XML Schema Evolution:
Incremental Validation and Efficient Document Adaptation

Giovanna Guerrini[1], Marco Mesiti[2], Matteo A. Sorrenti[2]
[1] DISI – Università degli Studi di Genova, Italy – guerrini@disi.unige.it
[2] DICO – Università degli Studi di Milano, Italy – mesiti@dico.unimi.it

Abstract

An XML Schema describes the structure of valid documents and can be exploited both for querying and for efficiently accessing valid documents. XML Schemas, however, may need to be updated to adhere to new requirements and to face the changes in the application domain. Starting from a set of schema modification primitives, in this paper we devise a new validation approach that allows to efficiently validate documents, known to be valid for the original schema, for an updated schema. Then, we enhance the approach to adapt the documents to the new schema. Experiments prove that our approach considerably increases the performance of standard validation algorithms in this setting. Moreover, the cost of the adaptation process is contained.

1 Introduction

XML Schemas [17] are a W3C recommendation to describe the structure and to constrain the content of XML documents. XML Schemas, as any form of schema, frequently need to be updated to reflect changing requirements of the application domain. Systems need indeed to be adapted to real-world changes, new functionalities need to be introduced, new data types need to be processed. Commercial alliances change and expand. XML data representation formats and domain-specific schemas, before being adopted as a standard, undergo several revisions resulting in many different versions and the need arises to adapt the corresponding documents.

XML Schemas can be updated in their basic components: elements declarations, simple and complex type declarations. In [8] a set of primitives for evolving XML Schemas has been defined, together with an analysis of the impact of such primitives on documents known to be valid for the original schema. Documents valid for the original schema, indeed, are no longer guaranteed to meet the constraints described by the evolved schema. In principle, these documents should be *revalidated* against the new schema. A naïve approach to revalidation consists in applying a standard validation algorithm (like MSXML, Xerces, and XSV) to each document d and the evolved schema sx', obtained by changing the original schema sx through an evolution primitive. This approach, however, does not take advantage of the fact that some evolution primitives are known not to impact document validity [8]. Moreover, also for primitives whose application can impact validity, the evolution most likely impacts a limited portion of the schema. Consequently, validity needs to be rechecked on restricted portions of a document. The naïve approach, moreover, does not take into account that document d is known to be valid for the original schema sx and that the possible effects on validity of a primitive can be foreseen. Thus, we propose in this paper an *incremental* validation approach for the validation of documents, known to be valid for an original schema sx, against an evolved schema obtain from sx through a specific evolution primitive.

If the evolution impacts validity, a related problem is how to *adapt* documents so to make them valid for the evolved schema. Documents should be adapted through a *minimal* set of updates, so to limit potential damages due to changes in the informative patrimony in the documents. A manual execution of such updates on documents is difficult and likely results in introducing errors and inconsistencies. Thus, approaches for adapting documents to the new schema are needed to maintain the documents valid for the associated schema. We remark that the availability of a schema is relevantly exploited in querying and efficiently accessing documents.

The main contributions of this paper are an algorithm for the incremental validation of XML documents upon XML Schema evolution and an efficient algorithm for adapting the documents, known to be valid for the original schema, to the evolved schema. Both the algorithms have been implemented in X-Evolution [11], a .NET system, and experimentally evaluated. Our incremental validation algorithm outperforms the .NET validation algorithm for primitives that do not alter document validity and improves of an average 20% for other primitives. The execution time of doc-

ument adaptation linearly depends on the document size.

The remainder of this paper is organized as follows. Section 2 briefly surveys related work. Section 3 introduces XML Schemas and evolution primitives. Section 4 introduces some basic functions on the structure of a complex type, auxiliary to both the evolution and the adaptation processes. Section 5 presents the incremental validation and adaptation algorithms, that are experimentally evaluated in Section 6. Section 7 concludes the work.

2 Related Work

The need for XML schema evolution mechanisms has been advocated by Tan and Goh [12] for XML based specifications. A classification of different required modifications is proposed but no specific primitives are proposed nor the impact on existing documents is discussed. Schema evolution had been previously investigated for schemas expressed by DTDs in [10], where a set of evolution operators is proposed and discussed in detail. Problems caused by DTD evolution and the impact on existing documents are however not addressed. Moreover, since DTDs are considerably simpler than XML Schemas [4] the proposed operators do not cover all the set of schema changes that can occur on an XML Schema. DTD evolution has also been investigated in [3] from a different perspective. The focus was on dynamically adapting the schema to the structure of most documents stored in an XML data source. Required modifications are deduced by means of structure mining techniques and documents are not required to exactly conform to the corresponding DTD.

In [6, 15] approaches for making an XML document valid to a given DTD, by applying minimal modifications detected relying on tree edit distances, have been proposed. No knowledge of conformance of the document to a DTD is however exploited. The problem of document revalidation is investigated in [13]. Documents to be revalidated may not be available in advance, they are known to be valid for a given schema S_1 and must be revalidated against a different schema S_2, but the transformations leading from S_1 to S_2 are not known. Incremental validation of XML documents, represented as trees, has been investigated for XML updates [1, 2, 5]. Given an atomic update operation on an XML document, the update is simulated, and only after verifying that the updated document is still valid for its schema the update is executed. Efficiency of those proposals is bound to the *conflict-free* schema property. A schema is said to be *conflict-free* when in type definitions subelement names appear only once. In this paper, we will address the revalidation and adaptation problem only for conflict-free schemas, both for what concerns the original schema and the evolved one. Most schemas employed on the Web do exhibit this property [7].

3 XML Schemas and Evolution Primitives

XML Schemas. We adopt the XML Schema representation of [8, 9], that extends the one proposed in [13]. $\mathcal{EN}$ denotes the set of element tags, $\mathcal{TN}$ the set of (both simple and complex) type names. $\mathcal{TN}$ is the union of $\mathcal{TT}$ and $\mathcal{AT}$, where $\mathcal{TT}$ is the set of explicitly assigned type names and $\mathcal{AT}$ is the set of system-assigned type names (to identify anonymous types).

Simple types, named $\mathcal{ST}$, can be XML Schema native types in the set $\mathcal{NT}$ or can be derived through `restrict`, `list`, and `union`. Each simple type is characterized by a set of *facets* allowing to state constraints on its legal values. We assume the presence of a predicate f that represents the constraints imposed by a set of facets. The set of simple types is inductively defined as follows: native types (e.g., `decimal`, `string`, `float`, `date`) are simple types; if τ is a simple type, `list`(τ) is a simple type; if $\tau_1, \ldots, \tau_n$ are simple types, `union`($\tau_1, \ldots, \tau_n$) is a simple type; if τ is a simple type and f is a predicate on the facets applicable on t, `restrict`(τ, f) is a simple type. $[\![\tau]\!]$ denotes the set of legal values for τ. Given τ_1, τ_2, $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$ can be determined both by exploiting the built-in native types hierarchy [17] and standard constraint subsumption approaches [14] when facets occur.

Complex types, named $\mathcal{CT}$, are associated with a structure specifying the possible children of a given element. A type structure is represented through a labelled tree. A tree on a set of nodes N is inductively defined by stating that: (i) $v \in$ N is a tree; and (ii) if $T_1, \ldots, T_n$ are trees and $v \in N$, $(v, [T_1, \ldots, T_n])$ is a tree. $childs(v)$ denotes the list of subtrees of v. labelled tree is a pair (T, φ), where T is a tree and φ is a total function from the set of T nodes to a set of labels. Labels of the tree representing a type structure are pairs (l, γ), where $l \in \mathcal{EN} \cup \mathcal{OP}$ and $\gamma \in \Gamma$. $\mathcal{OP} = \{$`SEQUENCE`, `ALL`, `CHOICE`$\}$ denotes the set of operators for building complex types. The `SEQUENCE` operator represents a sequence of elements, the `CHOICE` operator represents an alternative of elements, and the `ALL` operator represents a set of elements without order. By contrast, $\Gamma = \{(min, max) \mid min, max \in \mathbb{N}, min \leq max\}$ denotes the set of occurrence constraints, where min is the attribute `MinOccurs` and max is the attribute `MaxOccurs`. The default value $(1, 1)$ is not shown in our graphics. Let $root(T)$ be the root of tree T, $l(T)$ denote the label of the root of T, and $l_{|_i}(v)$, $i = 1, 2$ denote the i-th component of the label of v.

A *type structure* is a tree T defined on the set of labels $(\mathcal{EN} \cup \mathcal{OP}) \times \Gamma$ for which:

1. $l(T) \in \mathcal{OP} \times \Gamma$;
2. for each subtree $(v, [T_1, .., T_n])$ of T, $l(v) \in \mathcal{OP} \times \Gamma$;
3. for each leaf v of T, $l(v) \in \mathcal{EN} \times \Gamma$;

4. for each subtree $(v, [T_1, \ldots, T_n])$ of T, if $l(v) = \langle \texttt{ALL}, (min, max) \rangle$, $v = root(T)$ and $\forall i, j \in \{1, \ldots, n\}$ $l(T_i), l(T_j) \in \mathcal{EN} \times \Gamma$ and $i \neq j \Rightarrow l_{|_1}(T_i) \neq l_{|_1}(T_j)$, $0 \leq min_i \leq max_i \leq 1$ where $l(T_i) = \langle l_i, (min_i, max_i) \rangle$.

The last condition imposes that `all` labelled nodes can only appear as children of the root element and that their children must be all distinct elements.

XML Schemas, unlike DTDs, allow an element to have different types depending on its context, but an unique type is assigned to each element of the schema depending on its context (global or local to a type τ). A *consistent XML Schema* is a 4-tuple $(\mathcal{EN}_G, \mathcal{T}, \rho, type_G)$:

- $\mathcal{EN}_G \subseteq \mathcal{EN}$ is the set of labels of global elements,
- $\mathcal{T} = (\mathcal{TT} \cup \mathcal{AT}) \subseteq \mathcal{TN}$ is the set of type names,
- ρ associates each $\tau \in \mathcal{T}$ with its declaration, that is:
  - if $\tau \in \mathcal{ST}$, $\rho(\tau) \in \mathcal{NT} \cup \{\texttt{restrict}(\tau_1, f), \texttt{list}(\tau_1), \texttt{union}(\tau_1 \ldots \tau_N) \,|\, \tau_1, \ldots, \tau_n \in \mathcal{ST}\}$;
  - if $\tau \in \mathcal{CT}$, $\rho(\tau) = (\mathcal{EN}_\tau, S_\tau, type_\tau)$, where: $\mathcal{EN}_\tau \subseteq \mathcal{EN}$ is the set of local element names for τ; S_τ is the structure for τ; $type_\tau : \mathcal{EN}_\tau \rightarrow \mathcal{T}$ assigns each local element of S_τ its type.
- $type_G : \mathcal{EN}_G \rightarrow \mathcal{T}$ assigns a global element its type.

When no ambiguity arises we use function *type* to associate a (global or local) element with its type.

Example 1 *Table 1 shows the representation of our reference mail schema example. The first row reports the set of global element names, the set of type names, and function $type_G$ that associates each global element with the corresponding type. Then, for each complex type τ, its definition $\rho(\tau)$ is provided. Specifically, the type structure S_τ and the function $type_\tau$ that associates each local element name $\mathcal{EN}_\tau$ with the corresponding type.* ○

Function $valid$ is considered in the remainder of the paper for representing a standard approach for evaluating the validity of a document against a schema or an element against a type. Function $getPaths$ is defined on different input parameters (either a type, a type structure or an element tag) and returns the XPath expressions of elements presenting such a type, structure, or element tag in the schema. Referring to the mail schema in Fig. 1, $getPaths(personT, mail) = \{$ $/mails/mail/envelope/from, /mails/mail/envelope/to,$ $/mails/mail/envelope/cc\}$. Function $getPaths$ returns the right set of paths depending on the context in which it is invoked. For example, different paths are returned for element $mail$ in t_1 and $personT$. By contrast, function $getElems$ evaluates a set of XPath expressions on a document and returns the corresponding elements.

$\mathcal{EN}_G = \{mails, attachment\}$, $\mathcal{T} = \{mailT, envelopeT, personT\} \cup \{t_1, t_2\}$ $type_G(mails) = t_1, type_G(attachment) = t_2$		
$\rho(t_1)$	sequence(0,∝) — mail	$mail \mapsto mailT$
$\rho(t2)$	sequence — choice (0,1) [picture, audio, movie], text	$picture \mapsto Byte$ $audio \mapsto Byte$ $movie \mapsto Byte$ $text \mapsto string$
$\rho(mailT)$	sequence — envelope, choice (0,∝) [body, attachment]	$envelope \mapsto envelopeT$ $body \mapsto string$
$\rho(envelopeT)$	sequence — from, cc (0,∝), to, date, subject, header (1,∝)	$from \mapsto personT$ $cc \mapsto personT$ $to \mapsto personT$ $date \mapsto date$ $subject \mapsto string$ $header \mapsto string$
$\rho(personT)$	sequence — name (0,1), mail	$name \mapsto string$ $mail \mapsto string$

Table 1. Mail schema representation

Evolution Primitives. In [8, 9] three categories of atomic primitives have been devised: insertion, modification, and deletion of the XML Schema components (simple types, complex types, and elements). Modifications can be further classified in structural and relabelling modifications. Structural modifications allow to modify the structure of a type (subelements, operators that establish the structure and cardinality constraints) while relabelling modifications allow to change the name of an element/type. Table 2 reports the evolution primitives $\mathcal{P}$ relying on the proposed classification. For simple types the operators are further specialized to handle the derived types `restrict`, `list`, and `union`. Primitives marked $*$ in Table 2 (denoted by $\mathcal{P}^*$) do not alter the validity of documents, whereas primitives marked $\circ$ in Table 2 (denoted by $\mathcal{P}^{t_s}$) operates on a type structure and have the same treatment in our algorithms. Primitives in $\mathcal{P}^{t_s}$ require to identify the node in position p (in the preorder traversal of the type structure) to be updated/deleted and, in case of insertion, the position (j) where a node should be inserted. Primitives are associated with *applicability conditions* that must hold before their application to guarantee that the updated schema is still consistent. For example, global types/elements can be removed only if elements in the schema of such a type or that refer to it do not exist. Moreover, when renaming an element in a complex type τ, an element with the same tag should not occur in τ. These conditions should be verified when the corresponding primitive is handled in our algorithms.

Simple Type	$insert_glob_simple_type(\tau, dt, sx)^*$
	$insert_new_member_type(\tau, \tau_M, sx)^*$
	$change_restrict(\tau, f, sx)$
	$change_base_type(\tau, \tau_B, sx)$
	$rename_glob_type(\tau_O, \tau_N, sx)^*$
	$change_item_type(\tau, \tau_I, sx)$
	$glob_to_local(\tau, l, sx)^*$
	$local_to_glob(\tau_L, \tau_G, sx)^*$
	$remove_type(\tau, sx)^*$
	$remove_member_type(\tau, p, sx)^*$
Complex Type	$insert_glob_complex_type(\tau, (\mathcal{EN}_\tau, t_s, type_\tau), sx)^*$
	$insert_local_elem(l, (min, max), (p, j), t_s, sx)^\circ$
	$insert_ref_elem(l, (min, max), (p, j), t_s, sx)^\circ$
	$insert_operator(op, (min, max), (p, j), t_s, sx)^\circ$
	$rename_local_elem(l_N, l_O, t_s, sx)$
	$rename_glob_type(\tau_O, \tau_N, sx)^*$
	$change_type_local_elem(\tau_N, l, t_s, sx)^\circ$
	$change_cardinality((min_N, max_N), p, t_s, sx)^\circ$
	$change_operator(op_N, p, t_s, sx)^\circ$
	$glob_to_local(\tau, l, sx)^*$
	$local_to_glob(\tau_L, \tau_G, sx)^*$
	$remove_elem(l, t_s, sx)^\circ$
	$remove_operator(p, t_s, sx)^\circ$
	$remove_substructure(p, t_s, sx)^\circ$
	$remove_type(\tau, sx)^*$
Element	$insert_glob_elem(l, \tau, sx)^*$
	$rename_glob_elem(l_O, l_N, sx)$
	$change_type_glob_elem(l, \tau_N, sx)$
	$local_to_ref(l, t_s, sx)^*$
	$ref_to_local(l, t_s, sx)^*$
	$remove_glob_elem(l, sx)$

Table 2. The evolution primitives

Example 2 *Let t_e be the structure of* envelopT *of schema sx in Fig. 1. By applying the evolution primitive p_1 =* change_cardinality$((2,\infty), 3, t_e, sx)$*, the type structure t_e^1 in Fig. 1(a) is obtained. By contrast, by applying the evolution primitive p_2 =* change_operator(choice$, 1, t_e, sx)$*, the type structure t_e^2 in Fig. 1(b) is obtained.* ○

4 Type Structures for Validity and Adaptation

The type structure t_s of a type $\tau \in \mathcal{CT}$ determines which subelements occur and in which order in a document element declared of type τ. The tree representation of t_s in our context has two purposes: for easily identifying the components that need to be modified and for easily drawing a tree representation of a schema in a graphical interface (see X-Evolution [11]). Another way to see the type structure is as a grammar whose instances are the correct sequences of subelements for an element declared of type τ.

In this section we introduce some functions working on a type structure both for checking validity and for adaptation that exploit a type structure as a grammar.

Function $validS$. This function takes as input: a list of sibling elements $[T_1, \ldots, T_n]$ in a document, a structure t_s, and a set S of expected element tags relying on

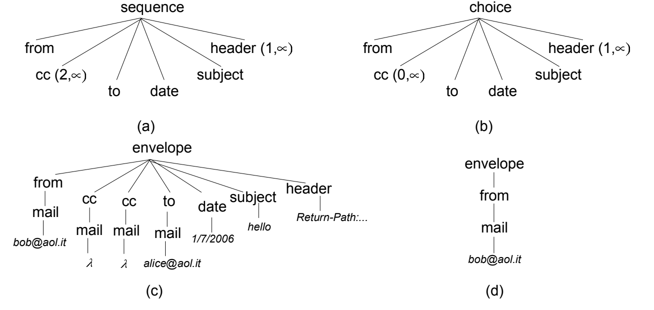

Fig. 1. Type Structures with valid elements

t_s and returns a boolean value. The set of expected element tags is initially determined by an auxiliary function $init$, and in the recursive calls by function $nxtEls$ that treat t_s as a grammar. $init : t_s \rightarrow 2^{\mathcal{EN} \cup \{\lambda\}}$ returns the set of tags S initially expected by t_s. S can contain the symbol λ denoting that t_s also allows empty content. More than one tag can occur in S because of the presence of choice and optional elements in t_s. Once the first tag of the list of sibling elements matches a tag in S, the next expected tags for t_s are determined by function $nxtEls : \mathcal{EN} \cup t_s \rightarrow 2^{\mathcal{EN} \cup \{\lambda\}}$. This function takes as input the identified tag $l \in S$ and t_s, and return the next set of expected tags. Consider the type structure t_2 in Fig. 1. $init(t_2) = \{\texttt{picture}, \texttt{audio}, \texttt{movie}, \texttt{text}\}$, $nxtEls(\texttt{text}, t_2) = \{\lambda\}$ whereas $nxtEls(\texttt{audio}, t_2) = \{\texttt{text}\}$. Function $validS$ is defined as follows:

$$validS([T_1, .., T_n], S, t_s) =$$

$$\begin{cases} validS([T_2,..,T_n], nxtEls(l_1,t_s), t_s) & \text{if } l_1 = l(T_1) \in S \wedge valid(T_1, type(l_1)) \\ \textbf{true} & \text{if } n = 0, \lambda \in S \\ \textbf{false} & \text{otherwise} \end{cases}$$

If the list of sibling trees $\mathcal{C}$ is not empty, the tag of the first element T_1 (l_1) of $\mathcal{C}$ belongs to S, and the content of T_1 is valid for its type, function $validS$ is invoked on the remaining elements, on the set of expected tags relying on t_s knowing that l_1 occurred on the list, and on t_s itself. By contrast, if $\mathcal{C}$ is empty and $\lambda \in S$ then $\mathcal{C}$ is accepted by t_s. Otherwise ($\mathcal{C}$ is not empty and $\lambda \notin S$ or $l_1 \notin S$), $\mathcal{C}$ is not valid for t_s and function $validS$ returns false.

Example 3 *Consider the element* envelope *whose tree representation is reported in Fig. 2 and the structure t_e of type* envelopeT *of Table 1. $validS$ is initially invoked on the five subelements of* envelope *$[T_1, \ldots, T_5]$, $\{\texttt{from}\}$, and t_e. Since $l(T_1) =$* from *and T_1 is valid for* personT*, then $validS$ is invoked on $[T_2, \ldots, T_5]$, $\{\texttt{cc}, \texttt{to}\}$, and t_e. Since $l(T_2) \in \{\texttt{cc}, \texttt{to}\}$ and T_2 is valid for* personT*, then $validS$ is invoked on $[T_3, \ldots, T_5]$, $\{\texttt{date}\}$, and t_e. The behavior is analogous for the rest of the elements and we can conclude that* envelope *is valid for* envelopeT*.* ○

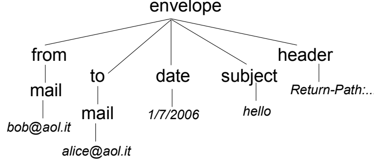

Fig. 2. The `envelope` element

$\sqsubseteq_{ST}$ **relationship**. This relationship holds between a type structure t_1 and a type structure t_2, obtained from t_1 by applying a primitive $p_e \in \mathcal{P}^{t_s}$, when the legal values of t_1 are contained in the legal value of t_2 and this check is performed directly on their tree representations. If p_e changes the cardinality of an element/operator from (min_O, max_O) to (min_N, max_N) and $min_N \leq min_O \wedge max_N \geq max_O$ (that is, the interval of allowed occurrences is extended) the elements valid according to t_1 are still valid for t_2. If p_e changes a `sequence` operator into an `all` operator or the group bound by the operator is composed by a single element, then the elements valid for t_1 are still valid for t_2. If p_e introduces a new optional element/operator in the structure, then the elements valid for t_1 are still valid for t_2. If none of the elements of sx have been defined according to a complex type whose structure is t_1, then no modification to t_1 alter the validity of documents. This relationship is thus exploited in the revalidation process to avoid accessing documents when it is not strictly required.

Algorithm 1: $adaptS$

Data: $[T_1, \ldots, T_n]$: Trees, $S : 2^{\mathcal{EN}}, t_s^N : \mathcal{ST}, opt : \{INS,DEL\}$
Result: $[T'_1, \ldots, T'_m]$ $valid\ for\ t_s^N$

1 Let $l_1 = l(T_1)$ and τ_{l_1} be its type
2 **if** $n \geq 1 \wedge l_1 \in S$ **then**
3   **if not** $valid(T_1,type(l_1))$ **then** $T_1 = genTree(l_1, type(l_1))$
4   **return** $T_1 \cdot adaptS([T_2, \ldots, T_n], nxtEls(t_s^N, l_1), t_s^N, opt)$
5 **end**
6 **if** $n \geq 1 \wedge l_1 \notin S$ **then**
7     $s = choose(S)$ whose type is τ_s
8   **if** $opt =$ INS **then return** $genTree(s, \tau_s) \cdot$
9         $adaptS([T_1, \ldots, T_n], nxtEls(t_s^N, s), t_s^N, opt)$
10   **else return** $adaptS([T_2, \ldots, T_n], S, t_s^N, opt)$
11 **end**
12 **while** $\lambda \notin S$ **do**
13    $s = choose(S)$ whose type is τ_s
14    $\mathcal{C} \leftarrow \mathcal{C} \cdot genTree(s, \tau_s)$
15    $S \leftarrow nxtEls(t_s^N, s)$
16 **end**
17 **return** $\mathcal{C}$

Function $adaptS$. This function is an extension of $validS$ that alters the list of subelements $[T_1, \ldots, T_n]$ of an element in the document when it is not valid for a structure t_s. Altering $[T_1, \ldots, T_n]$ means inserting and/or deleting elements to/from the list. This depends on the evolution primitive employed and will be discussed in next section. Here we present how insertion or deletion are performed. $adaptS$ exploits the auxiliary function $genTree$ that, given an element tag of type τ, generates a valid instance for such a type assigning default values for data content elements and choosing the minimal structure among those that can be obtained from τ. The `envelope` element in Fig. 2 where the data contents are substituted by the empty string is an example of tree generated by $genTree(\texttt{envelope}, envelopeT)$ Function $adaptS$ takes as input a list of sibling elements $[T_1, \ldots, T_n]$ in a document, a type structure t_s, the set of expected labels S according to t_s, and an option saying whether the function is invoked for the removal of elements or the insertion of elements according to t_s. If $n \geq 1$ and the label of the root of T_1 (l_1) belongs to S, the algorithm checks if the content of T_1 meets the constraints imposed by the type of l_1. If it does not, the content of T_1 should be generated, otherwise left unchanged. In both cases, the function returns T_1 concatenated to the list of trees generated by the recursive call of $adaptS$ to the rest of tree list and the next expected elements for t_s. If $n \geq 1$ and the label of T_1 does not belong to S, in case of insertion (i.e., $opt =$ INS) a tag s is chosen from S and an element valid for the type of s is inserted before the head of $\mathcal{C}$ (according to a policy discussed below) and the label of T_1 is checked in the next expected elements. In case of deletion (i.e., $opt =$ DEL), by contrast, T_1 is removed and the label of the next element is checked in the same set S. Whenever $n = 0$ and $\lambda \notin S$, new elements are appended to the result until $\lambda \in S$.

When $|S| > 1$ and one of the tags in S needs to be chosen, function $choose$ is invoked that applies the following heuristics. Tag $s \in S$ with minimal cardinality greater than 0 are chosen. If none is selected, S is considered for the next step. Then, among the identified tags, those having the lowest maximal cardinality are chosen. If more than one tag occurs, one of them is randomly chosen. This heuristics ensures to introduce only mandatory elements with the minimum number of occurrences.

Example 4 *Consider element* `envelope` *in Fig. 2, structures t_e^1, t_e^2, and primitives p_1, p_2 of Example 2. $adaptS$ is invoked with option* INS *for p_1 on $[T_1, .., T_5]$, $\{$*`from`*$\}$, t_e^1. Since $l(T_1) =$* `from` *and T_1 is valid for* `personT`, *$adaptS$ is invoked on $[T_2, .., T_5]$, $\{$*`cc`*$\}$, t_e^1. Since $l(T_2) =$* `to` *$\notin \{$*`cc`*$\}$, a tree is generated for* `cc` *and $adaptS$ is invoked on $[T_2, .., T_5]$, $\{$*`cc`*$\}$, t_e^1. Again, $l(T_2) =$* `to` *$\notin \{$*`cc`*$\}$, another tree is generated for* `cc` *and $adaptS$ is invoked on $[T_2, .., T_5]$, $\{$*`to`*$\}$, t_e^1. The remaining recursive calls return $[T_2, .., T_5]$. Fig. 1(c) shows the new element* `envelope`. *By contrast, $adaptS$ is invoked with option* DEL *for p_2 and same parameters. Since $l(T_1) =$* `from` *and T_1 is valid for* `personT`, *$adaptS$ is invoked on $[T_2, .., T_5]$, $\{\lambda\}$, t_e^2. Since $l(T_2) =$* `to` *$\notin \{\lambda\}$, T_2 is removed as well as the other elements of the list by the recursive invocations. Fig. 1(d) shows the new element* `envelope`. ○

5 Incremental Validation and Efficient Document Adaptation

Incremental Validation Algorithm. Our incremental validation algorithm takes as input a schema sx, a document d valid for sx, and an evolution operation $p_e \in \mathcal{P}$. Output of the algorithm is true only if d is still valid after the application of p_e to sx. The algorithm, relying on the positive verification of the applicability conditions of the evolution primitives, starts checking the validity of the document from the invoked evolution primitive and the characteristics of the schema and moves to check the document only when this is strictly needed. A more efficient algorithm than traditional validation approaches is thus obtained.

If $p_e \in \mathcal{P}^*$, its application does not alter the validity of d. Therefore, no checks need to be performed on d. If p_e renames an (either global or local) element tagged l, the validity of d depends on the occurrence of elements tagged l in d. Therefore, elements tagged l in d are identified and whenever a single occurrence is detected, d is no longer valid. If p_e changes type τ_O of a global element l in the complex type τ_N, the children of elements of type τ_O in d should be extracted to check though function $validS$ that they meet the constraints specified by the structure of τ_N. If p_e changes a simple type (either changing the restriction, the base/member type, or the type of a global element) the algorithm first checks whether the values of the old type τ_O are contained in the new type τ_N. If so, d is valid, otherwise, all the elements of type τ_O in d are identified and their content is checked to belong to the extension of τ_N. If p_e removes a global element l, the root label of d is compared with l. If they are equal, d is not valid. Otherwise, it is still valid. This check is very simple since the applicability conditions of the primitive allows the removal of a global element only if no elements in the schema refers to it. If p_e updates the structure of a complex type through the primitives in $\mathcal{P}^{t_s}$, the old structure t_s is compared with the new structure t_s^N to determine whether $t_s \sqsubseteq_{ST} t_s^N$. If so, d is valid. Otherwise, the children of elements with structure t_s in d are extracted to check through function $validS$ whether they meet the constraints specified by the structure of t_s^N.

Proposition 1 *Let sx be an XML Schema and d be an XML document valid for sx. Let sx^N be an XML Schema obtained from sx by applying $p_e \in \mathcal{P}$. Then,*

$$valid(d, sx^N) \ \textbf{iff} \ revalidate(p_e, d, sx).$$

Document Adaptation Algorithm. The document adaptation algorithm is an extension of the $revalidate$ algorithm (Algorithm 2) in which, when an element is not valid for the new schema, the minimal modifications are performed

Algorithm 2: Revalidate

Data: $p_e : \mathcal{P}, d : \mathcal{DOC}, sx : \mathcal{SX}$
Result: **true** $\iff$ d is valid for the updated schema

1 The applicability conditions of p_e to sx are met
2 **switch** p_e **do**
3 **case** $p_e \in \mathcal{P}^*$ **return** ***true***
4 **case** $p_e \in \{$*rename_glob/local_elem*$\}$
5 **Let** l_O **be** the element tag to remove/rename
6 **if** $getElems(getPaths(l_O, sx), d) = \emptyset$
7 **then return** ***true*** **else return** ***false***
8 **end**
9 **case** $p_e{=}change_type_glob/local_elem(l,\tau_N,sx) \wedge \tau_N \in \mathcal{CT}$
10 **Let** t_s^N **be** the structure of τ_N
11 **if** $\exists e \in getElems(getPaths(l, sx), d)$: $validS(childs(e), init(t_s^N), t_s^N) =$ ***false***
12 **then return** ***false*** **else return** ***true***
13 **end**
14 **case** $p_e{\in}\{change_restrict, change_base/member_type,$
15 $change_type_glob/local_elem\}$
16 **Let** τ_O **be** the old simple type and τ_N the updated one
17 **if** $\tau_O \sqsubseteq \tau_N$ **then return** ***true***
18 $\mathcal{E} \leftarrow getElems(getPaths(\tau_O, sx), d)$
19 **if** $\exists e \in \mathcal{E} : content(e) \notin [\![\tau_N]\!]$
20 **then return** ***false*** **else return** ***true***
21 **end**
22 **case** $p_e = remove_glob_elem(l, sx)$
23 **if** $l(root(d)) = l$ **then return** ***false***
24 **else return** ***true***
25 **end**
26 **case** $p_e \in \mathcal{P}^{t_s}$
27 **Let** $t_s, {t_s}^N$ **be** the old and new structure
28 **if** $t_s \sqsubseteq_{ST} t_s^N$ **then return** ***true***
29 **if** $\exists e \in getElems(getPaths(t_s, sx), d)$:
30 $validS(childs(e), init(t_s^N), t_s^N) =$ ***false***
31 **then return** ***false*** **else return** ***true***
32 **end**
33 **end**

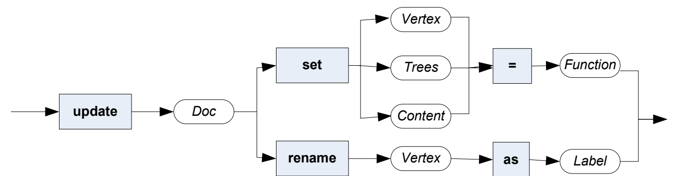

Fig. 3. An SQL-based language for the specification of document updates

on d to make it valid. The modifications are minimal because they only involve the document portions affected by the primitive p_e and because they require to insert/eliminate the minimal number of elements to guarantee validity. Document modifications can be of different types: element renaming, removal of an element with all its content, insertion of an element. In the last case, a default value should be associated with the inserted element e (either a value of the type of e or a default tree generated by function $genTree$). Document modifications are specified by means of a simple SQL-based language whose syntax graph is shown in Fig. 3. Squared nodes represent keywords and oval nodes represent parameters. The new nodes and contents can be specified by means of functions $adaptS$, and $defaultVal$ that returns a default value for a simple type.

The adaptation algorithm (Algorithm 3) works on a document d valid for a schema sx on which an evolution primitive p_e is applied. Depending on the primitive, the algorithm determines if d is still valid for the updated schema sx^N or performs modifications to d to make it valid for sx^N. The applicability conditions of the primitive should be met, otherwise the document is not modified.

If $p_e \in \mathcal{P}^*$ d is not modified at all because p_e does not alter validity. If p_e renames the l_O (either local or global) element, the occurrences of l_O in d are identified and renamed to l_N. If p_e removes a global element and the root of d has the same label, then the document d is removed. Otherwise, the document is left unchanged. If p_e changes the type of a global element in a complex type, all the elements of the original type are detected in the document. For element e, the children of e are checked to adhere to the new type. If not, the children of e are removed and a new content is specified for e by means of function $adaptS$ that adds subelements to e. Since an empty list of trees is passed to function $adaptS$, this function generates from scratch the content of e. If p_e updates a simple type (including `union`, `list`, `restrict` derived types) or changes the type of a global element in a simple type, first the algorithm checks whether the values of the new simple type extends the values of the original type. If so, the document is valid as it is. Otherwise, for each element e of d of the original type its content is changed by assigning a default value of the new type. If p_e updates a type structure t_s among those in $\mathcal{P}^{\mathcal{S}_\mathcal{T}}$, the new structure t_s^N can require to introduce new elements or to remove existing ones, depending on the specific primitive employed and, in case of insert or change of an operator, from the new operator.

Table 3 reports when elements should be inserted or removed. For primitive $change_type_local_elem$ neither insertions nor removals are required, because this primitive does not alter a type structure but the content of subelements. Primitives $insert_operator$ and $change_operator$ require to insert or remove elements depending on the new

Algorithm 3: Adapt

Data: $p_e : \mathcal{P}, d : \mathcal{DOC}, sx : \mathcal{SX}$
Result: d' obtained from d that is valid for sx^N

1 **switch** p_e **do**
2 The applicability conditions of p_e to sx are met
3 **case** $p_e \in \mathcal{P}^*$ **break**
4 **case** $p_e \in \{rename_glob/local_elem\}$
5 **Let** l_O **be** the element tag renamed l_N
6 **for** $e \in getElems(getPaths(l_O, sx), d)$ **do**
$update\ d\ rename\ e\ as\ l_N$
7 **end**
8 **case** $p_e = remove_glob_elem(l, sx)$
9 **if** $\varphi(root(d)) = l$ **then** $d = NULL$
10 **end**
11 **case** $p_e{=}change_type_glob/local_elem(l,\tau_N,sx){\wedge}\tau_N{\in}\mathcal{CT}$
12 **Let** t_s^N **be** the structure of type τ_N
13 **for** $e \in getElems(getPaths(l, sx), d)$ **do**
14 **if not** $validS(childs(e), init(t_s^N), t_s^N)$ **then**
$update\ d\ set\ childs(e) =$
15 $adaptS([], init(t_s^N), t_s^N, \texttt{INS})$
16 **end**
17 **case** $p_e \in \{change_restrict, change_base/item_type,$
18 $change_type_glob/local_elem\}$
19 **Let** τ_N **be** the new simple type updating τ_O
20 **if** $\tau_O \not\sqsubseteq \tau_N$ **then**
21 **for** $e \in getElems(getPaths(\tau_O, sx), d) :$
22 $content(e) \notin [\![\, \tau_N \,]\!]$ **do**
23 $update\ d\ set\ content(e) {=} defaultVal(\tau_N)$
24 **end**
25 **case** $p_e \in \mathcal{P}^{t_s}$
26 **Let** t_s^N **be** the new type structure updating t_s^O
27 **if** $t_s^O \not\sqsubseteq_{ST} t_s^N$ **then**
28 **for** $e \in getElems(getPaths(t_s^O, sx), d)$ **do**
29 **if** $delElems(p_e)$ **then**
$update\ d\ set\ childs(e) =$
30 $adaptS(childs(e), init(t_s^N), t_s^N, \texttt{DEL})$
31 **if** $addElems(p_e)$ **then**
$update\ d\ set\ childs(e) =$
32 $adaptS(childs(e), init(t_s^N), t_s^N, \texttt{INS})$
33 **end**
34 **end**
35 **end**
36 **return** d

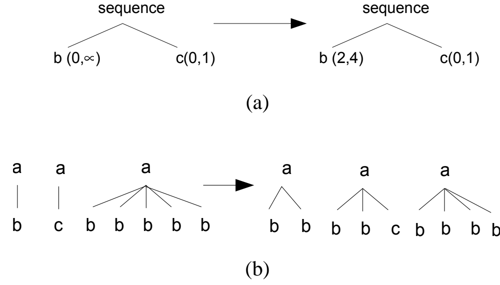

Fig. 4. Change of cardinality and its effects

operator. If the new operator is `choice` it means that operator `sequence` or `all` occurred before. Therefore, from sequences of elements in the document grouped by the operator we need to choose one of them. Thus, elements need to be removed. By contrast, if the old operator was `choice`, it means that the new operator is `sequence` or `all`. Therefore, from an element in the document bound by the operator, we need to insert other elements as specified by the `sequence` or `all` operator. Thus, elements need to be inserted. For primitive $change_cardinality$ both insertions and removals must be performed when both the minimal and maximal cardinalities are updated. This is because a single invocation of function $adaptS$ can add elements or alternatively remove elements. Thus, we need first to remove elements to adhere to the new maximal cardinality and then add elements to adhere to the new minimal cardinality.

Example 5 *Starting from the type structure in Fig. 4(a) the cardinality of* `b` *is changed from* $(0, \infty)$ *to* $(2, 4)$*. This requires two applications of function* $adaptS$*. One for removing elements* `b` *exceeding the maximal cardinality and one for adding elements* `b` *missing the minimal cardinality. The original and updated elements are in Fig. 4(b).* ○

Once the effects of the evolution primitives have been propagated to the document making it valid for the new schema, the document itself can be returned.

Proposition 2 *Let* sx *be an XML Schema and* d *be an XML document valid for* sx*. Let* sx^N *be an XML Schema obtained from* sx *by applying* $p_e \in \mathcal{P}$*. Then,*

$$valid(adapt(p_e, d, sx), sx^N) = \textbf{\textit{true}}.$$

6 Experimental Evaluation

X-Evolution. X-Evolution [11] is a .NET system for handling collections of XML documents and schemas. Documents and schemas are graphically represented as trees and users can specify on the tree representation of a schema the

primitive	addElems	delElems
$insert_local_elem$	**true**	**false**
$insert_ref_elem$	**true**	**false**
$insert_operator$ (op_N=`choice`)	**false**	**true**
$insert_operator$ (others)	**true**	**false**
$change_type_local_elem$	**false**	**false**
$change_cardinality$	*see table below*	
$change_operator$ (op_N=`choice`)	**false**	**true**
$change_operator$ (others)	**true**	**false**
$remove_operator$	**false**	**true**
$remove_substructure$	**false**	**true**
$remove_element$	**false**	**true**

min_N	max_N	addElems	delElems
$>$	$<$	**true**	**true**
$\leq$	$\geq$	**false**	**false**
$>$	$\geq$	**true**	**false**
$\leq$	$<$	**false**	**true**

Table 3. Output of $addElems$ **and** $delElems$

evolution primitives according to the kind of node (element tag, simple or complex type). The $revalidate$ algorithm is applied to check whether documents valid for the original schema are still valid for the updated one. In case of invalidity, the user can then decide to adapt those documents to the new schema (using the *adapt* algorithm) or to leave them without schema. In the back end a DBMS handles documents, schemas and information of which document is valid for which schema.

Experimental Results. Different experiments have been conducted to prove the effectiveness and efficiency of our approach. We gathered from the Web different schemas and corresponding valid documents. Among them the XML `DBLP` document (`http://dblp.uni-trier.de/xml/`), the statistics on American baseball competitions and plays of Shakespeare collections (`http://www.ibiblio.org/xml/examples/`). The considered collections have been classified according to their size and the level of nesting. *Small* documents are those with size less than 1 KB, *average* documents are those with size between 1 KB and 1 MB, and *big* documents are those with size greater than 1 MB. *Shallow* documents are those with at most 5 levels of nesting, *average depth* documents are those with 5 levels of nesting to 10, and *deep* documents are those with more than 10 levels of nesting. The average characteristics of the documents in each class are reported in the following table.

	small	average	big
shallow	256 B	1.3 KB	5 MB
average depth	736 B	232 KB	137 MB
deep	640 B	924 KB	30 MB

On documents of each class we have applied different kinds of evolution primitives that operate on the root of the

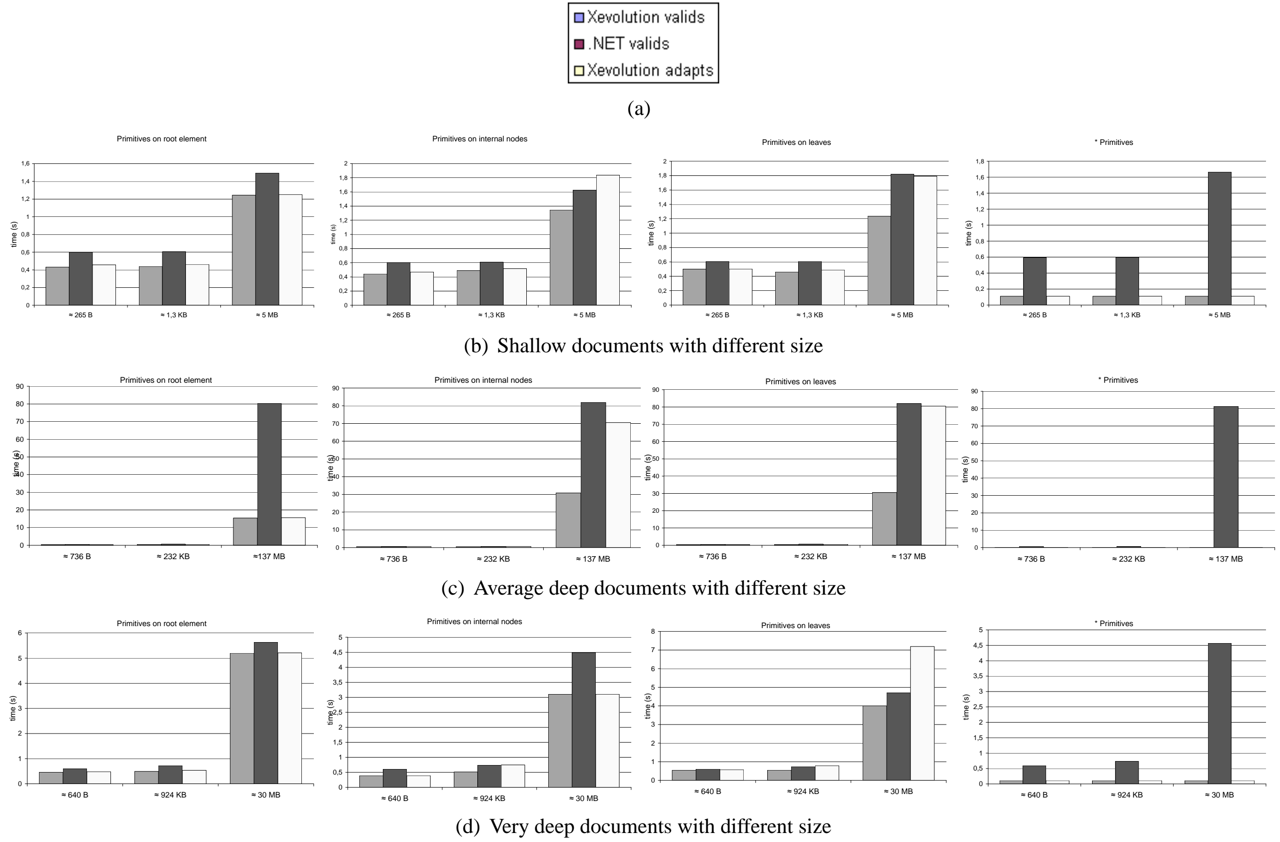

Fig. 5. Comparing the *revalidate*, MSXML 4.0 validation and *adapt* algorithms

document, on internal nodes, and on leaves. We conducted many repetitions of the same evolution primitives and considered the average execution times. Moreover, we also considered the execution time for primitives in $\mathcal{P}^*$. The revalidation algorithm has been compared with MSXML 4.0 validation algorithm available in the .NET framework.

Fig. 5 reports the experimental results on these collections of documents. Rows of the figure reports the experiments conducted on shallow, average depth, and deep documents. The first three graphics in each row represent the execution time for evolution primitives applied on the root of the document, on internal nodes, and on leaves. The last graphic reports the execution times when only primitives in $\mathcal{P}^*$ are used. Each single graphic reports the execution times of *revalidate*, MSXML 4.0 validation, and *adapt* algorithms applied on documents of small, average, and big dimensions.

Last column of Fig. 5 points out how our revalidation algorithm outperforms the MSXML 4.0 validation algorithm for primitives in $\mathcal{P}^*$. Indeed, documents are not accessed and validity is checked only through the schema (in constant time). The performance of MSXML 4.0 validation algorithm is mostly the same for documents with the same size and does not depend on the level of nesting of documents. Our validation algorithm improves the performances of an average of 20% for the other primitives for documents of big size because it operates on small portions of the document. In the *revalidate* algorithm the .NET facilities for accessing documents and evaluating XPath expressions have been used. That means that exploiting indexing techniques available in the back end DBMS the performance of our algorithm would further improve.

The execution time of the *adapt* and *revalidate* algorithms have been compared. The insertion and deletion of internal nodes from the schema have a deeper impact in adapting the structure of an element through function $adaptS$. The performance of the *adapt* algorithm decreases when documents of big dimensions are handled, and in particular when document leaves need to be updated, because the entire document should be loaded in main memory and the probability of "page swapping" increases. This behavior can be however mitigated exploiting indexing techniques and standard DBMS facilities as previously described for the *revalidation* algorithm. Moreover, the graphics point

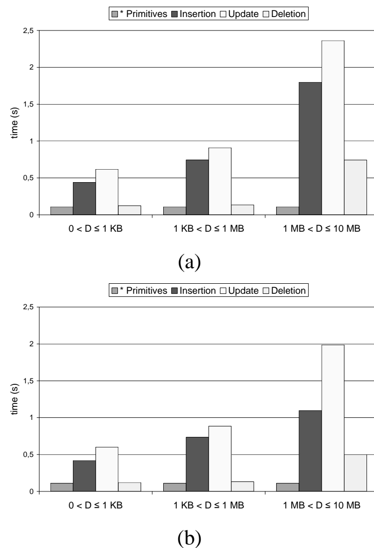

Fig. 6. Execution times of evolution primitive for revalidation and adaptation

out that updates of nodes deeply nested in the structure of a document require more time that those closer to the document root. To further analyse the *adapt* and *revalidation* algorithms we consider the two graphics in Fig. 5. They report the execution times in case of revalidation and adaptation for the evolution primitives that alter the validity of documents (i.e., those for inserting, deleting, and updating elements/types in the schema) and for primitives that do not alter the validity (i.e., those in $\mathcal{P}^*$). For space constraints, we only report the evaluations on documents of average nestings and primitives applied randomly in the schema. Despite the best performances for primitives in $\mathcal{P}^*$ (as expected), we can note that the execution time for revalidation and adaptation linearly increase as the size of documents increase. The update primitives are more expensive than the deletion primitives. These last ones have performances comparable to those of primitives in $\mathcal{P}^*$.

7 Conclusions and Future Work

In this paper we have proposed an approach for the incremental validation of XML documents upon schema evolution. The approach takes advantage of knowing the documents valid for the original schema and the applied evolution primitive to establish what needs to be checked in the documents, if some check is needed. An efficient adaptation algorithm to make the invalidated document portions conform to the evolved schema is also proposed. Both the algorithms have been experimentally evaluated. The validation algorithm has been demonstrated to improve considerably over the naïve solution. The adaptation process execution time linearly depends on the document size.

The work presented in this paper is being extended in several directions. For what concerns the evolution primitives, primitives allowing to *move* a portion of the schema and their impact on the revalidation and adaptation processes need to be investigated. In [8] high-level primitives allowing to conveniently express common sequences of atomic primitives have been proposed. The revalidation and adaptation algorithms are currently being extended to these high-level primitives, and, more generally, to sequences of atomic primitives. Finally, the adaptation mechanism is being enhanced with the possibility of specifying through a query the new contents to be inserted in the adapted documents.

References

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

[2] D. Barbosa, et al. Efficient Incremental Validation of XML Documents. *ICDE*, 671–682, 2004.

[3] E. Bertino, G. Guerrini, M. Mesiti, and L. Tosetto. Evolving a Set of DTDs according to a Dynamic Set of XML Documents. *EDBT Workshops*, LNCS 2490, 45–66, 2002.

[4] G.J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML Schema: A Practical Study. *WebDB*, 79–84, 2004.

[5] B. Bouchou and M.H. Ferrari Alves. Updates and Incremental Validation of XML Documents. *DBPL*, 216–232, 2003.

[6] U. Boobna and M. de Rougemont. Correctors for XML Data. *XSym*, 97-111, 2004.

[7] B. Choi. What are Real DTDs Like? *WebDB*, 43–48, 2002.

[8] G. Guerrini, M. Mesiti, and D. Rossi. XML Schema Evolution, TR Universit'a di Genova, 2005.

[9] G. Guerrini, M. Mesiti, and D. Rossi. Impact of XML Schema Evolution on Valid Documents. ACM-WIDM Workshop, 2005.

[10] D. K. Kramer and E. A. Rundensteiner. Xem: XML Evolution Management. *RIDE-DM*, 103–110, 2001.

[11] M. Mesiti, R. Celle, M.A. Sorrenti, G. Guerrini. X-Evolution: A System for XML Schema Evolution and Document Adaptation.*EDBT*,1143–1146,2006.

[12] M. B. L. Tan and A. Goh. Keeping Pace with Evolving XML-Based Specifications. *EDBT Workshops*, LNCS 3268, 280–288, 2004.

[13] M. Raghavachari and O. Shmueli. Efficient Schema-Based Revalidation of XML. *EDBT*, 639–657, 2004.

[14] D. Srivastava. Subsumption and Indexing in Constraint Query Languages with Linear Arithmetic Constraints. *Annals of Mathematics and Artificial Intelligence* 8(3-4): 315-343, 1993.

[15] S. Staworko and J. Chomicki. Validity-Sensitive Querying of XML Databases. *dataX* 2006.

[16] W3C. Extensible Markup Language 1.0, 2004.

[17] W3C. XML Schema Part 1: Structures, 2004.