our evaluator. Figs. 23 and 24 show that the overhead of evaluating satisfiable XPath queries is very close to the overhead of evaluating unsatisfiable XPath queries. Figs. 25 and 26 present the time of evaluating Q1'–Q11' on the XML data of different sizes by the Saxon and Qizx evaluator; Figs. 27 and 28 present the ratio of the time by our approach over the time used by Saxon and Qizx, respectively, for the evaluation of Q1'–Q11'. The results show that the ratio of the time of evaluating Q1'–Q11' by our approach over the time used by Saxon is 1% (and over the time used by Qizx is 5%, respectively) in the worst case when the size of data is

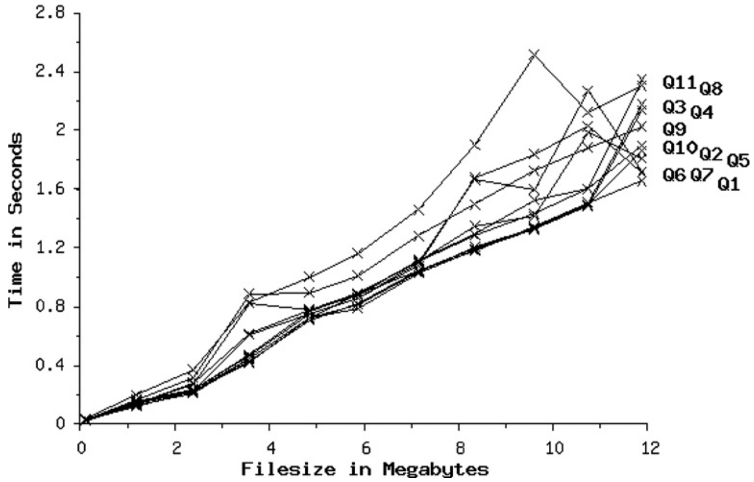

Fig. 15. Evaluating Q1–Q11 using the Qizx evaluator.

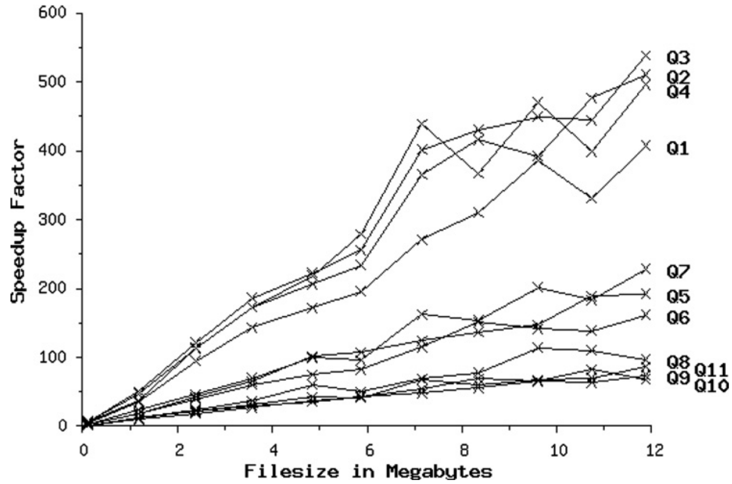

Fig. 16. Speedup by our approach over Saxon when evaluating Q1–Q11.

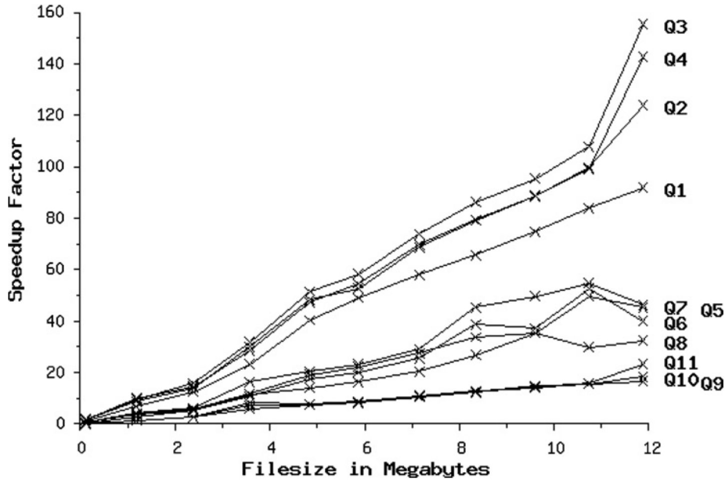

Fig. 17. Speedup by our approach over Qizx when evaluating Q1–Q11.

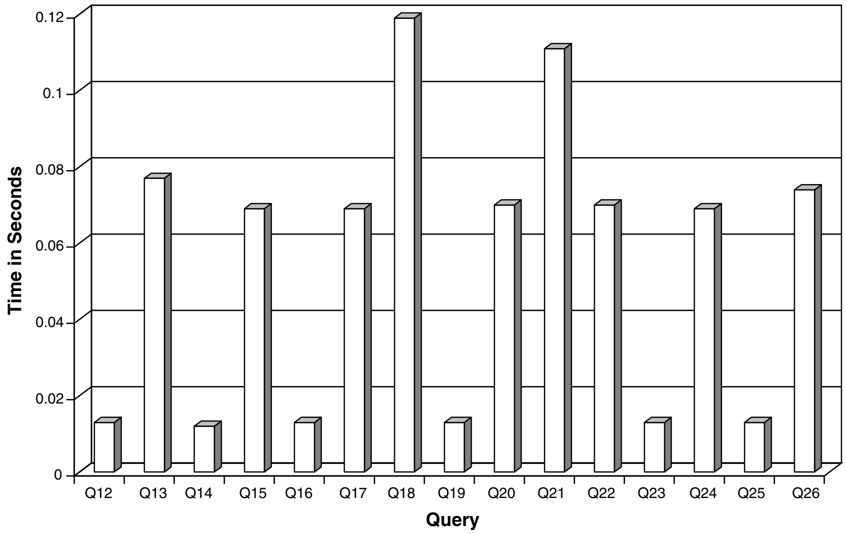

Fig. 18. Filtering `Q12`–`Q26` by our approach.

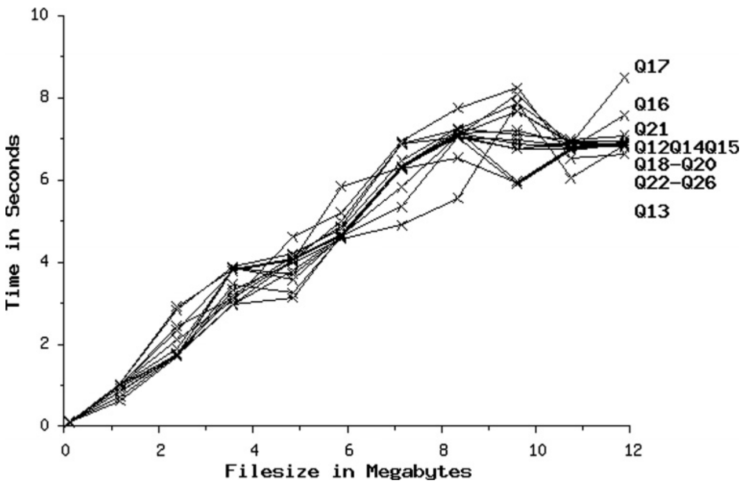

Fig. 19. Evaluating `Q12`–`Q26` using the Saxon evaluator.

12 MB. However, when the size of XML documents is very small (<100KB), the overhead of evaluating satisfiable XPath queries by our approach is quite high compared to the time of the evaluation by XPath evaluators. When the size of XML data is 100KB, the ratio of the time of evaluating the XPath queries `Q1'`–`Q8'` with at most one recursive axis (excluding // *) by our approach over Saxon (and Qizx) is 25% (and 200%, respectively); the ratio of the time of evaluating XPath queries `Q9'`–`Q11'` with two recursive axes (or with one // * location step, which selects all the nodes of XML documents) by our approach over the time used by Saxon (and by Qizx, respectively) is about 50% (and 400%, respectively). In the worst case, the ratio of the time of evaluating `Q1'`–`Q11'` by our approach over the time used by Saxon is 10% when the size of XML data is 1 MB, 5% when the size of data is 4 MB, and 2.5% when the size of data is 6 MB. In the worst case, the ratio of the time of evaluating `Q1'`–`Q11'` by our approach over the time used by Qizx is 75% when the size of XML data is 1 MB, 23% when the size of data is 2.3 MB, 10% when the size of data is 6.2 MB. Although the ratio of the time of evaluating satisfiable XPath queries by our approach over common XPath evaluators is high for small instance XML documents, the absolute time used by our approach is very small, i.e. 0.12 s in the worst case when evaluating `Q1'`–`Q11'`.

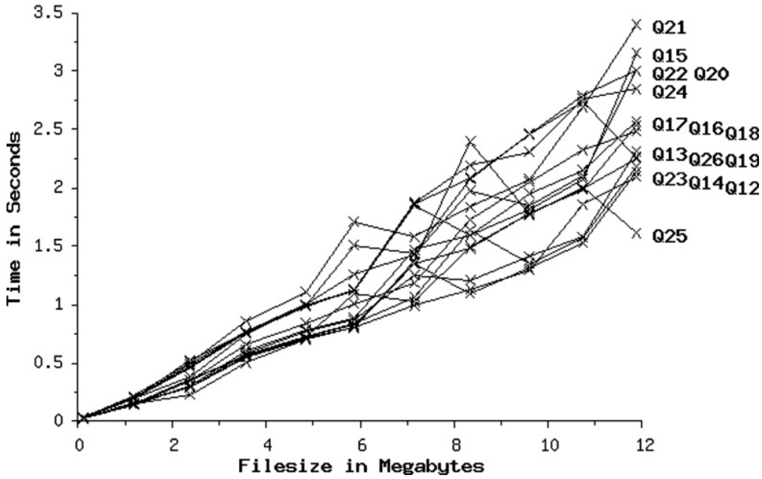

Fig. 20. Evaluating Q12–Q26 using the Qizx evaluator.

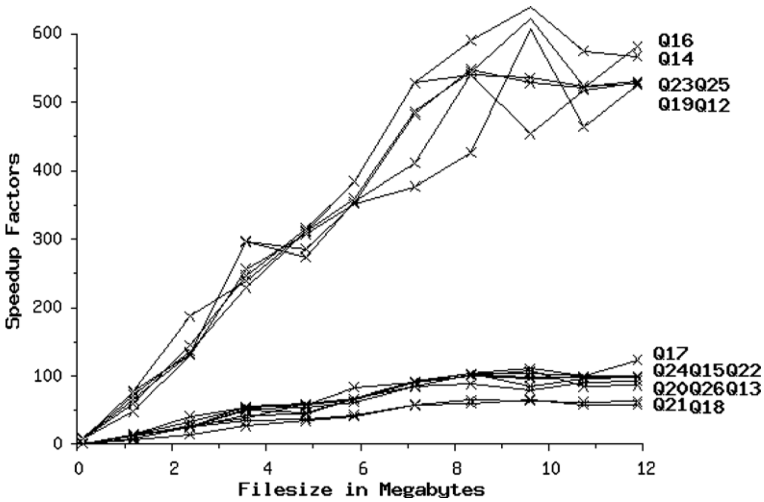

Fig. 21. Speedup by our approach over Saxon when evaluating Q12–Q26.

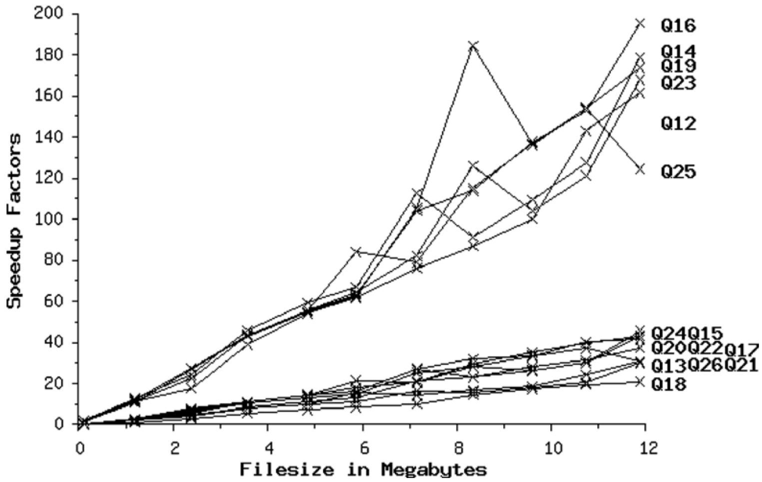

Fig. 22. Speedup by our approach over Qizx when evaluating Q12–Q26.

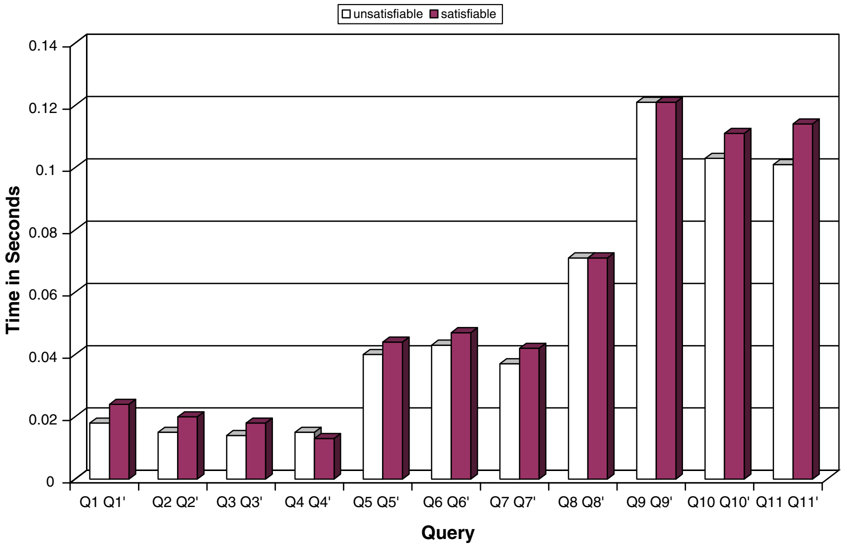

Fig. 23. Time of evaluating unsatisfiable queries (`Q1`–`Q11`) and satisfiable queries (`Q1'`–`Q11'`) by our evaluator.

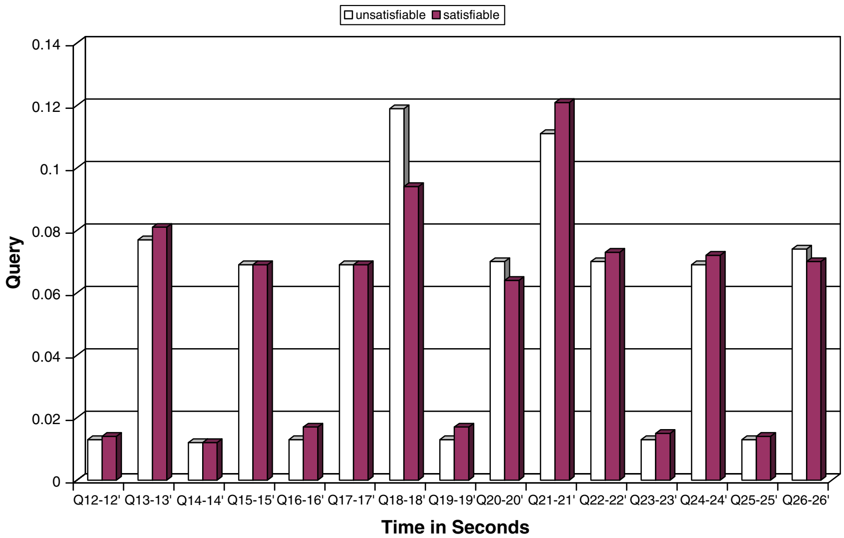

Fig. 24. Time of evaluating unsatisfiable queries (`Q12`–`Q26`) and satisfiable queries (`Q12'`–`Q26'`) by our evaluator.

7. Further related work

Many research efforts are dedicated to the satisfiability problem of XPath queries. Benedikt et al. [2] theoretically studies the complexity problem of XPath satisfiability in the presence of DTDs, and shows that the complexity of XPath satisfiability depends on the considered subsets of XPath queries and DTDs. We present a practical algorithm for testing the satisfiability of XPath queries. Hidders [14] investigates the problem of XPath satisfiability in the absence of schemas. Lakshmanan et al. [18] examines the test of satisfiability of tree pattern queries (i.e. reverse axes are not considered) with respect to non-recursive schemas. Kwong and Gertz [17] suggests an algorithm to test the satisfiability of XPath queries, but allows only non-recursive DTDs and does not support all XPath axes. We support recursive schemas and all XPath axes. Groppe et al. [12] filters the unsatisfiable XPath queries by a set of simplification rules. Groppe [9] extends the applications of satisfiability test to optimizations for XML query reformulation and shows how to reduce the containment and intersection test of XPath expressions to the satisfiability test.

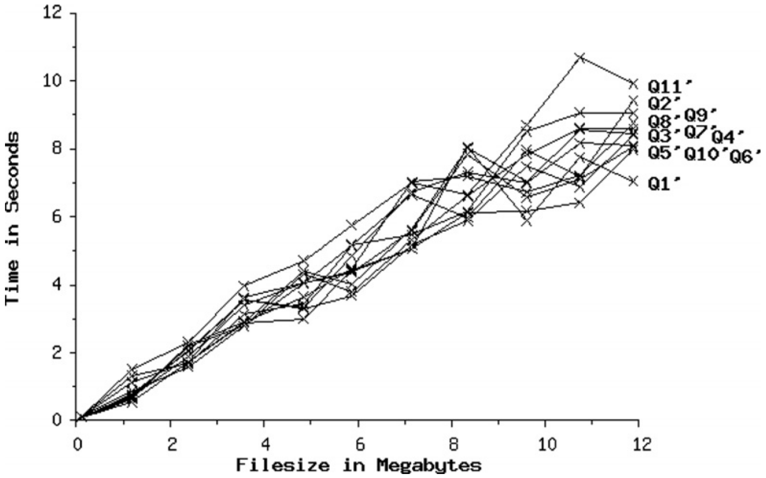

Fig. 25. Evaluation of queries `Q1'`–`Q11'` using Saxon.

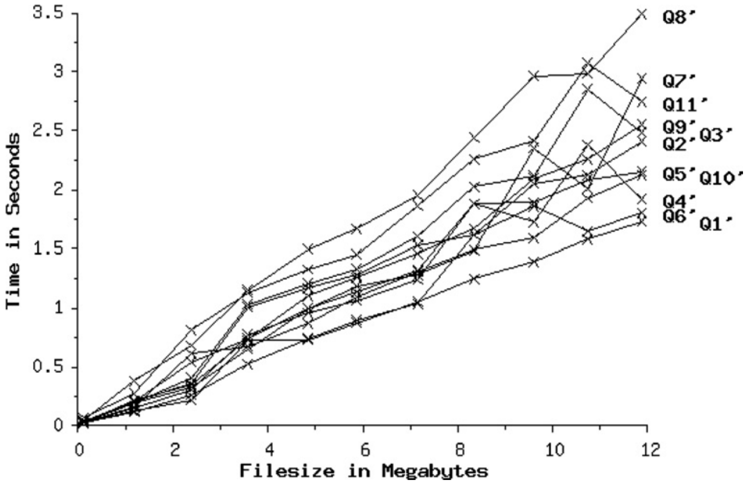

Fig. 26. Evaluation of queries `Q1'`–`Q11'` using Qizx.

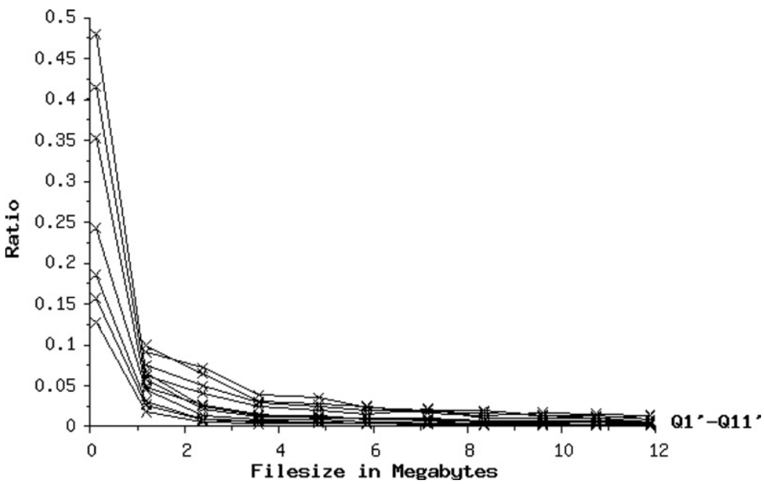

Fig. 27. Ratio of the time used by our approach over the time used by Saxon when evaluating `Q1'`–`Q11'`.

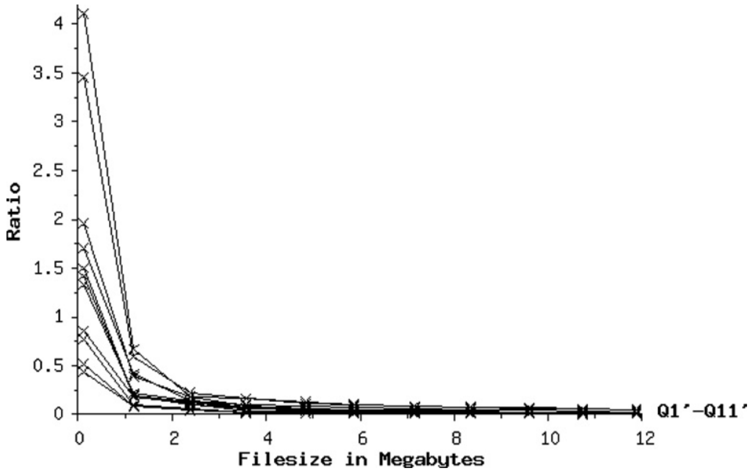

Fig. 28. Ratio of the time used by our approach over the time used by Qizx when evaluating `Q1'`–`Q11'`.

There has been work on physical optimization of XPath expressions, i.e. efficient algorithms for XPath evaluation, e.g. the XPath evaluator proposed in [13], which considers bottom-up processing of XPath expressions, indexing techniques (see [23,25]) and structural join algorithms (see [4,15]). Many research efforts focus on the minimization of XPath expressions (see [1,22,26]) by eliminating redundant steps since the size of XPath expressions significantly impacts the processing of queries. The study on the minimality of XPath closely relates to the issues of the equivalence and containment with respect to two XPath queries (see [20,27]). Olteanu et al. [21], Benedikt et al. [3] and Chan et al. [5] study logical rewriting and optimization of XPath based on the properties of XPath expressions: Olteanu et al. [21] eliminates reverse axes for efficient evaluation on streaming data, Benedikt et al. [3] identifies useful rewriting rules and Chan et al. [5] minimizes wildcard steps to speedup XPath evaluation.

8. Summary and conclusions

We have proposed a data model for the XML Schema language, which identifies the navigation paths of XPath queries on XML Schema definitions. Based on the data model, we have developed an XPath–XSchema evaluator, which evaluates XPath queries on an XML Schema definition in order to check whether or not the queries conform to the constraints imposed by the schema definition, where we also consider the powerful data typing capabilities of XML Schema. When an XPath query does not conform to the constraints in a given schema definition, our evaluator computes an empty set of schema paths, i.e. the XPath query is unsatisfiable. Otherwise, the XPath query is *maybe* satisfiable.

The experimental results of our prototype show that our approach has very low overhead, that our approach does not significantly increase the total processing time of satisfiable queries when the input XML documents are not very small, and that application of our approach can significantly optimize the evaluation of XPath queries by filtering unsatisfiable XPath queries. A speed-up factor up to several magnitudes is possible.

Future work includes filtering the XPath queries with location steps that contradict each other, and transferring our results for XPath to XQuery and XSLT.

Appendix A. Benchmark.xsd

In this section, we present the XML Schema definition `benchmark.xsd`, which we use for the performance analysis. This schema is manually adapted according to the DTD `benchmark.dtd` of the XPathMark benchmark [8] and the instance documents generated using the data generator of [8] in order to integrate as many constructs of XML Schema as possible and specify more specific data types for values of elements and attributes, which are only declared as `#PCDATA` in `benchmark.dtd`.

```
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>

  <xs:annotation>
    -- This schema was manually adapted according to --
    -- benchmark.dtd and the instance documents in order to  --
    -- integrate as many constructs of XML Schema as possible and  --
    -- specify more specific data types for values of elements and attributes, --
    -- which are only declared as #PCDATA in benchmark.dtd –
  </xs:annotation>

  <xs:element name='site' type='siteType'/>

  <xs:complexType name='siteType'>
    <xs:sequence>
      <xs:element name='regions' type='regionsType'/>
      <xs:element name='categories' type='categoriesType'/>
      <xs:element name='catgraph' type='catgraphType'/>
      <xs:element name='people' type='peopleType'/>
      <xs:element name='open_auctions' type='open_auctionsType'/>
      <xs:element name='closed_auctions' type='closed_auctionsType'/>
    </xs:sequence>
    <xs:attribute name='owner' type='xs:string' use='prohibited'/>
  </xs:complexType>

  <xs:complexType name='regionsType'>
    <xs:sequence>
      <xs:element name='africa' type='regionType'/>
      <xs:element name='asia' type='regionType'/>
      <xs:element name='australia' type='regionType'/>
      <xs:element name='europe' type='regionType'/>
      <xs:element name='namerica' type='regionType'/>
      <xs:element name='samerica' type='regionType'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='categoriesType'>
    <xs:sequence>
      <xs:element name='category' maxOccurs='unbounded'>
        <xs:complexType>
          <xs:sequence>
            <xs:element name='name' type='xs:string'/>
            <xs:element name='description' type='descriptionType'/>
          </xs:sequence>
          <xs:attribute name='id' use='required' type='xs:ID'/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='catgraphType'>
    <xs:sequence>
      <xs:element name='edge' type='edgeType' minOccurs='0' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='peopleType'>
    <xs:sequence>
      <xs:element name='person' type='personType' minOccurs='0' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='open_auctionsType'>
    <xs:sequence>
      <xs:element name='open_auction' type='open_auctionType' minOccurs='0' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='closed_auctionsType'>
    <xs:sequence>
      <xs:element name='closed_auction' type='closed_auctionType' minOccurs='0' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='regionType'>
    <xs:sequence>
      <xs:element name='item' type='itemType' minOccurs='0' maxOccurs='unbounded'/>
    </xs:sequence>
```

```
</xs:complexType>

<xs:complexType name='edgeType'>
  <xs:attribute name='from' use='required' type='xs:IDREF'/>
  <xs:attribute name='to' use='required' type='xs:IDREF'/>
</xs:complexType>

<xs:complexType name='personType'>
  <xs:complexContent>
    <xs:extension base='personType0'>
      <xs:sequence>
        <xs:element name='profile' type='profileType' minOccurs='0'/>
        <xs:element name='watches' type='watchesType' minOccurs='0'/>
        <xs:element name='race' type='xs:string' minOccurs='0' maxOccurs='0'/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='personType0'>
  <xs:sequence>
    <xs:element name='name' type='xs:string'/>
    <xs:element name='emailaddress' type='xs:string'/>
    <xs:element name='phone' type='xs:string' minOccurs='0'/>
    <xs:element name='address' type='addressType' minOccurs='0'/>
    <xs:element name='homepage' type='xs:string' minOccurs='0'/>
    <xs:element name='creditcard' type='creditcardType' minOccurs='0'/>
  </xs:sequence>
  <xs:attribute name='id' use='required' type='xs:ID'/>
</xs:complexType>

<xs:complexType name='open_auctionType'>
  <xs:sequence>
    <xs:element name='initial' type='xs:float'/>
    <xs:element name='reserve' type='reserveType' minOccurs='0'/>
    <xs:element name='bidder' type='bidderType' minOccurs='0' maxOccurs='unbounded'/>
    <xs:element name='current' type='xs:float'/>
    <xs:element name='privacy' type='privacyType' minOccurs='0'/>
    <xs:element name='itemref' type='itemrefType'/>
    <xs:element name='seller' type='sellerType'/>
    <xs:element name='annotation' type='annotationType'/>
    <xs:element name='quantity' type='quantityType'/>
    <xs:element name='type' type='typeType'/>
    <xs:element name='interval' type='intervalType'/>
  </xs:sequence>
  <xs:attribute name='id' use='required' type='xs:ID'/>
</xs:complexType>

<xs:complexType name='closed_auctionType'>
  <xs:sequence>
    <xs:element name='seller' type='sellerType'/>
    <xs:element name='buyer' type='buyerType'/>
    <xs:element name='itemref' type='itemrefType'/>
    <xs:element name='price' type='xs:float'/>
    <xs:element name='date' type='dateType'/>
    <xs:element name='quantity' type='quantityType'/>
    <xs:element name='type' type='typeType'/>
    <xs:element name='annotation' type='annotationType' minOccurs='0'/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name='itemType'>
  <xs:sequence>
    <xs:element name='location' type='xs:string'/>
    <xs:element name='quantity' type='quantityType'/>
    <xs:element name='name' type='xs:string'/>
    <xs:element name='payment' type='xs:string'/>
    <xs:element name='description' type='descriptionType'/>
    <xs:element name='shipping' type='xs:string'/>
    <xs:element name='incategory' type='incategoryType' maxOccurs='unbounded'/>
    <xs:element name='mailbox' type='mailboxType'/>
  </xs:sequence>
  <xs:attribute name='id' use='required' type='xs:ID'/>
  <xs:attribute name='featured'/>
</xs:complexType>
<xs:complexType name='descriptionType'>
  <xs:choice>
```

```
      <xs:element name='text' type='textType'/>
      <xs:element name='parlist' type='parlistType'/>
    </xs:choice>
  </xs:complexType>

  <xs:complexType type='addressType'>
    <xs:sequence>
      <xs:element name='street' type='xs:string'/>
      <xs:element name='city' type='xs:string'/>
      <xs:element name='country' type='xs:string'/>
      <xs:element name='province' type='xs:string' minOccurs='0'/>
      <xs:element name='zipcode' type='xs:string'/>
    </xs:sequence>
  </xs:complexType>

  <xs:simpleType name='creditcardType'>
    <xs:restriction base='xs:string'>
      <xs:pattern value='\d{4}\s*\d{4}\s*\d{4}\s*\d{4}'/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name='profileType'>
    <xs:sequence>
      <xs:element name='interest' type='interestType' minOccurs='0' maxOccurs='unbounded'/>
      <xs:element name='education' type='educationType' minOccurs='0'/>
      <xs:element name='gender' type='genderType' minOccurs='0'/>
      <xs:element name='business' type='businessType'/>
      <xs:element name='age' type='ageType' minOccurs='0'/>
    </xs:sequence>
  <xs:attribute name='income' type='xs:flocat'/>
</xs:complexType>

<xs:complexType name='watchesType'>
  <xs:sequence>
    <xs:element name='watch' minOccurs='0' maxOccurs='unbounded'>
      <xs:complexType>
        <xs:complexContent>
          <!-- emptyp content model -->
          <xs:restriction base='xs:anyType'>
            <xs:attribute name='open_auction' use='required' type='xs:IDREF'/>
            <xs:attribute name='expression' use='prohibited' type='xs:string'/>
          </xs:restriction>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

<xs:complexType name='reserveType' mixed='true'>
</xs:complexType>

<xs:complexType name='bidderType'>
  <xs:sequence>
    <xs:element name='date' type='dateType'/>
    <xs:element name='time' type='xs:time'/>
    <xs:element name='personref' type='personrefType'/>
    <xs:element name='increase' type='xs:float'/>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name='privacyType'>
  <xs:restriction base='xs:string'>
    <xs:enumeration value='Yes'/>
    <xs:enumeration value='No'/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name='itemrefType'>
  <xs:attribute name='item' use='required' type='xs:IDREF'/>
</xs:complexType>

<xs:complexType name='sellerType'>
  <!-- empty content model -->
  <xs:complexContent>
    <xs:restriction base='xs:anyType'>
      <xs:attribute name='person' use='required' type='xs:IDREF'/>
    </xs:restriction>
```

```
  </xs:complexContent>
</xs:complexType>

<xs:complexType name='annotationType'>
  <xs:sequence>
    <xs:element name='author'>
      <!-- anonomously complex type definition -->
      <xs:complexType>
        <!-- empty content model -->
        <xs:complexContent>
          <xs:restriction base='xs:anyType'>
            <xs:attribute name='person' use='required' type='xs:IDREF'/>
          </xs:restriction>
        </xs:complexContent>
      </xs:complexType>
    </xs:element>
    <xs:element name='description' type='descriptionType' minOccurs='0'/>
    <xs:element name='happiness'>
      <xs:simpleType>
        <xs:restriction base='happinessType1'/>
      </xs:simpleType>
    </xs:element>
  </xs:sequence>
</xs:complexType>

  <xs:simpleType name='quantityType'>
    <xs:restriction base='xs:int'>
      <xs:minInclusive value='0'/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name='typeType'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value='Regular'/>
      <xs:enumeration value='Featured'/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name='intervalType'>
    <xs:sequence>
      <xs:element name='start' type='dateType'/>
      <xs:element name='end' type='dateType'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='buyerType'>
    <xs:attribute name='person' use='required' type='xs:IDREF'/>
  </xs:complexType>

  <xs:complexType name='incategoryType'>
    <xs:attribute name='category' use='required' type='xs:IDREF'/>
  </xs:complexType>

  <xs:complexType name='mailboxType'>
    <xs:sequence>
      <xs:element name='mail' type='mailType' minOccurs='0' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='textType' mixed='true'>
    <xs:choice minOccurs='0' maxOccurs='unbounded'>
      <xs:element name='bold' type='textType'/>
      <xs:element name='keyword' type='textType'/>
      <xs:element name='emph' type='textType'/>
    </xs:choice>
  </xs:complexType>

  <xs:complexType name='parlistType'>
    <xs:sequence minOccurs='0' maxOccurs='unbounded'>
      <xs:element name='listitem' type='listitemType'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='interestType'>
    <xs:attribute name='category' use='required' type='xs:IDREF'/>
</xs:complexType>
```

```
<xs:complexType name='educationType' mixed='true'>
</xs:complexType>

<xs:simpleType name='genderType'>
  <xs:restriction base='xs:string'>
    <xs:enumeration value='male'/>
    <xs:enumeration value='female'/>
   </xs:restriction>
 </xs:simpleType>

  <xs:simpleType name='businessType'>
    <xs:restriction base='xs:string'>
      <xs:enumeration value='Yes'/>
      <xs:enumeration vaule='No'/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name='ageType'>
    <xs:restriction base='ageType1'>
      <xs:maxInclusive value='99'/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name='ageType1'>
    <xs:restriction base='xs:int'>
      <xs:minInclusive value='18'/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name='personrefType'>
    <xs:attribute name='person' use='required' type='xs:IDREF'/>
  </xs:complexType>

  <xs:complexType name='authorType'>
    <xs:attribute name='person' use='required' type='xs:IDREF'/>
  </xs:complexType>

  <xs:simpleType name='happinessType1'>
    <xs:restriction base='happinessType2'>
      <xs:maxInclusive value='10'/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name='happinessType2'>
    <xs:restriction base='xs:int'>
      <xs:maxInclusive value='100'/>
      <xs:minInclusive value='0'/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name='mailType'>
    <xs:sequence>
      <xs:element name='from' type='xs:string'/>
      <xs:element name='to' type='xs:string'/>
      <xs:element name='date' type='dateType'/>
      <xs:element name='text' type='textType'/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name='listitemType'>
    <xs:choice minOccurs='0' maxOccurs='unbounded'>
      <xs:element name='text' type='textType'/>
      <xs:element name='parlist' type='parlistType'/>
    </xs:choice>
  </xs:complexType>

  <xs:simpleType name='dateType'>
    <xs:restriction base='xs:string'>
      <xs:pattern value="\d{2}/\d{2}/\d{4}"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- never used elements

  <xs:element name='amount'>
    <xs:complexType mixed='true'>
    </xs:complexType>
```

```
        </xs:element>

        <xs:element name='income' type='xs:float'/>

        <xs:element name='status'>
          <xs:complexType mixed='true'>
          </xs:complexType>
        </xs:element>
        -->

</xs:schema>
```

References

[1] S. Amer-Uahis, S. Cho, L.K.S. Laksmanan, D. Srivastava, Mininization of tree pattern queries, in: SIGMOD 2001.
[2] M. Benedikt, W. Fan, F. Geerts, XPath Satisfiability in the presence of DTDs, in: PODS 2005.
[3] M. Benedikt, W. Fan, G.M. Kuper, Structural properties of XPath fragments, in: ICDT 2003.
[4] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: optimal XML pattern matching, in: SIGMOD 2002.
[5] C.Y. Chan, W. Fan, Y. Zeng, Taming XPath Queries by Minimizing Wildcard Steps, in: VLDB 2004.
[6] W. Fan, C. Chan, M. Garofalakis, Secure XML querying with security views, in: SIGMOD 2004.
[7] X. Franc, Qizx/open version 0.4p1, http://www.xfra.net/qizxopen/, 2004.
[8] M. Franceschet, XPathMark – An XPath benchmark for XMark. Research Report PP-2005-04, University of Amsterdam, the Netherlands, 2005.
[9] S. Groppe, XML Query Reformulation for XPath, XSLT and XQuery, Sierke-Verlag, Göttingen, Germany, 2005, ISBN 3-933893-24-0.
[10] J. Groppe, S. Groppe, Filtering Unsatisfiabile XPath Queries, in: ICEIS 2006.
[11] J. Groppe, S. Groppe, A prototype of a schema-based XPath satisfiability tester, in: DEXA 2006.
[12] S. Groppe, S. Böttcher, J. Groppe, XPath Query simplification with regard to the elimination of intersect and except operators, in XSDM'06 in association with ICDE'06.
[13] G. Gottlob, C. Koch, R. Pichler, Efficient algorithms for processing XPath queries, in: VLDB 2002.
[14] J. Hidders, Satisfiability of XPath expressions, DBPL 2003, LNCS, vol. 2921, pp. 21–36.
[15] H. Jiang, W. Wang, H. Lu, J.X. Yu, 2003. Holistic twig joins on indexed XML documents, in: VLDB 2003.
[16] M.H. Kay, Saxon – The XSLT and XQuery Processor, http://saxon.sourceforge.net, 2004.
[17] A. Kwong, M. Gertz, Schema-based Optimization of XPath Expressions. Technical Report, University of California, 2002.
[18] L. Lakshmanan, G. Ramesh, H. Wang, Z. Zhao, On testing satisfiability of tree pattern queries, in: VLDB 2004.
[19] W. Martens, F. Neven, Fronties of tractability for typechecking simple XML transformations, in: VLDB 2004.
[20] G. Miklau, D. Suciu, Containment and equivalence for an XPath fragment, in: PODS 2002.
[21] D. Olteanu, H. Meuss, T. Furche, F. Bry, XPath: Looking Forward. XML-based Data Management (XMLDM), EDBT Workshops, 2002.
[22] P. Ramanan, Efficient algorithms for minimizing tree pattern queries, in: SIGMOD 2002.
[23] P. Rao, B. Moon, PRIX: indexing and querying XML using Prufer sequences, in: ICDE 2004.
[24] P. Wadler, Two Semantics for XPath, Technical Report, 2000.
[25] H. Wang, S. Park, W. Fan, P.S. Yu, ViST: a dynamic index method for querying XML data by tree structures, in: SIGMOD 2003.
[26] P.T. Wood, Minimising simple XPath expressions, in: WebDB, 2001, pp. 13–18.
[27] P.T. Wood, On the equivalence of XML patterns, LNCS, vol. 1861, Springer, 2000, pp. 1152–1166.
[28] W3C, Extensible Markup Language (XML) 1.0 (third ed.), W3C Recommendation, www.w3.org/TR/REC-xml, 2004.
[29] W3C, XPath Version 1.0, W3C Recommendation, www.w3.org/TR/xpath/, 1999.
[30] W3C, XPath Version 2.0, W3C Working Draft, www.w3.org/TR/xpath20/, 2003.
[31] W3C, XML Schema Part 1: Structures Second Edition. W3C Recommendation, www.w3.org/TR/xmlschema-1, 2004.
[32] W3C, XML Schema Part 2: Datatypes Second Edition. W3C Recommendation, www.w3.org/TR/xmlschema-2, 2004.

Jinghua Groppe earned her Bachelor degree in Computer Science and Applications from the Beijing Polytechnic University in 1989 and her Master degree in Computer Science from the University of Amsterdam in 2001. She worked as Software Engineer in the Chinese Academy of Launch Vehicle Technology/China Aerospace Corporation from 1989 to 1999. She was Scientific Employee in the Department of Computer Science/University of Paderborn from 2001 to 2005 and in the Institute of Computer Science/University of Innsbruck from 2005 to 2006. She is currently working on her Doctorate thesis. She worked in the projects EUQOS, UBISEC, E-Colleg, VHE and ASG. All projects were funded by the European Union. Her research interests include XML and semistructured data, satisfiability tester, containment tester, profiles, Semantic Web, caching and mobile devices.

Sven Groppe earned his diploma degree in Informatik (Computer Science) from the University of Paderborn in 2002 and his Doctor degree from the University of Paderborn in 2005. From 2005 to 2007, he worked as postdoc in the University of Innsbruck. He is currently working as postdoc in the University of Lübeck. In 2001/2002, he worked in the project B2B-ECOM, which dealt with distributed internet market places for the electrical industry. From 2002 to 2004, he worked in the project MEMPHIS in the area of premium services. From 2005 to 2006, he worked in the projects ASG and TripCom in the areas of Semantic Web Services. All projects were funded by the European Union. His research interests include XML and semi-structured data, query reformulation, data integration in heterogenous environments, Semantic Web, SPARQL, distributed systems, electronic market places, web services and mobile devices.