E$\mathcal{X}$up: An Engine for the Evolution of XML Schemas and Associated Documents

Federico Cavalieri
DISI - University of Genova
Via Dodecaneso, 35
Genova, Italy
cavalieri@disi.unige.it

ABSTRACT

XML Schema is employed for describing the type and structure of information contained in XML documents. *Schema evolution* means that a schema is modified and the effects of the modification on instances are faced. *XSUpdate* is a language that allows to easily identify parts of an XML Schema, apply a modification primitive on them and define an adaptation for associated documents. Purpose of this paper is to present the engine we developed for the evaluation of XSUpdate statements against XML Schemas and associated documents. The presented engine relies on the translation of XSUpdate statements in XQuery Update expressions.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages;
H.2.4 [**Database Management**]: Systems

1. INTRODUCTION

XML Schema [14] is widely adopted to describe the structure of XML data. Knowing the schema of documents facilitates their querying, retrieval and sharing. New requirements may arise in application domains that lead to update the structure of data. Moreover, the integration of heterogeneous sources may require to modify the schema. Following the terminology introduced in [9], updates to the schema information, besides modifying the schema, can lead to schema evolution. Schema evolution means that the original schema is replaced by an updated schema and the effects of the update on instances are faced. Specifically, we focus on efficient revalidation of documents and adaptation of documents to the modified schema.

Despite the high dynamicity of the contexts where XML documents are employed, XML updates have received less attention than XML queries and the W3C proposal for XML document updates, XQuery Update Facility [12], has appeared as a candidate recommendation only in June 2009. As surveyed in [7], updates on XML Schemas have received even less attention despite the great impact they may have in the database field. Even if commercial tools (e.g. Stylus Studio, XML Spy) have been developed for graphically designing XML Schema definitions, they do not support the specification of schema updates nor the semi-automatic adaptation of associated documents. Commercial DBMSs, like Oracle 11g, Tamino, DB2 v.9, support XML Schema Definitions at different levels, but the support for schema evolution is quite limited.

Complex schema evolution specifications can easily be expressed by means of XSUpdate [4, 6] expressions. XSUpdate allows to identify a set of components in a schema, specify a modification primitive to apply on them and an adaptation approach for associated documents. In this paper we describe how XSUpdate statements can be translated into XQuery Update expressions achieving the intended effect. We also present E$\mathcal{X}$up, an engine for the translation and evaluation of XSUpdate statements locally, over the network, and within other applications.

In the remainder of the paper we present the XSUpdate language in Section 2 and then our translation algorithms in Section 3. In Section 4 the E$\mathcal{X}$up implementation is described. Finally, conclusions and future developments are discussed in Section 5.

Updates in XML (EDBT Workshop Proceedings), March 22, 2010, Lausanne, Switzerland.

2. XSUPDATE

XSUpdate is a SQL-like language to express evolution statements over an XML Schema and of associated documents. Every schema modification operation is executed over a set of one or more components in a specific schema, named the *evolution objects*. An evolution object can be the root element, a type definition or the declaration of an attribute, element or grouping operator (`sequence`, `choice` or `all`).

An XSUpdate statement is composed of three parts, discussed in the remainder of the section: (i) the identification of the evolution objects, (ii) the specification of the modification operation to carry out on the evolution objects and (iii) the specification of the adaptation to carry out on the associated documents.

2.1 Identifying the Evolution Objects

The identification of the evolution objects within an XML Schema is specified by means of an XSPath expression. The XSPath language [5] allows to express navigational path expressions on the hierarchical structure of an XML Schema

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Log" type="LogType"/>
  <xs:complexType name="LogType">
    <xs:sequence maxOccurs="unbounded">
      <xs:choice>
        <xs:element name="DestIP" type="IPAddressType"/>
        <xs:element name="SourceIP" type="IPAddressType"/>
      </xs:choice>
      <xs:element name="Cookies" type="xs:string" minOccurs="0"/>
      <xs:element ref="Packet" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="SchemaVersion" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:simpleType name="IPAddressType">
    <xs:restriction base="xs:string">
      <xs:minLength value="7"/>
      <xs:maxLength value="15"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="Packet" type="PacketType"/>
  <xs:complexType name="PacketType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="Date" type="xs:string"/>
        <xs:attribute name="Time" type="xs:string" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:schema>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Log SchemaVersion="String" xsi:noNamespaceSchemaLocation="HTML-Log.xsd"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <DestIP>208.69.34.132</DestIP>
  <Packet Date="11/11/09" Time="17:32:00.010">
    GET /index.htm HTTP/1.1
    Host: www.example.com
  </Packet>

  <SourceIP>208.69.34.132</SourceIP>
  <Packet Date="11/11/09" Time="17:32:00.050">
    HTTP/1.1 301 Moved Permanently
    Location: http://www.example.org/index.htm
    Keep-Alive: timeout=15, max=100
  </Packet>
  <Packet Date="11/11/09" Time="17:32:00.052">
    Content-Type: text/html; charset=iso-8859-1
    Set-cookie: sessionID="777"
  </Packet>

  <DestIP>130.251.61.13</DestIP>
  <Cookies>sessionID="777"</Cookies>
  <Packet Date="11/11/09" Time="17:32:00.062">
    GET /index.htm HTTP/1.1
    Host: www.example.org
    Cookie: sessionID="777"
  </Packet>

</Log>
```

Figure 1: XML Schema `HTML-Log.xsd` (left) and associated XML Document `HTML-Log.xml` (right).

to identify schema components as well as the corresponding elements/attributes in documents valid for the schema. Even if the XML Data Model [10] can be employed to represent schema contents, it cannot be profitably exploited to represent the graph structure of schemas. In fact, there can be elements or attributes that need to be bound with their global types or referred global declarations. To easily exploit these relations a graph representation of XML Schemas is adopted on which concise and intuitive navigational expressions can be specified, being able, for instance, to retrieve the elements in a declaration regardless on how their type has been actually defined.

Details of the internal structure of types and elements are seldom required, most notably when their identification is the purpose of the navigational expression. Therefore, a graph representation of schemas at high and low levels of abstraction has been proposed [5]. At low level all the details of a schema are represented, whereas at high level the structure of elements and types is simplified removing grouping operators.

The XML Schema `HTML-Log.xsd` in Figure 1 describes an HTTP communication of a web browser listing the messages sent and received. Each message contains the IP address of the server, stored in the `SourceIP` element for incoming messages or in the `DestIP` element for outgoing ones. The data sent and received are contained in one or more `Packet` elements, along with their processing date and time.

The low level graph representation of the schema is reported in Figure 2. In the graph four types of nodes can be distinguished: declarations of elements (rounded rectangles), attributes (hexagons), operators (ellipses) and type definitions (rectangles). Explicit links, those due to the schema hierarchical structure, are represented by solid arrows, while implicit links that lead to the graph representation are represented with dashed arrows. Each node reports the cardinality specified in the schema (for conciseness (1,1) is omitted), and is numbered according to the pre-ordered traversal of the schema nodes.

To identify the low/high level parents of an element, an attribute or an operator, starting from the corresponding node, implicit and explicit edges (with the exclusion of implicit edges representing global elements references) are navigated backwards towards the schema root. For low level parents along each path, the first element or operator is selected. In case of high level parents, instead, the first element is selected. If one of the selected node is global, also nodes referencing it are selected. The low/high level parents of a type are the union of the low/high level parents of the elements/attributes using it.

Example 1. *Consider the* `DestIP`^{6} *and the* `SourceIP`^{7} *elements, in both cases the low level parent is the* `choice`^{5} *operator and the high level parent is the* `Log`^{2} *element.*

The `IPAddressType`^{8} *definition is employed by the* `DestIP`^{6}*,* `SourceIP`^{7} *elements. Its low/high level parents are the union of the low/high level parents of these two elements, and therefore they are identical to that of the previous case.*

Consider the `Date`^{14} *attribute, its low and high level parents are the same. Backward navigating the inward edges leads to the identification of* `Packet`^{12} *and* `Packet`^{9} *elements referring it.*

XSPath employs a compositional semantic similar to that of XPath [11] but using the names specified in declarations and definitions as primary identification means. As in XPath, axes are employed to identify nodes in a given relation, while node selectors and predicates can be used to identify only those nodes exhibiting specific properties.

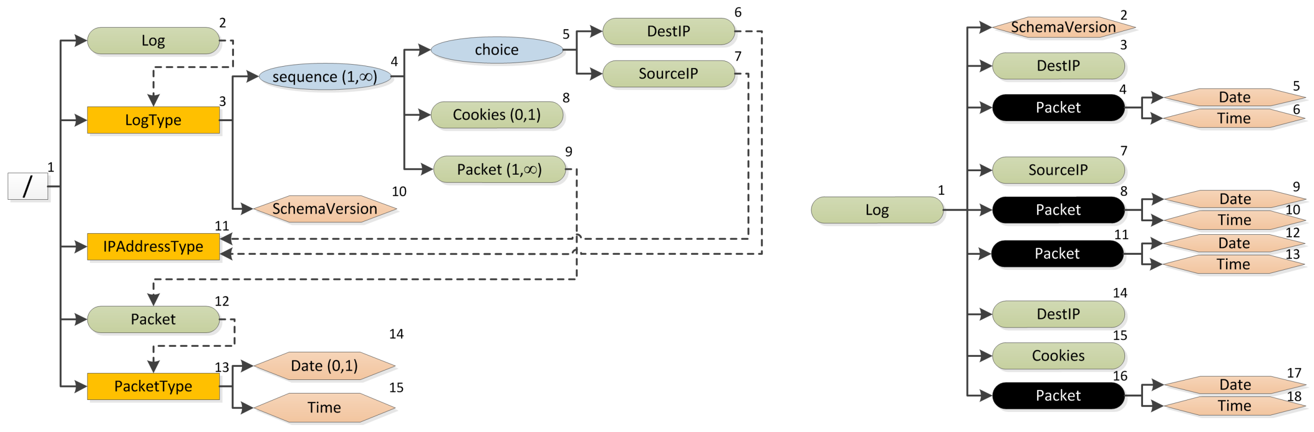

Figure 2: Low level schema representation (left) and document representation (right).

EXAMPLE 2. *As in XPath, expressions starting with a slash are absolute and are evaluated from the schema root. To identify the* `IPAddressType` *type definition the XSPath expression* `/HL::child::type(IPAddressType)` *can be used. The axis specification* `HL::child` *identifies the nodes child of the root in the high level graph, while the node selector* `(type(IPAddressType))` *selects only those types definition named* `IPAddressType`*. An abbreviated syntax is also defined, by which this XSPath expression can be equivalently written as:* `/#IPAddressType`*. To navigate the internal structure of an element the symbol* `!` *is used instead of* `/`*, while predicates can be employed to select nodes meeting a condition. To select the* `Date` *attributes in any global type derived from* `xs:string`*, the XSPath expression* `/#*[typeDerivedFrom(xs:string)]!@Date` *can be specified.*

2.2 Updating a Schema

The second part of an XSUpdate statement specifies the modification primitive to apply on the identified evolution objects. Each primitive can be applied only under specific conditions, for instance only on simple types, or only when a given global type exists.

EXAMPLE 3. *To modify the type definition* `IPAddressType` *replacing the* `minLength` *and* `maxLength` *restrictions with a* `pattern` *restriction the following statement can be specified:*

```
UPDATE SCHEMA ("HTTP-Log.xsd")/#IPAddressType
REPLACE RESTRICTIONS minLength, maxLength
WITH pattern =
"[0-9]{1-3}\.[0-9]{1-3}\.[0-9]{1-3}\.[0-9]{1-3}"
```

Types, elements, attributes, operators or group of nodes can be inserted specifying a position in the schema and providing their XML definition or specifying their peculiarities.

EXAMPLE 4. *To insert an element declaration named* `Direction` *of type* `xs:string` *in the* `Log` *declaration as the last child of the* `sequence` *operator, assuming the last child as the default position, the following statement can be specified:*

```
UPDATE SCHEMA ("HTTP-Log.xsd")/Log!sequence
INSERT ELEMENT Direction OF TYPE xs:string
```

Types, cardinality specifications and names can be changed as required. Any definition/declaration can be removed specifying how to deal with its effects on the rest of the schema.

EXAMPLE 5. *To remove the type* `IPAddressType` *and any element or type depending on it, the following statement can be specified:*

```
UPDATE SCHEMA ("HTTP-Log.xsd")/#IPAddressType
REMOVE CASCADE
```

A global definition or declaration can be made local to the referring elements or vice-versa, while any node in the schema can be moved simply specifying its new position.

EXAMPLE 6. *The global type definition* `LogType` *can be made local to the* `Log` *element by the following statement:*

```
UPDATE SCHEMA ("HTTP-Log.xsd")/Log
MIGRATE GLOBAL TO LOCAL
```

2.3 Adapting Associated Documents

The parts of documents that are affected by the schema modification can be determined through the XSPath expression, the modification primitive and the given schema. Whenever the XSPath expression identifies an element declaration or a type definition, the counterparts in a document are elements corresponding to such declaration/definition. Whenever it identifies a structural component of an element, each counterpart is composed by the set of elements such structural component binds. In the remainder we name *instances* of an evolution object any counterpart in a document. Instances that are repetitions of the same element declaration/type definition can be collected in *sets of instances*.

EXAMPLE 7. *Suppose that the evolution object is the element declaration* `Packet`*. Figure 2 contains a graph representation of the document* `HTML-Log.xml`*. The corresponding instances in this document are the elements highlighted in the figure, numbered 4, 8, 11 and 16. The corresponding sets of instances are three different sets containing respectively the elements {4}, {8,11} and {16}. The elements {8,11} are considered together because they are contained in the same occurrence of the* `sequence` *operator.*

Schema modifications can invalidate associated documents, therefore XSUpdate offers three different approaches to handle them. Documents can be left unmodified, an automatic approach can be applied to minimally change the documents in order to make them valid for the updated schema, or the entire adaptation process can be controlled by the user throughout XQuery Update expressions.

After the adaptation, the parts of the documents affected by the schema evolution are revalidated and invalid documents can be disassociated from the schema or can result in the rollback of the whole evolution process.

2.3.1 No Adaptation

The first and simplest approach is to leave documents unaltered by specifying the `NO ADAPT` clause and simply validate the parts of the documents affected by the modification. If the documents are no longer valid the operation is rollback unless the `REMOVE INVALID` option is specified. In this case documents are removed from the instances of the schema. As defined in [4, 6], analyzing the specified modification primitive and schema definition we can determine that document validity is unaffected.

Example 8. *To specify the restrictions replacement specified in Example 3 and leave any document containing invalid IP addresses disassociated from the modified schema, the clause `NO ADAPT REMOVE INVALID` can be added to the statement in Example 3.*

2.3.2 Automatic Adaptation

The automatic adaptation approach follows two major guidelines: (i) The smallest modification to document nodes affected by the modification to re-establish the document validity are made. (ii) Insertions of new nodes, alterations or removal of existing data are performed only if no other alternative is viable. This approach guarantees document validity after the adaptation.

Example 9. *To change the cardinality of the `Cookies` element from `(0,1)` to `(1,1)`, the following expression can be specified:*

```
UPDATE SCHEMA ("HTTP-Log.xsd")/Log/Cookies
CHANGE CARDINALITY TO 1,1
```

Since the adaptation approach is not specified, automatic adaptation is carried out. Therefore a new `Cookies` element is inserted in each message without one. The value of the inserted element is an empty string (the default value for the `string` type).

2.3.3 User-defined Adaptation

The automatic adaptation approach previously presented does not allow to specify contents for new inserted elements. Moreover, it does not allow to specify how to modify documents in order to make them valid for the new schema taking into account values of other elements, especially those surrounding the inserted/modified/deleted element(s).

For this purpose, we propose to associate an XSUpdate statement with XQuery Update expressions that will be evaluated on each document instance of the schema being modified. Since the XSUpdate statement applies a modification primitive on the evolution objects and more than one set of elements in a document can be one of its instances, it is required to identify the *environments* where the XQuery Update expressions should be evaluated within a document. Therefore, the XQuery update expressions should present free variables that will be automatically bound to values contained in the environment.

Given an XSUpdate statement specifying a modification at schema level, the following template can be instantiated for a user-defined document adaptation.

```
FOR EACH ENVIRONMENT
REFERENCING [target AS variable]
            [container AS variable]
            [parent AS variable]
      DO iteration query
FOR EACH DOCUMENT AS variable
      DO document query
```

The iteration and document queries are XQuery Update expressions presenting free variables that should be automatically bound relying on the specified XSUpdate statement. `Target`, `container`, and `parent` are parts of the environment and they are optional. If specified, the corresponding variable(s) should occur free in the iteration query. In what follows we will discuss the notion of environment and the two types of queries.

Environments. Each environment is formed by a quadruple: (`target`, `parent`, `container`, `insertPos`). The first tree components are sets of document nodes, while the last one specifies a position in a document. Given an evolution object *eo*, for insertion primitives, the `target` is a set of nodes corresponding to a *single instance* of *eo* in a document. For other primitives the `target` is a set of nodes corresponding to a *single set of instances* of *eo* in a document. The different definition of `target` allows users to better formulate the adaptation of document in case of insertion or modification/deletion. Independently from the type of primitive, the `container` and the `parent` are a set of nodes corresponding to the single instance of the low/high level parent of *eo*, respectively. `insPos` only appears when the invoked modification primitive is an insertion, and it represents whether the inserted node(s) should be positioned before a child of the `parent` node or as its last child. The nodes contained in the `target`, `parent`, and `container` are those which values can be used in the iteration query.

Example 10. *Consider the insert operation of Example 4. The evolution object is the* `sequence` *operator ranked 4 in Figure 2, therefore the* `target` *is composed of the elements in a document bound by the operator. The corresponding environments with respect to the document in Figure 1 are reported in the following table.*

target	***container***	***insPos***
DestIP^3, *Packet*^{4}	*Log*^{1}	*BEFORE SourceIP*^{7}
SourceIP^7, *Packet*^{8}, *Packet*^{11}	*Log*^{1}	*BEFORE DestIP*^{14}
DestIP^{14},*Cookies*^{15}, *Packet*^{16}	*Log*^{1}	*AS LAST INTO Log*^{1}

As the instances of the low and high level parent of the evolution object coincide, in each environment the `container` *is identical to the* `parent`*. The operators used in the definition of the position within the children of* `Log` *have been preferred to the* `after` *operator because of the XQuery Update processing model.*

EXAMPLE 11. *Suppose that the evolution object is the local* `Packet` *element definition and that the primitive to invoke is REMOVE. The* `target` *is composed by the set of instances of the evolution object* `Packet` *grouped according to the repetition of the* `sequence` *operator that contains* `Packet`*. The* `container` *and the* `parent` *correspond, respectively, to the single instances of the* `sequence` *operator and the* `Log` *element containing the* `Packet` *nodes in the target. The environments are thus the following.*

target	***container***	***parent***
Packet^{4}	*DestIP*^{3}, *Packet*^{4}	*Log*^{1}
Packet^{8}, *Packet*^{11}	*SourceIP*^{7}, *Packet*^{8}, *Packet*^{11}	*Log*^{1}
Packet^{16}	*DestIP*^{14}, *Cookies*^{15} *Packet*^{16}	*Log*^{1}

EXAMPLE 12. *Suppose that the evolution object is the* `Date` *attribute declaration and that the primitive to invoke is* `REMOVE`*. Four environments can be identified in the* `HTML-Log.xml` *document, one for each* `Date` *attribute. In each environment the target is the corresponding* `Date` *attribute, the container is the* `Packet` *element containing it.*

target	***container***
Date^{5}	*Packet*^{4}
Date^{9}	*Packet*^{8}
Date^{12}	*Packet*^{11}
Date^{17}	*Packet*^{16}

As the instances of the low and high level parent of the evolution object coincide, in each environment the `container` *is identical to the* `parent`*.*

Iteration Query. The *iteration query* is an XQuery Update expression that is evaluated once for each identified environment. A user defined variable can be associated with each component of the environment and used in the specification of the iteration query.

EXAMPLE 13. *Consider the schema modification of Example 4 and suppose we wish to adapt the associated documents inserting a* `Direction` *element where needed, whose value is determined according to the target that will contain the inserted element. In case the target contains a* `SourceIP` *element the value should be* `inbound` *otherwise* `outbound`*. The following XSUpdate statement can be specified, where the* `insertAtCurrentPosition` *function is used to insert one or more nodes at the insert position of the current environment. The* `target` *is bound to the* `Message` *variable at line 4 and employed in the iteration query, defined at lines 5 to 8, to inspect the current* `target`*.*

```
1  UPDATE SCHEMA ("HTTP-Log.xsd")/Log!sequence
2  INSERT ELEMENT Direction OF TYPE xs:string
3  FOR EACH ENVIRONMENT
4  REFERENCING target AS $Message DO
5  local:insertAtCurrentPosition
6  (element Direction {
7  if ($Message/self::*:sourceIP)
8  then ("inbound") else ("outbound")})
```

Document Query. The *document query* is an XQuery Update expression that is evaluated once for each document associated with the modified schema. The document being adapted can be bound to a variable that can be referenced from the document query.

EXAMPLE 14. *Suppose we wish to remove the attribute declaration* `Date` *from the schema and adapt associated documents concatenating the value of each removed* `Date` *attribute with that of the sibling attribute* `Time`*. We also wish to update the* `SchemaVersion` *attribute to the value* `1.1`*. The following XSUpdate statement can be specified, where the* `container` *is bound to the* `Packet` *variable at line 5, the* `target` *to the* `Date` *variable at line 6 and employed in the iteration query, specified at lines 7 to 10, to perform the required adaptation. The document query, specified at lines 12 and 13, updates the* `SchemaVersion` *attribute, referencing the document with the* `Doc` *variable bound at line 11.*

```
1  UPDATE SCHEMA ("HTTP-Log.xsd")
2  /#*[typeDerivedFrom(xs:string)]!@Date
3  REMOVE
4  FOR EACH ENVIRONMENT
5  REFERENCING container AS $Packet,
6  target AS $Date DO
7  let $Time:= $Packet/@Time
8  replace value of $Time with
9  concat($Date,' ',$Time),
10 delete node $Date
11 FOR EACH DOCUMENT AS $Doc DO
12 replace value of node $Doc/@schemaVersion
13 with "1.1"
```

3. TRANSLATING IN XQUERYUPDATE

Each XSUpdate statement can be translated into two XQuery Update expressions whose effects, on the XML Schema and the associated XML documents, are the intended effect of the original statement in terms of schema update and document adaptation, respectively. Section 3.1 presents how an XSPath expression, employed for the evolution objects identification, can be translated in XPath/XQuery. Section 3.2 discusses how the schema modification primitives can be represented through XQuery Update expressions. Document validation and the translation of user-defined adaptation statements are discussed in Section 3.3. The interested reader can find an in-depth discussion of these aspects in [4].

3.1 Evolution Objects Identification

XSPath expressions can be translated into XPath/XQuery expressions which achieve the intended effect. Both a generic and specific schema translation approach have been devised.

The main difference between the two approaches is that the generic translation does not consider the XML Schema on which the expression should be evaluated. Therefore, the expression can be evaluated on any schema. By contrast, the specific translation approach takes advantage of the knowledge of the schema peculiarities to develop a more specific XPath/XQuery expression that can be evaluated more efficiently.

The generated translation is a simple XPath/XQuery expression identifying the same set of nodes in that specific schema. For space constrains in what follows we only detail the generic translation approach.

The XSPath expression is split into single paths, that are further split into steps and then into step components (level, axis, node selector and predicate expression). Finally, predicate expressions are split into single predicates. As predicates can contain paths the process is recursively applied, resulting in a list of atomic path components. For each possible component an XPath expression fragment with the same meaning is generated; these fragments are then combined in order to obtain the complete translation of the XSPath expression.

In case of XPath translations the fragments corresponding to level and axis combinations are path expressions (or steps), while node selectors and predicates are translated into predicate expressions and inserted, in the appropriate position, in the path expressions obtained in this way. Those parts of the resulting expression which are redundant or cannot yield to the identification of nodes due to the constraints of schema definitions are then removed simplifying the resulting expression.

Even if some components of an XSPath expression can be efficiently translated in XPath as such, most can take advantage, or even require, the additional features of XQuery, especially user-defined functions and FLWOR expressions. For instance navigating the hierarchical relation from a complex type towards its base type may involve an arbitrary number of nodes and can therefore be more efficiently realized in XQuery employing a recursive function. As a consequence the XQuery translation is generally shorter and more efficient than the XPath one.

Example 15. *Consider the XSPath expression*

`/#*[typeDerivedFrom(xs:string)]!@Date`

Its translation is reported in Figure 3 and discussed in the following. The XSPath expression is split into steps and then the steps are divided into their components.

The / at the beginning of the expression identifies an absolute path, so the expression should be evaluated starting from the schema root. As no axes has been explicitly specified, the high level child axis, selecting declarations of elements and attributes and type definition child of the context in the high level representation of the schema, is considered. Therefore it selects global (child of the root) types, elements and attributes. It is thus translated in the XQuery expression at lines 3 to 5.

```
1   declare function local:evoObjects() as node()*
2   {
3   doc("HTML-Log.xsd")/*:schema/child::element()
4   [self::*:element or self::*:attribute or
5   self::*:simpleType or self::*:complexType)]
6   [self::*:simpleType or self:complexType]
7   [(*:simpleContent | *:complexContent)/
8   (*:extension | *:restriction)/@base="xs:string"
9   or *:list/@itemType="xs:string" or
10  *:restriction/@base="xs:string" or
11  matches(*:union/@memberTypes,
12  "^(.*\W)?xs:string(\W.*)?$")]
13  /(. union getReferredElements())
14  /(. union getReferredTypes())
15  /(. union (*:simpleContent | *:complexContent)
16  /(*:extension | *:restriction))/
17  child::element()[self::*:element or
18  self::*:attribute or self::*:simpleType or
19  self::*:complexType or self::*:sequence
20  or self::*:choice or self::*:all)]
21  [self::*:attribute]
22  [@name="Date" or @ref="Date"]
```

Figure 3: Generic schema translation of the XSPath expression `/#*[typeDerivedFrom(xs:string)]!@Date`

The next component of the first step is the node selector (#) identifying simple and complex types definitions among those selected by the specified axis and level combination. It is enforced by applying the predicate at line 6 to the nodes selected by the translation of /.*

The predicate [typeDerivedFrom(xs:string)] filters among the nodes selected so far only those whose type is an extension or restriction of `xs:string` *or, for union and list simple types, refers* `xs:string` *in its definition. This condition is expressed by adding to the previous condition the predicate at lines 7 to 12. Specifically, the condition at lines 7 and 8 is true for complex types extending or restricting* `xs:string`*, while those at lines 9 and 10 are true for* `xs:string`*-based list and restriction simple types. Finally, the condition at lines 11 and 12 is true for those simple union types with a* `xs:string` *member. Thus, the XPath expression between lines 3 and 12 forms a complete translation of the first step.*

The second step starts with the low level child axis which identifies attributes, types, elements and operators children of the nodes identified in the first step in the low level representation of the schema. Its translation can be found at lines 13 to 20. With respect to the starting nodes, the XPath steps for the identification of: referred global elements (line 13) and global or local types (line 14) are specified. If needed the type inheritance specification in a definition must also be navigated (line 15 and 16). Implicit links are navigated through getReferredGlobalElements, getReferredTypes and getDerivedTypes functions detailed in [4]. The node selector (@Date) identifies attribute declarations named `Date` *and its translation is at lines 21, 22. This second step has to be evaluated starting from the nodes identified by the first one, thus its translation is appended to the translation of the first step.*

The removal of the redundant predicates (i.e. the predicate at lines 4 and 5 which is implied by that of line 6) and unnecessary expression parts yield to the following simplified and more efficient expression:

```
declare function local:evoObjects() as node()*
{
doc("HTML-Log.xsd")/*:schema/child::element()
[self:complexType][(*:simpleContent |
*:complexContent)/(*:extension | *:restriction)
/@base="xs:string"']/(*:simpleContent |
*:complexContent)/(*:extension | *:restriction)
/child::element()[self::*:attribute]
[@name="Date" or @ref="Date"]
};
```

The simplified expression is obtained on the fly from the original one by keeping track of the generated predicates and schema definition constraints contained in the XSPath expression.

The following example discusses the specific translation of the XSPath expression of the previous example.

Example 16. *By applying a specific translation of the expression of Example 15 the following expression would be generated:*

```
declare function local:evoObjects() as node()*
{
doc("HTML-Log.xsd")/xs:schema
    /xs:complexType[@name="PacketType"]
    /xs:simpleContent/xs:extension
    /xs:attribute[@name="Date"]
};
```

Note that, in the specific translation we have not to specify that the base type of the `PacketType` *complex type should be* `xs:string`*. Moreover, we know that the* `Date` *attribute is not a reference to a global attribute. This knowledge can be achieved only by considering the schema and cannot be assumed in the generic translation.*

3.2 Schema Modification

The modification primitive translation is based upon a set of XQuery Update templates, one or more for each primitive, which are instantiated according to the specified parameters. For instance, in case of an element removal, it is checked that the no dependent declaration exists, instead if the `CASCADE` parameter is specified any dependent declaration is removed as well. XSUpdate generic translations check for the applicability preconditions and employs the generic schema translation of the XSPath expression, while in case of specific translations the preconditions are evaluated during the translation process and the specific schema translation is adopted. In both cases the resulting expression is simplified, using techniques similar to those of Example 15. The user can require that the preconditions are matched by all evolution objects, or decide to modify only those evolution objects matching the preconditions. In this case, the form of the generic translation is the following:

```
for $eo in local:evoObjects()
where applicability preconditions hold
return modification application
```

where `local:evoObjects()` is the function obtained by the translation of the XSPath expression.

Example 17. *Consider the XSUpdate statement of Example 14, the translation of the* ***REMOVE*** *primivitive, when no parameters are specified, is very simple. The applicability conditions are the following: it can be invoked on any node of the schema except the root (line 3) and the node should not be referred by element declarations (line 4) or type definitions (line 5). The behavior of the functions* ***getDependentTypes*** *and* ***getDependentElements*** *is trivial (details in [4]). The modification consists in removing the identified evolution object (line 6).*

```
1  for $eo in local:evoObject()
2  where $eo[not(self::*:schema)] and
3  not(local:getDependentElements($eo) and
4  not(local:getDependentTypes($eo))
5  return delete node $eo
```

Since in this case the evolution object is a local attribute declaration we are sure that no other declaration/definitions depend on it and therefore the translation can be simplified removing lines 4 and 5.

3.3 Document Adaptation

In order to translate the document adaptation specification contained in an XSUpdate statement the modification/insertion environment and, consequently, instances and set of instances must be identified. The knowledge of the evolution primitive employed and the fact that documents are valid for the original schema are also exploited to perform an efficient revalidation.

In this section we present a unified approach for dealing with document validation and adaptation relying on the concept of the *structure of an element declaration.* In the remainder of the section we first present this concept and then detail document validation and adaptation.

Element Structures. For each (local or global) declaration of an element in the schema, its structure, in form of a tree, is generated. The root of the tree is the element itself. For nodes corresponding to the declaration of elements with complex type, implicit and explicit links are recursively navigated adding a copy of the operators and the identified element/attribute declarations without considering the structure of subelements. For nodes corresponding to the declaration of simple type elements and attributes the information about the type is attached to the node. A graphical representation of the structures of the `Log` and the `Packet` declarations are reported in Figure 4. For each node the cardinality constraints of the corresponding declaration are reported next to the node name between round brackets.

Each node of an element structure is also associated with two counters, one to track the number of visited instances and the other for the number of visited sets of instances.

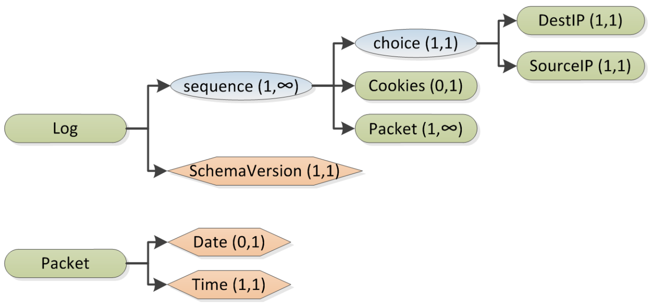

Figure 4: The element structures of `Log` and `Packet` of the `HTML-Log.xsd` schema.

Therefore, the structure of an element e contains all elements/attributes declarations that can appear within e along with the information required to validate corresponding children in a document (type definitions and cardinality constraints).

Document Validation. The validation of a document is performed by visiting the entire document and by considering the generated element structures.

Starting from the document root and, recursively for each of its descendant, the element occurrences are checked against the corresponding structures. The cardinality constraints associated with each element structure node allow the verification, through a single scan of the document, that mandatory nodes are present and that visited nodes are allowed.

We remark that validation can be restricted to the parts of the documents affected by schema modification. In this case, the elements of the document corresponding to the high level parent of the evolution object are determined and the corresponding element structure employed to check their structure. Note that, whenever the update operation is related to the structure of an element, there is no need to check the underlying structure of its children.

Example 18. *Consider the following XSUpdate modification primitives, applied on the `Date` attribute declaration, which is a child of the `Packet` element.*

1. *`CHANGE CARDINALITY TO 1,1`*
2. *`CHANGE TYPE TO xs:date`*
3. *`REMOVE`*

The first primitive requires the mandatory occurrence of the attribute. Therefore, the validation process should check that each `Packet` element (the instances of the parent of the evolution object) contains one `Date` attribute.

The second primitive requires to change the type of the attribute, therefore only the values of the `Date` attributes (the instances of the evolution object) must be checked against the new type definition.

The last primitive specifies to remove the attribute. The validation process should thus simply check that no `Date` instances occur in the associated documents.

The parts of a document affected by a schema modification are determined by an XPath expression generated according to the evolution statement and the schema definition.

Example 19. *Consider the XSPath expression of Example 14 and the `HTML-Log.xsd` schema. The `Date` attribute can appear within the `Packet` element of the root element `Log` or within the root element `Packet`. Therefore, the XPath expressions to locate the `Date` attribute within a document valid for the `HTML-Log.xsd` schema are `/Log/Packet/@Date` and `/Packet/@Date`.*

Enforcing document adaptation. To enforce user-defined adaptation as specified within an XSUpdate statement, the document and iteration queries should be translated into XQuery Update functions.

The translation is performed as follows: we create two functions (named the iteration and the document functions) whose bodies are the iteration and document queries, respectively. The parameters of these functions are the free variables contained in the iteration and document queries.

Example 20. *Consider the XSUpdate statement of Example 14; the functions declared at line 1 and 10 are, the iteration and the document function, respectively.*

```
1  declare updating function local:iterationFunc
2    ($Date as node(), $Packet as node())
3  {
4  let $Time:= $Packet/@Time
5  replace value of $Time with
6  concat($Date,' ',$Time),
7  delete node $Date
8  };
9
10 declare updating function
11 local:documentFunc($Doc as node())
12 {
13 replace value of node $Doc/@schemaVersion
14 with "1.1"
15 };
```

The document function is evaluated once for each document instance of the original schema. The iteration function is evaluated once for each environment within a document. The environments are determined by means of the `identifyEnvironments` function that depends on the kind of modification primitive invoked on the schema.

In case of modification, the behavior of this function depends on the evolution object (identified by means of an XSPath expression), the document to adapt, and the original schema. Each element structure corresponding to one of the high level parents of the evolution object is annotated, marking the nodes belonging to the environment. The function then analyzes the document contents against the schema definition and the XSUpdate statement specified. When an environment is detected, the iteration function is evaluated with the parts of the environment which have been bound in the XSUpdate statement as parameters.

In case of insertion, this function also determines the position where the new element(s) should be inserted. For this purpose the element structure is further annotated with place-holders representing the position within the element structure where the new element(s) should be inserted. By checking at the same time the annotated element structure and the content of the document element where the new element(s) should be inserted, the position of new elements are determined.

Example 21. *Continuing Example 20 the following translation will be produced. The body of the query iterates on all the documents associated with the modified schema, identified by means of the* `documentCollection` *variable (line 5). For each document, the document function is evaluated (line 6). In each document the environments are then identified by means of the* `identifyEnvironments` *function (line 7), which analyzes the document contents against the schema definition and the XSUpdate statement specified. In each detected environment, the iteration function is evaluated with the parts of the environment which have been bound in the XSUpdate statement as parameters (line 9 and 10).*

```
1  let $evoObject :=
2  "/#*[typeDerivedFrom(xs:string)]!@Date"
3  let $modPrimitive= "REMOVE"
4  let $schema := doc("HTML-Log.xsd")
5  for $document in $documentCollection
6   local:documentFunction($document),
7   for $env in local:identifyEnvironments
8   ($schema, $doc, $evoObject,$modPrimitive)
9    local:iterationFunc(local:getTarget($env),
10     local:getContainer($env))
```

4. E$\mathcal{X}$UP

E$\mathcal{X}$up is a Java application for the translation and evaluation of XSUpdate statements. E$\mathcal{X}$up offers two user interfaces, one applet-based for Web use and one for local use. As E$\mathcal{X}$up fully implements the algorithms presented in Section 3 it can also be used to validate XML documents or parts of them and to translate XSPath expressions. Other important features include detailed syntactic or semantic error reporting and visual analysis of the modifications performed on a schema or its associated documents. A comprehensive set of APIs is also available, offering all features of the user interfaces and additional implementation-related options.

Document collections and schemas can be loaded from files or from an XML native or enabled DBMS. The translation of XSUpdate evolution statements on documents employs external Java functions to perform the environments identification processes described above. An effort has been made to support all common Java XQuery Update libraries supporting external functions: Saxon EE[1], MXQuery[2] and Qizx/open[3]. Due to the different features and characteristics offered by these libraries, the actual translations may have a different structure than that reported in Example 20. The E$\mathcal{X}$up architecture is reported in Figure 5.

[1]http://www.saxonica.com
[2]http://mxquery.org
[3]http://www.xmlmind.com/qizx/

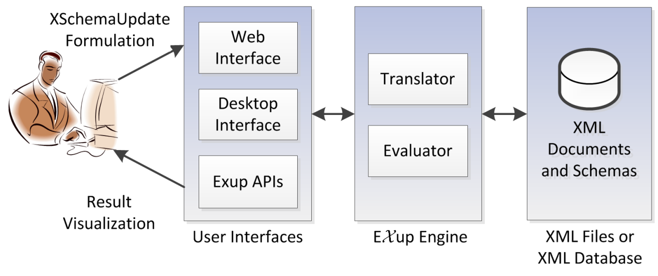

Figure 5: E$\mathcal{X}$up architecture.

We plan to conduct an extensive analysis of the performance of E$\mathcal{X}$up and introduce further optimizations in the near future, nonetheless some considerations, based on preliminary experimental results can be made.

Without any optimization, the translation process of XSPath expressions in both XQuery and XPath, with respect to their length, requires linear time to be generated and produces expression of linear length. The translation of a schema modification, with respect to the XSUpdate statement length, has linear length and requires linear time. To translate and evaluate an XSUpdate statement against a schema usually no more than a tenth of a second is required (with schemas of size 100KB). We also contrasted the performance of the translation and evaluation of the document adaptation specified in an XSUpdate statement with a similar hand-written XQuery Update expression. Note that, especially when optional surrounding elements have to be considered, a similar expression might be too complex for most XQuery users. Even if the translation on average is no more than 20% slower, we are working to enhance the performance of the translation using XPath expressions to identify the environments whenever possible and employing the algorithm of Section 3.3 only when needed.

5. CONCLUSIONS AND FUTURE WORK

XSUpdate expressions allow to specify a schema evolution, defining both the schema modification and the document adaptation to carry out. In this paper we have proposed E$\mathcal{X}$up, a tool for translating and evaluating XSUpdate statements, locally, over the network and from within other applications. Several translation approaches and algorithms have been presented and implemented, to ensure a sound and reasonably fast evaluation.

We have focused on the key features of XML Schema (global and local element declarations, simple and complex type definitions, references, arbitrary nesting of `sequence`, `all`, `choice` grouping operators). However, specific support should be included for other XML Schema peculiarities (like derived types, group elements, substitution groups, abstract definitions, uniqueness and keys).

As the update primitives that can be applied on the schema are low-level, the specification of a complex schema modification different XSUpdate statements may be required. The need arises to consider a sequence of evolution statements with a robust transaction mechanism including rollbacks and commits.

Efficient dynamic techniques for checking that atomic updates preserve validity with respect to a schema have been proposed in [1, 2]. These techniques are exact, but impose run-time overhead on all updates and do not deal with changes to schemas. Raghavachari and Shmueli [8] study the problem of XML documents (known to conform to one schema) that must be validated with respect to another schema. They propose an algorithm that take advantage of similarities between the schemas to avoid validating portions of the documents explicitly. The algorithm, however, does not take advantage of any knowledge of the updates leading from the first schema to the second one in revalidation.

These approaches can be introduced into our incremental validation algorithm, also exploiting the knowledge of the modification primitive applied to the schema.

Currently, after the execution of the document adaptation through the iteration and document queries, the validity of the new document is checked. This is strictly needed because starting from the nodes in the environment the user can require to modify any part of the document. It would be nice to statically determine whether the execution of the document and iteration queries would lead to documents valid for the updated schema. For this purpose a static analysis of the nodes affected by an XQuery Update expression, extending that proposed in [3], can be profitably exploited.

Despite the implemented query optimizations, the translations of XSPath/XSUpdate statements sometimes is longer than needed. We believe that the XML Schema constraints and a formal type system for XSPath can be profitably exploited to further reduce translated expression length.

6. REFERENCES

[1] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental Validation of XML Documents. *ACM Trans. Database Syst.*, 29(4):710–751, 2004.

[2] D. Barbosa, A. O. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficient Incremental Validation of XML Documents. In *ICDE*, pages 671–682. 2004.

[3] M. Benedikt and J. Cheney. Semantics, Types and Effects for XML Updates. In *DBPL*, LNCS(5708), pages 1–17. 2009.

[4] F. Cavalieri, G. Guerrini, and M. Mesiti. Querying and Evolution of XML Schemas and Related Documents Master's thesis, University of Genoa, 2009, URL: http://www.disi.unige.it/person/GuerriniG/2009Cavalieri.pdf.

[5] F. Cavalieri, G. Guerrini, and M. Mesiti. Navigational Path Expressions on XML Schemas. In *DEXA*, LNCS(5181), pages 718–726. 2008.

[6] F. Cavalieri, G. Guerrini, and M. Mesiti. XSchemaUpdate: Schema Evolution and Document Adaptation. Technical report, University of Genova, 2008.

[7] D. Colazzo, G. Guerrini, M. Mesiti, B. Oliboni, and E. Waller. *Document and Schema XML Updates.* In *C. Li and T.W. Ling* editors, *Advanced Applications and Structures in XML Processing: Label Streams, Semantics.*. IGI Global, Information Science Reference, USA/UK, 2009.

[8] M. Raghavachari and O. Shmueli. Efficient Revalidation of XML Documents. *IEEE Trans. Knowl. Data Eng.*, 19(4):554–567, 2007.

[9] J. F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7):383–393, 1995.

[10] W3C. Document Object Model (DOM), 1998.

[11] W3C. XML Path Language (XPath) 2.0, 2007.

[12] W3C. XQuery Update Facility, 2008.

[13] W3C. XML 1.0, Fifth Edition, 2008.

[14] W3C. XML Schema 1.0, 2004.