XSPath: Navigation on XML Schemas Made Easy

Federico Cavalieri, Member, IEEE, Giovanna Guerrini, Member, IEEE, and Marco Mesiti, Member, IEEE

Abstract—Schemas are often used to constrain the content and structure of XML documents. They can be quite big and complex and, thus, difficult to be accessed manually. The ability to query a single schema, a collection of schemas or to retrieve schema components that meet certain structural constraints significantly eases schema management and is, thus, useful in many contexts. In this paper, we propose a query language, named XSPath, specifically tailored for XML schema that works on logical graph-based representations of schemas, on which it enables the navigation, and allows the selection of nodes. We also propose XSPath/XQuery-based translations that can be exploited for the evaluation of XSPath queries. An extensive evaluation of the usability and efficiency of the proposed approach is finally presented within the EXup system [9].

Index Terms—XML schema, XPath, schema querying

1 INTRODUCTION

Despite XML has been conceived schema-free, there are contexts in which applications, database servers, and users can take advantage of the knowledge of schema information to constrain the content and structure of XML documents. Different schema definition languages have been proposed (DTD, RelaxNG [12], XML schema [40]) and their expressiveness has been compared [33]. The most commonly adopted language is the W3C recommendation XML schema, which employs an XML-based representation. Schemas can be small and simple in application contexts where data are quite regular, as the DBLP schema [27] for scientific publications, or complex and big, in large domains such as aviation (AIXM) or weather information (WXXM). In the last case, their manual inspection is quite complex, also because of the verbose textual representation of XML schema. Moreover, several versions of the same schema are sometimes produced [22], to reflect its evolution over time and to capture domain constraints as accurately as possible. In this case, manual management of schema versions is even more problematic.

As a solution of this problem, logical representations of XML schemas [35], [23], [31] have been proposed as well as tools [1], [37], [16] allowing the exploration of schemas by navigating on their graphical representations. These graphical exploration tools overcome the problems arising from the verbose, textual representation of XML schema but do not adequately fulfill all the schema retrieval needs. Specifically, a graphical exploration tool is not adequate to query schema collections, representing for instance related schemas or different versions of the same schema, nor for all the situations where schema elements need to be retrieved (based on some conditions) and to be subsequently target of some schema manipulation operations.

A notable example of application context where graphical exploration is not enough and a schema query language is needed is schema evolution [9]. In such a context, a query language enables the identification of schema components to be updated. Other relevant application contexts include retrieving information from multiple heterogeneous sources, in both query formulation and query optimization [11], [26], as well as identification and specification of mappings between schema elements [3].

Since a schema in XML schema is an XML document itself, a simple approach to query schemas could be to use an XML query language like XPath [42] or XQuery [43] for fulfilling the previously discussed retrieval needs. However, this solution would result in the specification of complex expressions that do not reflect the user intuitions in query formulation. Suppose, for instance, that we wish to denote the global library element in the schema in Fig. 1. The XPath expression /schema/element[@name="library"] could be specified to retrieve the library element. This expression is verbose and a simpler expression like /library would be preferable. A simple extension of this query like: “find the book element declaration within library” would make the XPath specification much more complicated, while an expression like /library/book would be much more intuitive. Moreover, the occurrence of references to element declarations and the possibility to define the type of an element as global require to specify expressions over internal links. Navigation of such links, however, is awkward in XPath.

\begin{itemize}
  \item F. Cavalieri and G. Guerrini are with the DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi, Università di Genova, Via Dodecaneso, 35, Genova 16146, Italy. E-mail: federico.cavalieri, giovanna.guerrini@unige.it.
  \item M. Mesiti is with the Dipartimento di Informatica, Università degli Studi di Milano, Milano, Italy. E-mail: mesiti@di.unimi.it.
\end{itemize}

Manuscript received 22 Nov. 2011; revised 5 July 2012; accepted 22 Nov. 2012; published online 13 Dec. 2012.

Recommended for acceptance by T. Grust.

For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2011-11-0718. Digital Object Identifier no. 10.1109/TKDE.2012.247.
Thus, in this paper we propose a query language, named XSPa\textit{th}, specifically tailored for the retrieval of XML schema components. This language offers the ability of expressing retrieval needs on a logical representation of schemas, leaving aside the verbose XML schema syntax, thus greatly simplifying retrieval tasks, offering at the same time all the power and flexibility of a query language over graphical inspection tools. A key feature of the proposed language is that the expressions are specified on a two-level graph-based abstraction of schemas. These abstract representations make the specification of the expressions easier and leave to the language interpreter the burden of solving the gap between the logical (graph-based) and physical (textual) representations of schemas. The language captures a wide spectrum of retrieval needs in an XML schema that include: navigation on the nesting structure of element declarations, navigation on the type hierarchy generated by type restriction and extension, expression of conditions on cardinality, uniqueness and key constraints, types of element content and annotations. Moreover, full-text querying on textual content is supported. The queries capturing such needs can be issued against a specific schema, when we are interested in determining some of its components, or against a collection of schemas, when we are interested in retrieving the schemas in the collection containing a certain component or in contrasting or relating the different specifications of a given component across the schemas in the collection. As special, relevant cases, the collection may correspond to different versions of the same schema, or to alternative schemas constraining documents that need to be interchanged among applications, or to schemas among which a mapping needs to be devised.

The paper also proposes approaches for the translation of XSPa\textit{th} expressions in XPath/XQuery expressions\(^1\) to be evaluated on the textual representations of the target schemas. The choice of implementing XSPa\textit{th} via translation is mainly due to the possibility of relying on XPath/XQuery engines rather than developing a new engine specifically tailored for the evaluation of XSPa\textit{th} expressions. While the two different usage modalities of the language (on a given schema, on an arbitrary and unconstrained schema collection) have limited impact on its syntax and semantics, the impact on the translation process is high. Indeed, a schema-dependent translation produces an XPath expression that is only guaranteed to obtain the correct result when evaluated exactly on the same schema on which it has been translated. By contrast, with a schema-independent translation, an XQuery expression is obtained, guaranteed to produce the correct result when evaluated on an arbitrary schema. However, a naive translation often produces, in this second case, significantly longer expressions. Thus, specific static type analysis techniques have been defined to reduce the size of the resulting expression by taking advantage, in the translation of each step, of information that we can extract from all the steps in the expression.

Our contributions can be summarized as follows:

1. definition of the syntax and semantics of a XPath-like navigational language for querying schemas expressed in XML schema (Section 4);
2. definition of a type analysis for the developed language that enables optimized schema-independent translations (Section 5);
3. definition of the schema-dependent translation of XSPa\textit{th} to XPath and of its schema-independent translation to XQuery, relying on the type analysis (Section 6); and
4. extensive evaluation of the usability of the developed language and efficiency of the translation process (Section 7) within the \textsc{Exup} system\[^{[9]}\].

Before defining the XSPa\textit{th} language, most closely related work is surveyed in Section 2 and a two-level representation of XML schemas as labeled directed graphs is defined in Section 3. A preliminary version of a very limited part of this paper already appeared\[^{[8]}\]. However, only the XSPa\textit{th} syntax was presented in\[^{[8]}\], no formal definition of the semantics, no translation specification, and no experimental evaluation were provided.

2 RELATED WORK

Our work takes advantage of concepts developed in the context of schema querying and exploration for various kinds of data and of schema-based XML processing. In what follows, we survey the most closely related work.

Schema querying and exploration. Schema querying has been considered in the relational\[^{[26]}\] and object-oriented

\[^{1}\]Actually, on their full-text extension, if full-text facilities are used in the XSPa\textit{th} expression to be translated.
[11] context. However, the structure of the schemas considered in such proposals is much less variable than XML schemas, and so are the corresponding schema queries. Visual environments have been proposed for navigating conceptual schemas such as E-R diagrams [15]. Query languages have been proposed for UML class diagram as well [7], but they are intended to querying instances rather than schemas and are grounded on logic languages and ontological modeling rather than on hierarchical structure navigation. In the context of ontologies, SparQL navigational capabilities have been recently extended [25], [28] by introducing property-paths, which are regular expressions that retrieve pairs of nodes in an RDF graph that are connected by paths conforming to the expression. Languages for specifying and retrieving relevant OWL ontology fragments, by contrast, have been proposed in [24], [36], with the purpose of ontology modularization. With the goal to build new closed and consistent ontologies, the language OntoPath, inspired by XPath, has been proposed in [24]. Finally, other approaches, among which a relevant recent representative is [32], addressed the issue of ontology exploration and navigation, abstracting from representational details for effective visualization, and allowing the user to make sense of the ontologies content and structure.

XML schema exploration and querying. Many approaches have been proposed by the academic [35], [23], [31] and industrial communities (e.g., XMLSpy [1], <oXygen/> [37], and Eclipse schema editor [16]) to provide a logical model of an XML schema that highlights its graph structure omitting the details of the syntax used to describe it. Our representation has a similar goal, allowing the modeling of the type hierarchy and the representation of the schema at two levels of abstraction.

Few approaches have been proposed to evaluate XPath expressions on XML schemas [31], [29]. In these approaches, XPath expressions are specified on an instance document and evaluated on the graph representation of its schema. This is useful to check the expression correctness but it does not allow the identification of specific XML schema components (like finding the elements that are alternative the definition of a complex type). By contrast, our main goal is making the identification of schema components easier and our language includes a wide spectrum of facilities in this direction. A different approach, based on the specification of APIs to navigate the structure of XML schemas within Java programs, has been proposed in [30], [41]. Its relationship with XPath is the same holding between DOM APIs [39] and XPath.

XPath and XPath extensions. XPath 2.0 [38] introduces the concept of element retrieval relying on type information. When an XML document is associated with a schema, its internal representation is annotated with schema information (PVTI representation [42]). This way, XPath 2.0 expressions can contain conditions on the type of elements and attributes, on the repeatability of an element, and so on. XSIPath, however, differs from XPath 2.0 because it is specifically tailored to work on schemas rather than on documents. Moreover, XSIPath has been designed to work on a graph schema model more complex than the XDM (tree) model of XML documents. The XSIPath semantics relies on the XPath semantics [5], [44]. Key difference with XPath semantics is the underlying data model which is a graph that can be navigated at two levels of abstraction. Moreover, XSIPath includes specifically tailored constructs to define conditions on types rather than on values. Other XPath extensions have been proposed in the literature. Among them, we mention the XSIPath language [34], proposed for the spatial querying of webpages, and the extension for querying webpages combining web services [17].

Schema-based XML query and update processing. Schemas have been widely exploited for enhancing safety and efficiency of XML processing, an overview of different proposals can be found in [13]. Specifically, schema-based logical satisfiability testers for XML queries (e.g., [6], [19], [20]) as well as XML query rewrites according to a schema for query optimization purposes (e.g., [10], [19]) have been proposed. This body of work is related to ours because of the reliance on schemas, which are, however, a mean (to achieve safe and efficient data management) rather than a goal (schema management) as in our proposal. Moreover, a considerable amount of work have been carried on in the context of type inference for XML queries and updates (e.g., [4], [14]). This work is related to the type analysis we propose, though our type analysis is targeted at schema navigational expressions.

3 SCHEMA REPRESENTATION

XML schema [40] provides a rich set of facilities to constrain the structure and content of XML documents. In this paper, we focus on the characteristics of an XML schema that are contained in the schema library.xsd in Fig. 1.

Under the root element <schema>, global types, elements, attributes, and groups of elements/attributes may appear. A unique name is associated with them and is exploited for referencing in the declaration of other types/elements/attributes (e.g., the declaration of the library global element contains a reference to the global book element). Local elements, attributes, and types may be specified inside global elements/types. Local type definitions are nameless. Types can be simple (both built-in and user-defined) or complex (e.g., personType). Complex types can be the restriction or extension of other types (e.g., authorType is an extension of the complex type personType), whereas simple types can be derived as list, union or restriction of other simple types (e.g., isbnType is a restriction of the built-in type unsignedLong).

Type derivation leads to type hierarchies. Complex types can be structured by aggregating subelements through the grouping operators sequence, choice, and all. Groups of elements/attributes that appear in different parts of the schema can be collected in named groups and then referred (e.g., attribute group acctg). Constraints can be specified on the repeatability of operators, local attributes, and elements (e.g., minOccurs and maxOccurs for the book element). Annotations can be specified within the declarations of elements and types (e.g., documentation for the type of the library element). Substitution groups can be defined for global elements, so that their instances can be used in place of the instances of other global elements. Moreover, the nillable attribute can be specified in element
declarations to allow null-valued instances. Finally, uniqueness, key, and key reference constraints for elements and attributes can be specified by means of the unique, key, keyRef, and keyRefTo attributes. Complex types and elements can be defined abstract, if desired, while specific kinds of derivation or substitution can be disallowed by means of the final and block attributes. The content of complex types content can be defined, according to the XML schema definition, as:

1. empty, if no element or text children are allowed,
2. simple, if only text children are allowed,
3. element-only, if only element children are allowed, and finally,
4. mixed, if both element and text children are allowed.

### 3.1 Low-Level Schema Representation

The low-level representation of a schema is a graph $S$. The set $NT = \{\text{root, element, type, attribute, group, operator, annotation}\}$ contains the node types that can appear in $S$. Each node, independently from its type, has a name property. For elements, global types, attributes and groups it has the same value of their name or ref attribute, while operators are named sequence, choice or all, and annotations are named either documentation or appinfo. A unique system generated name is associated with local (anonymous) types. Moreover, further properties can be specified for each node depending on its type. For example, local element and operator nodes can have the minOccurs, maxOccurs properties to specify repetition constraints. Nodes representing simple types can be associated with a base type and the restrictions represented through (name, value) pairs like (pattern, \(\text{\d(3)}\*\text{\d(4)}\)). To distinguish node types in the graphical representations, different graphical symbols are employed. The most common properties (minOccurs and maxOccurs) are drawn next to the node name (default cardinalities are omitted for simplicity).

Nodes can be connected through explicit, implicit or type hierarchy edges. Explicit edges correspond to child/ descendant relationships in the XDM representation of the schema and can connect: 1) the root node with global element, attribute, group, annotation and type nodes; 2) type nodes with their attribute, group and annotation nodes (if any), and operator nodes; 3) operator nodes with their children operator, element and group nodes. Implicit edges bind: 1) elements or attributes with their global types or declarations (e.g., the edge between the author element and the authorType complex type); and 2) complex types defined as extension with the operators and attributes of the base complex types (e.g., the edge between the authorType node and the sequence node of personType). Type hierarchy edges bind pairs of types such that the source type is a restriction or an extension of the destination type.

In the left side of Fig. 2, the graph representation of the library.xsd schema in Fig. 1 is reported. Built-in types are shown next to the symbol representing the element. Since schemas are XML documents, a total document order $<$ is defined among its elements. For each node a numerical identifier is shown in Fig. 2(left side).

### 3.2 High-Level Schema Representation

The high-level representation of a schema is a graph obtained from its low-level representation by removing the nodes of operator, annotation, and group types. The elements/attributes the operators bind are attached to the corresponding type. Similarly, a type definition is connected with the attributes and elements specified in the referenced attribute/element groups. This representation allows the specification of simple and concise path expressions that do not require to consider the operators and groups used within type definitions. Steps in the path expressions can move from the two-levels representation depending on the specific needs, thus leading to a more flexible query language. The right side of Fig. 2 shows the high-level representation of our schema.

### 4 Syntax and Semantics

The basic building block of XSParse is the path expression. The notion of XSParse expression is similar to that of XPath
expression and consists of a sequence of one or more steps. Key differences are the kinds of predicates that can be exploited and the possibility to navigate the implicit and type hierarchy edges as well as the reliance on low- and high-level schema representations. In the remainder of the section, we first focus on the specification of a single step. Then, we present the abbreviated syntax, examples of navigational expressions, and the semantics of the developed language.

4.1 Steps
Starting from a context node, a step is used to identify a set of nodes. Step specification, described by the BNF in Fig. 3, includes the step components: axis, node selector, and (optionally) predicates, discussed in the following:

Axis. The axis determines the direction toward which nodes are identified starting from the context node and moving at a given level of abstraction. An axis is, thus, composed by an abstraction level and an axis name. The abstraction level is denoted by LL for low level and by HL for high level. Besides, the XPath axes, the axes base-type, and derived-type are introduced for navigating the type hierarchy edges.

In our graph representation, an element node has at most a single child type node. Thus, for simplifying the specification of expressions, the child axis is not limited to the direct children in the graph. Indeed, whenever the child axis identifies a local type, also the children of that type are identified. Node selectors, presented in what follows, allow one to distinguish among the identified nodes.

Example 1. The following table reports, for different context nodes and axes, the identified nodes. To uniquely identify each node, the node name is superscripted by its identifier.

<table>
<thead>
<tr>
<th>Context node</th>
<th>Identified nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>library'</td>
<td>LL::self library'</td>
</tr>
<tr>
<td>sequence'</td>
<td>LL::child choice'5 title', author', character'5</td>
</tr>
<tr>
<td>library'</td>
<td>LL::child anonymousT, book'5</td>
</tr>
<tr>
<td>Isbn'</td>
<td>LL::parent choice'5</td>
</tr>
<tr>
<td>Isbn'</td>
<td>LL::parent anonymousT</td>
</tr>
<tr>
<td>authorType'</td>
<td>LL::base-type personType'</td>
</tr>
<tr>
<td>personType'</td>
<td>LL::derived-type authorType'</td>
</tr>
</tbody>
</table>

Node selector. The node selector filters some nodes among those identified by an axis. Selection criteria can be the node type and name. For each node type, a construct with the same name is available to select all the corresponding nodes. The node() construct allows the selection of all nodes regardless of their type. Each one of these constructs takes an optional QName parameter that further refines the selection, identifying only the nodes with the specified name property value. Local type declarations can be selected using the type() node selector. The name, eventually specified for nodes whose type is operator, can only be sequence, choice, or all, whereas for nodes whose type is annotation, it can only be documentation or appinfo.

Example 2. Starting from the given context nodes, the node selector allows to filter the nodes as follows:

<table>
<thead>
<tr>
<th>Context nodes</th>
<th>Node selector</th>
<th>Identified nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>choice'5 title', attrG'4</td>
<td>node()</td>
<td>choice'5 title', attrG'4</td>
</tr>
<tr>
<td>choice'5 title', attrG'4</td>
<td>element(title) title'</td>
<td></td>
</tr>
<tr>
<td>choice'5 title', attrG'4</td>
<td>operator(choice) choice'5</td>
<td></td>
</tr>
</tbody>
</table>

Predicates. A Boolean combination of conditions enclosed by square brackets can be employed to further refine the sequence of nodes identified by a step. XSPaht defines several Boolean conditions, some of which are inherited from XPath, whose BNF is reported in Fig. 3. The following conditions are specifically tailored to XSPaht:

- **TypeCond.** The type is condition is true when the context node is an element or attribute declaration which type is the specified one, or refers to a global declaration with the specified type. Specific conditions are introduced for simple/complex/anonymous types. When the context node is a type T and a path between T and a type T' exists in the type hierarchy, the typeDerivedFrom(T) condition is true. (The opposite for typeBaseOf).
- **ContentCond, TypeDefCond, and UniqueCond.** These conditions are used to express different constraints on the elements and attributes characteristics. The ContentCond condition is used to specify constraints on the allowed content of an element (empty, only text, only elements or mixed content), the TypeDefCond condition is used to specify constraints on
the type definition itself, whereas the UniqueCond condition is used to specify constraints on the elements uniqueness, keys, and key references.

- PropertyCond and RestrictionCond. The PropertyCond condition is true if the context node has the specified property (e.g., property("maxOccurs")), or has the specified property and its value meets the comparison criteria (e.g., property("maxOccurs") = 2). RestrictionCond is similar but checks simple-type restrictions.

- FullTextCond. These conditions allow to express full-text predicates to be evaluated on the schema components identified by the given XSPath expression.

Example 3. Starting from the given context nodes, the application of predicates filters the nodes as follows:

<table>
<thead>
<tr>
<th>Context nodes</th>
<th>Predicate</th>
<th>Ident. nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;17&gt;</td>
<td>exists(HL::child::operator())</td>
<td>sequence&lt;17&gt;</td>
</tr>
<tr>
<td>attribute()</td>
<td>operator()</td>
<td>attribute QName</td>
</tr>
<tr>
<td>attribute(QName)</td>
<td>operator(OperatorName)</td>
<td>type()</td>
</tr>
<tr>
<td>type(QName)</td>
<td>type(QName)</td>
<td>element()</td>
</tr>
<tr>
<td>element(QName)</td>
<td>element(QName)</td>
<td>HL::self::node()</td>
</tr>
<tr>
<td>/HL::parent::node()</td>
<td>/HL::parent::node()</td>
<td>/HL::descendant-or-self::node()</td>
</tr>
<tr>
<td>/HL::descendant-or-self::node()</td>
<td>/HL::descendant-or-self::node()</td>
<td>[position()=IndexValue]</td>
</tr>
</tbody>
</table>

4.2 Abbreviated Syntax and Examples of XSPath Expressions

More concise path expressions can be specified using an abbreviated syntax based on the shortcuts in Table 1. The most common name and type tests, such as attribute, operator, type, element tests, with or without additional name test, have shorter alternatives as well.

Fig. 4 presents some examples of navigational expressions. For each query, the figure reports the corresponding extended and abbreviated XSPath expressions and the nodes identified in the schema. Query 1 navigates through elements starting from the root node to reach a node representing an attribute (denoted through the $ symbol in the abbreviated syntax). The types of the library and book nodes have been implicitly navigated. Query 2 shows how the elements specified by the type authorType can be retrieved. Query 3 shows a path navigating the descendant-or-self axis (using // in the abbreviated syntax) and identifying all nodes representing elements (* means any element). A condition is imposed for selecting nodes whose type is personType. Query 4 shows an expression used to find types. Specifically, the filtering condition selects nodes having a pattern restriction property. The previous expressions work on the high-level representation. Query 5, by contrast, is an example of expression working at both levels. Indeed, a high-level step is employed to identify element book starting from the context node. Then, through a low-level step a sequence operator is reached. If evaluated from the library element the sequence node is identified. This query illustrates another possible abbreviation: when // is used to separate two steps, it can be abbreviated as !. Query 6, finally, is an example of expression specifying a positional predicate. It selects the third high-level child element of book.

4.3 Semantics

Let $HS$ be the high-level representation and $LS$ be the low-level representation of a schema $S$. Given a graph $S$, let $V_S$ and $E_S$ denote the set of its nodes and edges, respectively.

In our two-level view of schemas, we extend the child and parent axes to facilitate the writing of expressions. Specifically, as shown in Example 1, the child axis, whenever evaluated on an element, identifies the top-level nodes specified in its type definition (both at high and low levels), along with all the element and type nodes on the shortest path between the context node and the identified nodes.

The meaning of an axis $l :: a$, with respect to a context node $x$, is given through function $R_l^a(x)$. $R_l^a(x)$ consists of the set of nodes identified by axis $a$ at level $l$ from $x$. $R_{sel}^a(x) = \{x\}$ and $R_{sel}^a(x) = \{x\}$ if $x \in V_{HS}$, $\emptyset$ otherwise. $R_{child}^a(x)$ is a quite intuitive extension of $child_{LS}(x)$ and corresponds to the following recursive definition:

$$R_{child}^a(x) = child_{LS}(x) \cup \{R_{child}^a(x) \mid x \in child_{LS}(x) \land nodetype(x) \in \{element, type\}\}$$

### TABLE 1 Abbreviated Syntax

<table>
<thead>
<tr>
<th>Long form</th>
<th>Short form</th>
</tr>
</thead>
<tbody>
<tr>
<td>HL::child:</td>
<td>!</td>
</tr>
<tr>
<td>LL::child:</td>
<td>$*$</td>
</tr>
<tr>
<td>attribute()</td>
<td>$@* Name $</td>
</tr>
<tr>
<td>type()</td>
<td>OperatorName</td>
</tr>
<tr>
<td>element()</td>
<td>QName $</td>
</tr>
<tr>
<td>element(QName)</td>
<td>QName $</td>
</tr>
<tr>
<td>/HL::parent::node()</td>
<td>/HL::parent::node()</td>
</tr>
<tr>
<td>/HL::descendant-or-self::node()</td>
<td>/HL::descendant-or-self::node()</td>
</tr>
<tr>
<td>[position()=IndexValue]</td>
<td>[IndexValue]</td>
</tr>
<tr>
<td>[exists(SinglePath)]</td>
<td>[SinglePath]</td>
</tr>
</tbody>
</table>

Fig. 4. Examples of XSPath expressions.
if \( x \) is an element, \( R_{child}^{HL}(x) = child_{LS}(x) \) otherwise. At low level, the parent axis identifies the parents of a node in the low-level graph, that is, \( R_{parent}^{LL}(x) = parent_{LS}(x) \). The high-level child and parent axes are defined also for nodes not belonging to the high-level graph. Intuitively, the high-level parents/children of a node \( x \) are the first encountered low-level ancestors/descendants of \( x \) belonging to the high-level graph. Specifically:

\[
R_{child}^{HL}(x) = R_{child}^{LL}(x) \cap V_{HS} \cup \bigcup_{y \in R_{child}^{NL}(x) \cap V_{HS}} R_{child}^{HL}(y);
\]

\[
R_{parent}^{HL}(x) = R_{parent}^{LL}(x) \cap V_{HS} \cup \bigcup_{y \in R_{parent}^{NL}(x) \cap V_{HS}} R_{parent}^{HL}(y).
\]

The descendant and ancestor axis at high level are the transitive closure of the child and parent axis, specifically:

\[
R_{descendant}^{HL}(x) = R_{child}^{HL}(x) \cup R_{parent}^{HL}(x);
\]

\[
R_{ancestor}^{HL}(x) = R_{child}^{HL}(x) \cup R_{parent}^{HL}(x).
\]

In what follows, given a schema \( S \) and a node \( x \in S \), \( local_{LS}(x) \) denotes the union of the sets of nodes of the subgraphs of \( S \) containing \( x \). Referring to Example 4, \( local_{LS}(21) \) is the set of nodes from 15 to 28. Starting from the definition of \( local \), we define the meaning of the descendant and ancestor axis at low level:

\[
R_{descendant}^{LL}(x) = R_{child}^{LL}(x) \cap local_{LS}(x)
\]

and

\[
R_{ancestor}^{LL}(x) = R_{parent}^{LL}(x) \cap local_{LS}(x).
\]

Function \( D \), extension of the one proposed in [44], determines the axis direction, that is, \( backward \) for ancestor, \( ancestor-or-self \), and \( base-type, forward \) otherwise.

The semantics of a path \( p \) evaluated on a context node \( x \) of a schema \( S \) is given through function \( [p]_{NS}(x) : Node \rightarrow NS \) in Fig. 5, where \( NS \) denotes the power set of \( S \). This function relies on the auxiliary functions \( [p]_{local}(x, k, n) : Node \rightarrow Int \rightarrow Int \rightarrow Bool \) and \( [p]_{local}(x, k, n) : Node \rightarrow Int \rightarrow Int \rightarrow Int \) defined in Fig. 5 and those defined in Table 2. We remark that Fig. 5 reports an excerpt of the function definition for simple paths and part of the predicates.

### 5 Type Analysis

In this section, we first present a type system for XSPF that determines, given an XPath expression \( xe \), the types of the nodes that can be identified by evaluating \( xe \). We then discuss another form of type analysis, the extended type analysis, which determines, for each step \( si \) in \( xe \), the types of the nodes, among those on which \( si \) is evaluated, that may contribute to the results of the evaluation of \( xe \).

The set \( NT \) of node types previously introduced is here refined to take into account whether a declaration is...
local/global or simple/complex. We remark that the type of an expression (or expression component) is a set of node types.

Definition 1 (Node Types). Let $G$ and $L$ denote global and local, $C$ and $S$ denote complex and simple, $E$, $A$, $T$, and $Z$ denote element, attribute, type, and group, and $OP$, $ROOT$, and $ANN$ denote an operator, a schema root and an annotation. The set $NT = \{GSE, GCE, GST, GCT, GA, GAZ, GEZ, LSE, LCE, LST, LCT, LA, LAZ, LEZ, OP, ROOT, ANN\}$ denotes the types of the nodes in an XML schema. Each schema node has a single type.

Example 5. The nodes library and id of the schema in Fig. 2 have type $GCE$ and $LA$, respectively.

5.1 XSLPath Type System

Fig. 6 presents the set of XSLPath typing rules, which determine the types of the nodes that can be identified by an XSLPath expression. The rules rely on a context type $T$ that denotes the types of the nodes on which an expression (or expression component) can be evaluated. The type of the first step of an absolute XSLPath expression is determined with respect to the context $\{\text{ROOT}\}$ ($NT$ for relative expressions).

In the rules, functions $T_{\text{ns}}(\text{ns})$ and $T_{\text{cond}}(\text{cond})$ are used to determine the type of the nodes that can satisfy a node selector $\text{ns}$ or a condition $\text{cond}$, respectively. The family of functions $T_{\text{pl}}(t)$, instead, determines the type of nodes that can be identified evaluating an axis $l :: a$ starting from a node of type $t$. Table 3 reports the definition of function $T_{\text{HL}}$, while the definitions of functions $T_{\text{ns}}$ and $T_{\text{cond}}$ are reported in Tables 4 and 5, respectively. In the last table, a smaller set of types can be determined for the cases superscripted with an asterisk depending on condition arguments.

Example 6. Consider the context $T = \{\text{ROOT}\}$ and the step $\text{HL}::\text{child}::\text{element()}$. Function $T_{\text{HL}}$ determines that only nodes whose type is $\{\text{GSE, GCE, GST, GCT, GA}\}$ can be identified. Starting from nodes of one these types, the node selector $\text{element()}$ can only be satisfied by $\{\text{GSE, GCE}\}$. Thus, the type of the step is $\{\text{GSE, GCE}\}$ denoting that the step can identify only global elements. Consider now the step $\text{HL}::\text{child}::\text{type()}[\text{anonymousType()} \& \exists\text{HL}::\text{child}::\text{element()}]$. Function $T_{\text{HL}}$ determines that only nodes whose type is $\{\text{GSE, GCE, GST, GCT}\}$ can be identified. Starting from nodes of one these types, the node selector $\text{element()}$ can only be satisfied by $\{\text{GSE, GCE}\}$. Thus, the type of the step is $\{\text{GSE, GCE}\}$ denoting that the step can identify only global elements.
child::element(book)], evaluated from the context \( T = \{ \text{GSE}, \text{GCE} \} \). Functions \( T_{\text{HL}} \) and \( T_{\text{us}} \) determine that only nodes whose type is GST, GCT, LST or LCT can be identified by \( \text{HL}::\text{child}::\text{type()} \). Among them, according to \( T_{\text{us}} \), only local complex types can satisfy both predicates. Thus, the type of the step is \( \{ \text{LCT} \} \).

In typing an XSPath expression \( s_1/\ldots/s_n \) the type of each step is determined starting from the types of the nodes that can be identified by the previous step.

**Example 7.** Consider the following two steps expression \( xe: /\text{HL}::\text{child}::\text{element()} /\text{HL}::\text{child}::\text{type()} [\text{anonymousType()}] \text{ and exists}(\text{HL}::\text{child}::\text{element(book))} \). Since \( xe \) is absolute, the first step is evaluated from nodes of type \( \{\text{ROOT}\} \). The type of the first step is \( T_1 = \{ \text{GCE}, \text{GSE} \} \). The second step is evaluated from nodes whose type belongs to \( T_1 \) and its type is \( T_2 = \{ \text{LCT} \} \). The type of \( xe \) is, thus, \( \{ \text{LCT} \} \).

### 5.2 Extended Type Analysis

The type analysis is helpful in determining the types of the nodes that can be identified by an XPath expression \( xe \) or by its steps. However, it does not determine the types of the nodes, among those on which the expression (or a step) can be evaluated, that may contribute to the result of the evaluation of \( xe \). This is the goal of the extended type analysis that relies on the concept of *relevant context types*.

**Example 8.** Consider the expression of Example 7. The analysis determines that the first step may identify complex or simple types. However, only complex types may contribute to the final result, as simple types are not structured.

To present the extended type analysis, the concept of *relevant context types* of a step \( s_i \) and the family of functions \( T^i_s(t) \) need to be introduced.

**Definition 2 (Relevant Context Types).** Let \( s_1/\ldots/s_i/\ldots/s_n \) be an XSPath expression and let \( T \) be the types of the nodes on which \( s_i \) can be evaluated, that is, the typing context of \( s_i \). The relevant context types of \( s_i \) are those types \( t \in T \) which satisfy: \( \{t\} \vdash s_i/\ldots/s_n : T' \) with \( T' \neq \emptyset \).

The family of functions \( T^i_s(t) \) determines, given an axis \( l : a \) and a node type \( t \), the types of the nodes starting from which nodes of type \( t \) can be reached following the axis \( l : a \), that is \( T^i_s(t) = \{t'|t' \in NT \land t \in T^i_s(t') \} \).

**Example 9.** Let \( T_1 = \{ \text{GCE}, \text{GSE} \} \) and \( T_2 = \{ \text{LCT} \} \) be two sets of node types. \( T^\text{HL}_{\text{child}}(T_1) \) is \( \{ \text{LSE}, \text{LCE}, \text{ROOT} \} \), that is, global elements may be identified by the \( \text{HL}::\text{child}::\text{axis} \) only when it is evaluated on a local element or the schema root. \( T^\text{HL}_{\text{child}}(T_2) \) is \( \{ \text{GCE}, \text{LCE} \} \).

The extended type analysis is performed as follows: Let \( xe = s_1/\ldots/s_n \) be an XPath expression, \( T_0 \) is the context from which \( s_1 \) is evaluated, and \( T_1, \ldots, T_n \) be the types of its steps. The extended type analysis determines an additional set of types \( C_1, \ldots, C_n \), where \( C_i \subseteq T_{i-1} \) is the relevant context type of \( s_i \), \( 1 \leq i \leq n \). The relevant context type of step \( s_i \) is \( l_i : a_i : n_{s_i}[r_i] \) is \( C_i = T_{i-1} \cap (\cup_{t \in T^i_s(t)} T^i_s(t)) \).

**Example 10.** In the expression \( xe \) of Example 7, \( T_0 = \{ \text{ROOT} \} \), \( T_1 = \{ \text{GCE}, \text{GSE} \} \), and \( T_2 = \{ \text{LCT} \} \). The extended type analysis of \( xe \) determines that \( C_1 = \{ \text{ROOT} \} \) and \( C_2 = \{ \text{GCE} \} \). That is, among the nodes on which the first (or second) step is evaluated, only those of type \( \text{ROOT} \) (or \( \text{GCE} \)) may contribute to the result of the evaluation of \( xe \).

### 6 Translation in Standard Languages

In this section, we provide two approaches for the translation of an XSPath expression \( xe \) into a standard language for XML (XPath/XQuery). This translation allows us to exploit standard query engines (with their optimization strategies) for the evaluation of \( xe \). The key requirement of the proposed translation approaches is that the semantics of the original expression is preserved in the translated expression.

In the first approach, named *schema-dependent*, the translation of \( xe \) into an XPath expression \( e \) takes into account a specific schema \( S \) on which \( e \) will be evaluated. The resulting translation can be easily applied to replicated schemas contained in other servers holding standard XPath engines and the expression \( e \) is extremely simple. However, the semantics of \( e \) is equivalent to the semantics of \( xe \) only on schema \( S \). The evaluation of \( e \) on another schema is not guaranteed to preserve the semantics of the original expression.

In the latter approach, named *schema-independent*, the translation of \( xe \) produces an XQuery expression that can be evaluated on an arbitrary schema. This translation can be useful in different contexts, like, for example, querying collections of schemas, checking structural properties of different versions of the same schema, or simply to avoid (re-)translating the same expression every time the target schema is updated.

#### 6.1 Schema-Dependent Translation

The translation of an XSPath expression \( xe \) into a simple XPath expression to be evaluated on a schema \( S \) is quite easy. The basic translation algorithm \( T_p(xe, S, x) \) consists of the following two activities: 1) \( xe \) is evaluated on \( S \) starting from \( x \) according to the semantics defined in Fig. 5, identifying a set of nodes \( N, x \), 2) for each node \( n \in N \), an absolute XPath expression identifying \( n \) can be generated to be applied on the textual representation of the schema. The union of the obtained XPath expressions identifies the nodes in the set \( N \). This translation can be further compacted by avoiding to specify the traversal of the same node multiple times.

**Example 11.** Let \( \text{*/[typeIs("authorType") or type-Is("personType")]} \) be the expression identifying all elements whose type derives from personType. When evaluated on the schema in Fig. 1, it identifies the author and character element declarations in book. The XPath expressions \( a = \alpha/\text{element[@name="author"]} \) and \( b = \alpha/\text{element[@name="character"]} \) can be specified, where \( \alpha \) denotes \(/\text{schema/element[@name="book"]}/\text{complexType/sequence} \). The schema-dependent translation is, thus, the XPath expression \( a \mid b \). The same nodes can be identified by the more compact XPath expression \( \alpha/\text{element[@name="author"]} \mid \alpha/\text{element[@name="character"]} \).
takes as input the relevant context types $C$ selectors, and predicates are evaluated at translation time.

The obtained translation can only be applied on the schema $S$ (or one of its replica) because axes, node selectors, and its predicates are evaluated at translation time. Its evaluation on a different schema $S'$, may identify incorrect nodes.

The following correctness and complexity results for the schema-independent translation process hold.

**Proposition 1 (Correctness of $T_B$).** Let $S$ be a schema, $xe$ be an XQuery expression, $x$ be a context node, $e = T_B(xe, S, x)$ the result of the translation, and Eval be a XPath evaluation algorithm. Then, $\langle xe \rangle_{NS}(x) = \text{Eval}(e, S, x)$.

**Proposition 2 (Complexity of $T_B$).** Let $S$ be a schema, $xe$ be an XQuery expression, $x$ be the number of step components in $xe$, and $n$ the number of nodes in $S$. The complexity of $T_B$ applied on $xe$ and $S$ is the same of evaluating an XPath expression with $c$ components on $S$. Specifically, according to [18], the complexity is $O(n^3c^2)$, if positional predicates are employed, $O(nc)$ otherwise.

### 6.2 Schema-Independent Translation

The schema-independent translation of an expression $xe = s_1/.../s_n$ is realized through function $T_P$. This function takes as input the relevant context types $C_1,...,C_n$ of the steps determined by the extended type analysis of $xe$, and the type $T_e$ (denoted $C_{n+1}$). The translation of $xe$ is obtained through the juxtaposition of the translation of each step within $xe$. Since the step translation can contain invocation of XQuery functions, the bodies of the functions used in the translated expression must be integrated in the expression.

The devised translation approach is both correct and efficient as stated by the following propositions.

**Proposition 3 (Correctness of $T_P$).** Let $xe$ be an XPath expression, $T$ a set of node types, and $e = T_P[xe]T$ the generated XQuery expression. For each schema $S$ and any node $x$ in $S$, $\text{nodetype}(x) \in T \Rightarrow \langle xe \rangle_{NS}(x) = \text{Eval}(e, S, x)$.

**Proposition 4 (Complexity of $T_P$).** Let $xe$ be an XPath expression and $c$ be the number of its components. The complexity of the translation of $xe$ is $O(c)$.

In the remainder of the section, we detail the translation of each single component of $s_i$, denoted $l_i :: a_i :: ns_i[r_i]$, because the translation of an entire step is a straightforward composition of the translation of its components. In the examples, we will consider the translation of the XQuery expression \(/\text{HL::child::element()}/\text{HL::child::type()}[\text{anonymousType()} \text{ and exists(\text{HL::child::element(book)})}]\) of Example 7.

**Translation of an axis.** The axis translation determines a short XQuery expression $e_i$ that starts from nodes whose type belongs to $C_i$, identifies the nodes whose type belongs to $C_{i+1}$, which can be reached following the axis $l_i :: a_i$.

A complex XQuery expression can be specified that fully captures the behavior of each axis $l_i :: a_i$, independently from the type of the context node from which it will be evaluated. However, because $C_i$ and $C_{i+1}$ are known, shorter and more efficient translations can be realized such that: 1) no node that $l_i :: a_i$ would not identify from $C_i$ is identified, 2) each node which type is in $C_{i+1}$ that $l_i :: a_i$ would identify is identified. The milestone for identifying XQuery expressions with these properties is that the translation of $l_i :: a_i$ is composed of different parts (named blocks) that can be combined together relying on a common pattern, so that the expression has the same semantics of $l_i :: a_i$. Each block is devoted to identifying nodes of specific types starting from nodes of specific types. For instance an XQuery expression that translates the $\text{HL::child}$ axis may contain a block that identifies referred global elements when it is evaluated from an element declaration. When translating $l_i :: a_i$ with respect to $C_i$ and $C_{i+1}$, some blocks can be omitted still ensuring that the two translation properties are satisfied and often the pattern can be simplified leading to a shorter expression.

Let $B$ be a place-holder representing blocks, we express patterns through the following grammar:


**Example 12.** Consider the $\text{HL::child}$ axis. The pattern $(\text{GRREF}\{D_1\})/(\text{EREF} \ AREF) D_2)/(\text{TREF} D_2)/(\text{LCHILD})\ CHILDD_2 | D_1) \text{ and } \text{CHILD}_1$ has been identified, where the blocks are reported in Fig. 8 with the associated XQuery fragment.

For each axis $l_i :: a_i$, different patterns and blocks have been identified, and translation tables, like the one in Fig. 7 for $\text{HL::child}$, have been developed. Each row of a translation table reports the blocks that are required in the
translated expression to identify the nodes in a specific relationship with the context node.

The translation process consists of three activities: 1) by considering \( C_i \) and \( C_{i+1} \), the blocks which need to be present in the translation expression are identified; 2) blocks are instantiated in the pattern, pointing out blocks that do not occur (denoted \( \lambda \)); 3) the pattern is simplified by applying some rewriting rules that remove the missing blocks.

In activity 1, the blocks to be instantiated are selected as follows: we check, row by row in the translation table, whether both \( C_i \) and \( C_{i+1} \) have nonempty intersection with origin and target. If so, the corresponding blocks are collected.

Example 13. Consider \( s_1 = \text{HL}::\text{child}::\text{element}() \), \( C_1 = \{\text{ROOT}\} \), and \( C_2 = \{\text{GCE}\} \) of Example 7. In activity 1, the following set of block is identified: \( \{\text{CHILD}_1\} \). Consider now a step \( s_2 = \text{HL}::\text{child}::\text{type}() \), \( C_2 = \{\text{GCE}\} \), and \( C_3 = \{\text{LCT}\} \). The set \( \{\text{CHILD}_2\} \) is identified.

In activity 2, the set of blocks \( Q = \{B_1,\ldots,B_t\} \) identified in activity 1 are instantiated in the pattern with the corresponding XQuery expression while the others are replaced with \( \lambda \). In activity 3, the instantiated pattern can be simplified through the following rewriting rules that allow the removal of \( \lambda \) symbols:

\[
\begin{align*}
B|\lambda \to B & \quad \lambda|B \to B \quad \lambda|\lambda \to \lambda \\
B/\lambda \to B & \quad \lambda/B \to B \quad \lambda/\lambda \to \lambda \\
(B) \to B & \quad (\lambda) \to B \quad B/D_a \to B \quad D_a/B \to B.
\end{align*}
\]

Example 14. Consider the block sets \( \{\text{CHILD}_1\} \) of Example 13. Its instantiation in the pattern is: \( (\lambda | \lambda)/(\lambda | \lambda | \lambda)/ (\lambda | \lambda)/(\lambda | \lambda | \lambda) \) \( \text{CHILD}_1 \). By applying the rewriting rules, we simply obtain \( \text{CHILD}_1 \).

The so obtained instantiated pattern identifies all the nodes of one of the types in \( C_{i+1} \) reachable through \( l_i::a_i \) from \( C_i \). Note that additional nodes might be identified as well; however, the extra nodes will not affect the XPath expression translation semantics. Moreover, when all \( B_i \) blocks are present, also the context node could be identified by the translation. In this case, an additional condition is added to the translation.

Translation of a node selector. Function \( NS \) in Fig. 9 realizes the translation of the node selector \( ns_l \), at level \( l_l \) by considering the relevant context type \( C_i \).

Example 15. Consider the node selector \( \text{element}() \) and the type \( \{\text{GCE}\} \). The translated expression is \( \{\text{self::element}\}. \) Consider now the node selector \( \text{type}() \) and the type \( \{\text{LCT}\} \). The translation is \( \{\text{self::complexType}\}. \)

Translation of a predicate. The translation of a predicate \( r \) is handled as follows: The basic conditions contained in \( r \) are substituted with conditions to be evaluated on the textual representation of the schema. This translation is realized through function \( C \) in Fig. 5. Function \( C \), given a basic condition \( c \), the direction of the step axis \( d \), and the relevant context type \( C_i \), returns the translation of \( c \).

Example 16. The translation with respect to the type \( \{\text{LCT}\} \) of the predicate \( \text{anonymousType()} \) and \( \exists\text{HL::element(book)}(\text{book}::\text{element}()) \) is

\[
\{\text{not}(@\text{name}) \text{and exists(local}: \text{ETD}(/\text{schema}.))@\text{name}/\text{C3}@\text{name}/\text{C2}@\text{name}/\text{C1}\} @\text{name} /\text{C1} @\text{name} /\text{C1}/@\text{name} [\text{not}(@\text{name}) \text{and exists(local}: \text{ETD}(/\text{schema}.))@\text{name}/\text{C3}@\text{name}/\text{C2}@\text{name}/\text{C1}\} @\text{name} /\text{C1} @\text{name} /\text{C1}/@\text{name}].
\]

We are now ready to present the entire translation process for the XPath expression \( xe \) of Example 7. \( xe \) is composed of two steps \( s_1/s_2 \). Since \( xe \) is absolute, its translation requires the presence of \( /\text{schema}. \). The type analysis has determined that the relevant context types of \( s_1 \) and \( s_2 \) are \( C_1 = \{\text{ROOT}\} \) and \( C_2 = \{\text{GCE}\} \) and the type of the entire expression is \( T_2 = \{\text{LCT}\} \). By means of the translation table in Fig. 7 the axis of both steps is translated as *.

The node selector (as discussed in Example 15) becomes \( \{\text{self::element}\} \) and \( \{\text{self::complexType}\} \), respectively. The translation of the condition of \( s_2 \) consists in the conjunction of the translation of the basic conditions within specified. The translation of anonymousType() is \( \text{not}(@\text{name}) \) because the relevant context type is \( \{\text{GCE}\} \). The translation of an \( \exists\text{HL::element(book)}(\text{book}::\text{element}()) \) requires to translate the XPath within contained. The translation of the condition is the one obtained in Example 16. The complete translated expression is, thus,

\[
/\text{schema}/*/\text{self::element}/>/*\text{self::complexType}
/\text{not}(@\text{name}) \text{and exists(local}: \text{ETD}(/\text{schema}.))@\text{name} /\text{C3}@\text{name} /\text{C2}@\text{name} /\text{C1}@\text{name} /\text{C1}/@\text{name} [\text{not}(@\text{name}) \text{and exists(local}: \text{ETD}(/\text{schema}.))@\text{name} /\text{C3}@\text{name} /\text{C2}@\text{name} /\text{C1}@\text{name} /\text{C1}/@\text{name}].
\]

7 Prototype and Experiments

E\text{X}up [9] is a Java application developed for supporting users in updating XML schemas, which includes the translation and evaluation of XPath expressions. E\text{X}up fully implements the algorithms presented in this paper and contains other features that, for space constraints, we have not discussed. Expressions involving full-text predicates, however, cannot be evaluated by E\text{X}up due to license.
limitations in the employed XQuery evaluation library. Other important features include syntactic or semantic error reporting and visual analysis of the queries executed on a schema or its associated documents.

To assess the usefulness of the proposed XSPath language, both its usability and the efficiency of the translation process have been extensively evaluated on top of the Exup system. The results of this experimental evaluation are summarized in the following sections. The experimental materials, along with the Exup system, can be found at the URL http://felix.disi.unige.it/exup.

7.1 Language Usability

To evaluate the usability of the language, we downloaded from the web two XML schemas (one about cooking recipes and one about online auctions), translated them in Italian and adapted them to a one page size to be easily interpretable by nonexpert users. Moreover, we specified an instance document for each schema and provided nine tasks on both schemas and associated documents that can be executed in XPath and XSPath (we remark that XSPath can also be used to retrieve information from the instance documents of a schema).

Relying on such material, we developed four questionnaires differing in the order in which the tasks on XP and XSPath are asked to be completed, and on the schema on which they need to be executed. Moreover, a brief introduction to XSPath and to the abstraction levels have been included (with some sample XSPath expressions).

The questionnaires have been proposed to 20 users equally distributed in the four kinds (users are students that attended courses on XML and researchers that are knowledgeable of XPath). To each user, we asked her level of knowledge (low, average or high) of XML, XPath, and XML schema. Fig. 10a reports the histogram of their answers. Almost all the users (19/20) have no prior knowledge of our work on XSPath. The average time employed by the users to learn XSPath is 23 minutes (median 20, minimum 5, maximum 40).

Our experiment has two dependent variables, on which treatments are compared: 1) correctness and 2) time required to perform the assigned tasks. The correctness of each task was assessed by two of the authors evaluating the response to the task and giving a score from 0 (task not executed) to 3 (completely correct) to each task. Finally, the total correctness for the tasks in each language was computed summing up the nine scores. Thus, variable correctness ranges from 0 (no correct tasks) to 27 (nine totally correct tasks). Time was measured by asking users to take note of the number of minutes required to complete each single task. Finally, variable time is computed by summing up the time required to complete the nine tasks for each language. Thus, we can state the null hypotheses for the study as follows:

- $H_{C_l}$: correctness(XPath)=correctness(XSPath)
- $H_{T_l}$: time(XPath)= time(XSPath).

Since we could not find a clear empirical evidence that points out a clear advantage of one approach versus the other, we formulate $H_{C_l}$ and $H_{T_l}$ as two-tailed hypotheses. Because of the sample size, we adopted the paired Wilcoxon test to check the two null hypotheses.

Fig. 10b summarizes the distribution of correctness by means of boxplots for both experiments. Observations are grouped by language (XPath versus XSPath). The y-axis represents the cumulative correctness of the nine tasks: score $= 27$ represents the maximum value of correctness and corresponds to nine (completely) correct tasks. The boxplots show that the users achieved a better correctness level when accomplishing the tasks with XSPath. By applying the Wilcoxon test, we found that the difference in terms of correctness is statistically significant (p-value = 0.005). Therefore, we can reject the null hypothesis $H_{C_l}$. Moreover, Fig. 10d reports the boxplots for the XPath correctness experiments splitting the users according to their level of knowledge of XPath. These boxplots allow us to show the distribution of correctness of the tasks on XPath and on XSPath according to the XPath expertise of the users. As the graphics show, users with low and average levels of knowledge on XPath had many problems in writing XPath expressions. By contrast, in writing XSPath expression, there is no significant difference among the medians of the three categories of users.

The second variable can be easily tested by looking at the time required to complete all the nine tasks. Whenever a user was not able to complete a task (and therefore, did not report the time), we considered the maximal time that other users devoted to complete the same task. Fig. 10c summarizes the distribution of the time variable by means of boxplots. The boxplots show that users using XSPath usually employ less time to complete the same task. We used the Wilcoxon test to check the second null hypothesis by executing a paired analysis. In this case, even if the statistical significance is questionable (p-value = 0.13), we remark that the absolute difference between medians is quite large.
Table 6 reports the more significant questions posed to the users after the tasks and the percentage of positive answers.

### 7.2 Performance

To evaluate and contrast the different XSPath translation processes, we adapted nine schemas found on the web, including all supported XML schema features and 153 XSPath expressions covering all language features. These schemas are available on the eXup website and are of small size, from 2 to 10 KB, whereas the corresponding graph diameter ranges from 6 to 19 (mean 11.78). We remark that we also successfully tested the E\textsuperscript{X}up system with much bigger schemas several hundred kilobytes large. We measured the time required for translating/evaluating each expression and the length of the generated expression. We considered the $T_P$ and $T_B$ algorithms discussed in Section 6. For the last algorithm, we considered two variants: $T_P$, which relies on the extended type analysis, and $T_N$, which does not perform any type analysis ($NT$ is used as type of each step). For both variants, we also investigated whether a schema-aware XQuery evaluation, denoted $T_{SP}$, $T_{SN}$, and $T_{SB}$, respectively, has a beneficial effect on the translated expression evaluation time.

Our test machine uses a Core i7 processor and runs a 64-bit Oracle Java 1.7 JVM under a 64-bit Linux distribution. Generated XPath and XQuery expressions are evaluated using the Saxon EE 9.3 library. Each measure has been repeated a thousand times averaging the results. Fig. 11a reports the time required for translating the considered expressions, while Fig. 11b their length. Finally, Fig. 11c contrasts the time required for the evaluation of the translated expressions.

For what concerns the translation process, the $T_P$ algorithm required less than 0.1 ms in all tests. The $T_N$ algorithm required about double the time than $T_P$, even if no type analysis is performed, because the generated expression is often much longer than the one produced by $T_P$. Finally, the schema-dependent translation identified by means of $T_B$ is much slower (about 50 times) than $T_P$, as it requires the evaluation of the expression to translate on the schema.

For what concerns the translated expression length, the benefit of performing the type analysis is significant and on the considered queries, on average, reduces the expression length by 65 percent with respect to $T_N$. The length of the schema-independent translation is affected by the complexity of the expression to translate and by the used steps. The schema-dependent translations, instead, are mainly affected by the number of identified nodes, as they need to identify each node independently. While in general schema-dependent translations are very short (92 percent shorter than $T_N$ translations), when a simple expression (such as `//*`) identifies a large number of nodes, $T_B$ translations can be much longer than $T_N$ ones.

For what concerns, the evaluation of the translated expressions, the time required is mainly affected by the length of the expression and its complexity and only secondarily by the schema size or the number of nodes identified. The expressions produced by $T_B$ are often very simple and short and, not surprising, the fastest. The expressions produced by $T_P$ and $T_N$, by contrast, are longer and more complex. If no type analysis is performed, they are on average 20 times slower. Analysis, usually, greatly reduces the number of nodes to be processed, the expression length and its complexity. Indeed, expressions produced by $T_N$ are only five time slower than $T_B$ translations and their evaluation, for the considered expressions, requires no more than 10 ms. Schema-aware XQuery processing, even ignoring the time required for processing the normative XML schema definition [40] has, on average, a negative effect on evaluation time. This is due to the small size of the schema on which they are evaluated, the more expensive casts required and the complexity of XML schema. Indeed, on average, schema-aware XQuery processing slows down $T_N$, $T_P$, and $T_B$ expression evaluation by 7, 3, and 27 percent, respectively.

### 8 Conclusions and Future Work

In this paper, we have proposed a navigational XML schema query language. The language derives from XPath and is characterized by a seamless switching between abstraction levels in navigating the schemas, where the low level exposes the operators in the definition of complex type structures while the high level masks them. The language...
has been defined, by specifying its syntax and semantics, and a translation process has been proposed to evaluate expressions in the language through existing XPath/XQuery engines. Specifically, when the navigational expression is to be evaluated on arbitrary schemas, the translation process exploits static type analysis techniques to produce a shorter expression. An experimental evaluation demonstrated the usability of the language and its practical applicability, together with the improvements brought by the static analyses in the translation process.

We are extending the work presented in the paper in different directions. First, we are enhancing the XSPaTh language to cover the XML schema features that are currently unsupported. Most notably, the support for queries on composition of schemas is under investigation, because very large schemas, with references to and obtained by including other schemas, represent targets on which a query language can be very useful. We also wish to enrich the XSPaTh language with additional comparison predicates among types, allowing, for instance, to require that two types have overlapping or disjoint sets of allowed instances, as in [11]. As a second direction, approximate schema matching would be interesting in many of the application scenarios we devise for the language. Thus, as approximate retrieval for XML documents has been widely investigated [2], [21], approximate evaluation (both in terms of names, types and structures) of schema queries can be proposed. The current version of XSPaTh, as discussed in the paper, support full-text predicates on textual schema components, but other approximations could be useful as well. Such approximation would also allow the language to be employed on not completely correct schemas without requiring a preprocessing for producing a correct schema before querying it.

REFERENCES

Federico Cavalieri received the BS and MS degrees in computer science magna cum laude, in 2008 and 2010, respectively. He is working toward the PhD degree in the Department of Computer and Information Sciences, University of Genoa, Italy. He was an intern at FLOWR Foundation, Palo Alto, California, and at 28msec, Zurich, Switzerland. His research interests are focused in the area of databases and information systems and specifically on XML schema evolution and XML query and update processing. He is a member of the IEEE.

Giovanna Guerrini received the Laurea degree (cum laude) in information science in 1993 and the PhD degree in computer science in 1998, both from the University of Genova, Italy, where she has been an associate professor since 2005. Previously, she was an associate professor at the University of Pisa (2001-2005) and an assistant professor at the University of Genova (1996-2001). Her research activity has recently focused on semistructured and XML data handling, with specific interests in issues concerning data heterogeneity, approximate retrieval, problems in heterogeneous data sources integration and evolution. He has been a coeditor of five proceedings of international workshops published by Springer and ACM. He co-organized several international and national events: SEBD 2009, the DataX workshop at EDBT’04, EDBT’06, and EDBT’08, XOODB 2009, and the Updates and XML Workshop at EDBT/ICDT 2010. He is serving as a publicity chair at EDBT/ICDT 2013 and ADBIS 2013. He is a member of the IEEE.

Marco Mesiti received the Laurea degree (cum laude) in information science in 1998 and the PhD degree in computer science in 2003, both from the University of Genova, Italy. Since 2005, he has been an assistant professor at the University of Milano, Italy. In 1998, he has been a visiting researcher at the Applied Research Center of Telcordia Technologies, Morristown, New Jersey. His research activity has been carried out in the database area, and has focused on semistructured and XML data handling, with specific interests in access control models, issues concerning data heterogeneity, approximate retrieval, problems in heterogeneous data sources integration and evolution. He has been a coeditor of five proceedings of international workshops published by Springer and ACM. He co-organized several international and national events: SEBD 2009, the DataX workshop at EDBT’04, EDBT’06, and EDBT’08, XOODB 2009, and the Updates and XML Workshop at EDBT/ICDT 2010. He is serving as a publicity chair at EDBT/ICDT 2013 and ADBIS 2013. He is a member of the IEEE.