Impact of XML Schema Evolution on Valid Documents

Giovanna Guerrini
Università di Pisa, Italy
guerrini@di.unipi.it

Marco Mesiti
Università di Milano, Italy
mesiti@dico.unimi.it

Daniele Rossi
Università di Genova, Italy
dani_reds@yahoo.it

ABSTRACT

In this paper we investigate the problem of XML Schema evolution. We first discuss the different kinds of changes that may be needed on an XML Schema. Then, we investigate how to minimize document revalidation, that is, detecting the document parts potentially invalidated by the schema changes that should be revalidated.

Categories and Subject Descriptors: H.2.1Logical Design:schema and subschema

General Terms: Management, Algorithms

Keywords: XML, schema evolution, document revalidation

1. INTRODUCTION

The amount and speed of data interchanged nowadays have enormously increased and so has the number of data sources. In the same time, the need has arisen of providing data and documents available on and interchanged through the Web with a structure, so to make their retrieval more efficient and effective. This need has lead to the development and to the quick acceptance of XML [13]. An XML document can be coupled with a DTD (Document Type Definition) or with an XML Schema [14] that describes the structure, the order, and the type of subelements and attributes of each element appearing in the document. A Schema presents some relevant differences with respect to a DTD [5]: a Schema is an XML document and allows a more sophisticated typing of elements, providing a set of data types compatible with those employed in most data models. It supports the specification of order-irrelevant sets of subelements (`all`), the definition of elements with empty content (`nil`), and a finer specification of cardinality constraints.

It is natural and unavoidable that both data and schemas continuously change. Systems must be frequently adapted to real-world changes, new functionalities must be introduced, new data types must be processed. Complex structures can be defined even in not fully specified contexts, that are subsequently refined, design errors occurring in a schema need to be fixed. Commercial alliances change and expand. Moreover, in many contexts data representation format and domain-specific schemas are being specified in XML. Before a proposal can officially be adopted as a standard, however, different versions are produced and the developed instances continuously adapted. There is therefore a strong need to evolve XML Schemas and to propagate the effects on documents.

Schema updates have several consequences. First of all, documents valid for the original schema are no more guaranteed to meet the constraints described by the evolved schema. In principle, these documents should be *revalidated* against the new schema. If some of these documents are no more valid, they should be *adapted* to the new schema. Moreover, if the documents to be revalidated are stored on different sites, their transfer cost should be considered in addition to the whole document revalidation cost. Apart from the impact on documents, schema evolution may require a revision of document access policies, such as access control policies or index organizations. Finally, the evolution also impacts programs working on documents whose structure is described by the schema.

In this paper we address the problem of XML Schema evolution. We first present a set of *atomic evolution primitives* to be applied to the basic components of a schema (i.e., elements and types), being them local or global. This set of primitives is complete, in that all required transformations can be expressed through a sequence of primitives in the set. Moreover, the primitives are ensured to transform a consistent schema into a consistent schema. Then, a set of *high level evolution primitives* is devised as well, that can be expressed as a sequence of atomic primitives but that correspond to frequent evolution needs and allow to express complex changes in a more compact way. These primitives (both the atomic and the high level ones) are made available to the user through a graphical interface that we are currently finalizing.

The impact of the devised evolution primitives on documents known to be valid for the original schema is then investigated. Many documents can be associated with the same schema and validation cost is known to be high [12]. Moreover, documents can be stored in outsourced XML databases and thus the revalidation process tends to be highly distributed. A *brute-force* revalidation should thus be avoided and a more sophisticated solution should be devised. The basic idea is to keep track of the updates made to the schema and to identify the portions of the schema that, because of these updates, require a revalidation. The document portions affected by those updates can then be identified and revalidated, thus avoiding a costly revalidation of the whole documents. Since a schema evolution process may involve several updates, document revalidation is postponed to the end of the process. Thus, our approach aims at identifying the parts of documents to be revalidated after a certain number of updates have occurred on the schema.

In the remainder of the paper, Section 2 surveys related work. Section 3 introduces the adopted XML Schema representation. Section 4 presents the evolution primitives and Section 5 discusses the impact on documents. Finally, Section 6 concludes.

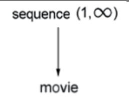
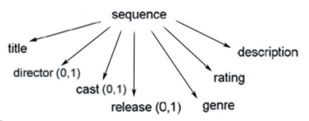
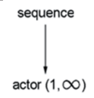
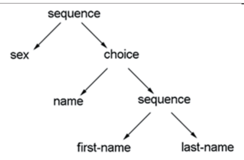
$\mathcal{EN}_G = \{description, movies\}, \mathcal{T} = \mathcal{TT} \cup \mathcal{AT}, \{personType\} \subseteq \mathcal{TT}, \mathcal{AT} = \{t1, t2, t3\}$ $\mathcal{R}_G(movies) = t3, \mathcal{R}_G(description) = string$			
$\rho(t3)$	$movie$		$movie \mapsto t2$
$\rho(t2)$	$title$ $director$ $cast$ $release$ $genre$ $rating$ $description$	 sequence title director (0,1) cast (0,1) release (0,1) genre rating description	$title \mapsto string$ $director \mapsto personType$ $cast \mapsto t1$ $release\text{-}\mapsto date$ $genre \mapsto string$ $rating \mapsto integer$ $description \mapsto string$
$\rho(t1)$	$actor$		$actor \mapsto personType$
$\rho(personType)$	sex $name$ $first\text{-}name$ $last\text{-}name$	 sequence sex choice name sequence first-name last-name	$sex \mapsto string$ $name \mapsto string$ $first\text{-}name \mapsto string$ $last\text{-}name \mapsto string$

Table 1: Movie schema representation

2. RELATED WORK

The need for XML schema evolution mechanisms has been advocated by Tan and Goh [11] 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 [5] 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 [4] 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.

Schema evolution and its consequences on instances has been thoroughly investigated in object-oriented databases [2, 8]. Though object-oriented schemas bring some similarities with XML Schemas, there are also fundamental differences that prevent to smoothly adapt techniques developed in that context to XML Schemas. When a class evolves, updates must be propagated to its instances to adapt them to the new definition. In this process inheritance plays a fundamental role. The high XML flexibility make these problems more subtle. Moreover, the XML Schema inheritance notion simply is a support for the user in easily composing schema taking advantage of modularity and declaration reuse, but has no impact on element validity.

The problem of document revalidation is investigated in [12]. 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, 3, 6]. 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 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 SCHEMA REPRESENTATION

In this section we introduce our representation for XML Schemas, that extends the one proposed in [12] to our context. $\mathcal{EN}$ 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. A simple type can be an XML Schema native type in the set τ_N or can be derived through `restriction`, `list`, and `union`. Each simple type is characterized by a set of *facets* allowing to state constraints on its legal values. The set of simple types is thus inductively defined as follows: native types in τ_N (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 facet applicable on t, `restrict`(τ, f) is a simple type.

Complex Types. The structure of a complex type 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. A 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 the structure of a complex type 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 represents the attribute `MinOccurs` and max represents the

	Insertion	Modification	Deletion
Simple Type	$insert\_glob\_simple\_type^*$ $insert\_new\_member\_type^*$	$change\_restriction$ $change\_base\_type$ $rename\_type^*$ $change\_member\_type$ $global\_to\_local^*$ $local\_to\_global^*$	$remove\_type^*$ $remove\_member\_type^*$
Complex Type	$insert\_glob\_complex\_type^*$ $insert\_local\_elem$ $insert\_ref\_elem$ $insert\_operator$	$rename\_local\_elem$ $rename\_global\_type^*$ $change\_type\_local\_elem$ $change\_cardinality$ $change\_operator$ $global\_to\_local^*$ $local\_to\_global^*$	$remove\_element$ $remove\_operator$ $remove\_substructure$ $remove\_type^*$
Element	$insert\_glob\_elem$	$rename\_glob\_elem^*$ $change\_type\_glob\_elem$ $ref\_to\_local^*$ $local\_to\_ref^*$	$remove\_glob\_elem^*$

Table 2: Classification of the evolution primitives

attribute `MaxOccurs`. The default value $(1,1)$ is not shown in our graphical representation. Let $root(T)$ be the root of tree T, $\varphi(T)$ denote the label of T root, and $\varphi_{|_i}(v)$, $i = 1, 2$, denote the i-th component of node v label. The *structure of a complex type* is a tree T defined on the set of labels $(\mathcal{EN} \cup \mathcal{OP}) \times \Gamma$ for which:

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

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. XML Schemas, unlike DTDs, allow an element to have different types depending on its context; however, 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 $(\mathcal{EN}_G, \mathcal{T}, \rho, \mathcal{R}_G)$, where:

- $\mathcal{EN}_G \subseteq \mathcal{EN}$ 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 type $\tau \in \mathcal{T}$ with its declaration, that is:
  - if τ is a simple type, $\rho(\tau) \in \tau_N \cup \{\texttt{restrict}(\tau_1, f), \texttt{list}(\tau_1), \texttt{union}(\tau_1 \ldots \tau_N) \mid \tau_1, \ldots, \tau_n$ simple types$\}$;
  - if τ is a complex type, $\rho(\tau)$= $(\mathcal{EN}_\tau, S_\tau, R_\tau)$, where: $\mathcal{EN}_\tau \subseteq \mathcal{EN}$ is the set of local element names for type τ, S_τ is the structure declared for type τ, $R_\tau : \mathcal{EN}_\tau \rightarrow \mathcal{T}$ is a total function assigning to each local element of τ the corresponding type;
- $\mathcal{R}_G : \mathcal{EN}_G \rightarrow \mathcal{T}$ is a total function assigning to each global element the corresponding type.

EXAMPLE 1. *Table 1 shows the representation of our reference movie schema example. The first row reports the set of global element names, the set of type names, and function $\mathcal{R}_G$ that associates each global element with the corresponding type. Then, for each complex type τ, its definition ρ_τ is provided, specifically the names of local subelements in $\mathcal{EN}_\tau$ are listed, together with the type structure S_τ and function R_τ that associates each local element name with the corresponding type.* ○

4. SCHEMA EVOLUTION PRIMITIVES

In this section we discuss atomic primitives and high level ones.

4.1 Atomic Primitives

Three categories of 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 three sub-categories: structural, re-labelling, and migration modifications. Structural modifications allow to modify the type of a (sub)element and its cardinality constraints. Re-labelling modifications allow to change the name of an element/type. Migration modifications include: moving a subelement from an element to another one and transforming a local type/element to a global type/element (and viceversa). These categories should be complemented with semantic modifications, that is, the possibility to use the same element (with the same structure) to represent a different kind of data. Such updates are hard to model and require the use of ontologies associating each term with the corresponding meaning.

Table 2 reports the evolution primitives relying on the proposed classification. The meaning of the * near some of the operators will be clarified in Section 5. For simple types the operators are further specialized to handle `restriction`, `list`, and `union` simple types. Because of space limitations the operators together with their formal specifications are discussed in [9].

EXAMPLE 2. *Suppose to evolve the XML Schema S in Table 1, obtaining the schema in Table 3, as follows:*

- *element `description` becomes optional in the definition of type $t2$; this is realized through the primitive $change\_cardinality((0, 1), 7, t2, S)$; since 7 is the position of element `description` in type $t2$ structure;*
- *a new `stage-name` element is added to the `personType` structure, this change is realized through the primitive $insert\_local\_element(stage\text{-}name, (1, 1), string, (4, 3), personType, S)$, where $(4, 3)$ denotes the position in the structure the new element is to be inserted;*
- *element `rating` is deleted by $remove\_element(6, t2, S)$.* ○

PROPOSITION 1. *Let $\mathcal{EP}$ be the set of evolution primitives summarized in Table 2.*

- *$\mathcal{EP}$ is sound: each primitive applied to a consistent schema produces a consistent schema.*
- *$\mathcal{EP}$ is complete: each schema can be generated starting from the empty schema by applying a sequence of primitives and for each schema a sequence of primitives exist transforming it in the empty schema.* □

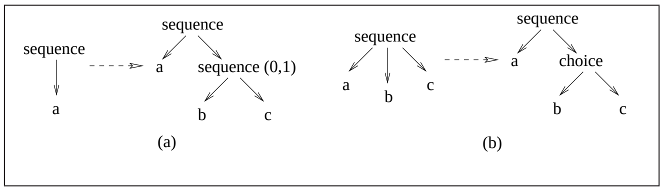


Figure 1: High level primitives

4.2 High Level Primitives

Atomic primitives can be composed in high level primitives in order to express in a more compact way more complex updates corresponding to common evolution needs. Their applications allow to perform sequences of atomic primitives as a single update. Since they are realized through atomic primitives the consistency of the resulting schema is guaranteed. Because of space limitations, we cannot discuss all devised high-level primitives, which mainly include primitives for inserting, moving, changing whole substructures rather than single elements.

EXAMPLE 3. *Suppose that starting from the structure in the left-hand side of Figure 1(a), we wish to add an optional sequence of elements* `b` *and* `c`*. This can be obtained by the atomic primitives:* $insert\_local\_element(b,(1,1),\tau_b,(0,2),t,S)$, $insert\_operator(sequence,(0,1),(2,0),t,S)$, $insert\_local\_element(c,(1,1),\tau_c,(2,2),t,S)$.
Alternatively, it can be obtained by the high level primitive: $insert\_substruct(T,(0,2),t,S))$*, where* T *represents the structure of the tree inserted in the right-hand side of Figure 1(a).*

Suppose moreover that starting from the structure in Figure 1(b), we wish to collapse the `b` *and* `c` *elements under a* `choice` *operator (as shown in the figure). This can be obtained through the atomic primitives:* $insert\_operator(choice,(1,1),(2,0),t,S)$, $change\_position(4,t,(2,2),S)$*. Alternatively, it can be obtained by the primitive* $collapse\_substruct(choice,t,0,2,3,S)$. ○

5. IMPACT ON VALIDITY

Consider an XML Schema and a set of XML documents known to be valid for the schema. When the schema evolves, documents are no longer ensured to be valid for the new schema and, in principle, should be revalidated. However, taking into account that updates usually affect only few schema elements and that an update does not necessarily compromise the document validity whole document revalidation can be avoided and validity checks can be restricted to the needed elements only. Well-formedness of documents need not to be checked, since documents are known to be well-formed. Moreover, since documents are known to be valid for the original schema and the applied evolution primitive is known, validity checks can be considerably simplified.

The definition of a type τ corresponds to a grammar that defines a set of documents $L(\tau)$: each instance in $L(t)$ is valid for τ. A function $validPreserving$ [12] can be defined, that, taken two type definitions τ and τ', returns $\mathcal{OK}$ if all valid instances for τ are also valid for τ', it returns $\mathcal{KO}$ if τ and τ' share no valid instances, and it returns $\mathcal{MAYBE}$ if neither disjointness nor inclusion holds.

$$validPreserving(\tau,\tau') = \begin{cases} \mathcal{OK} & \text{if } L(\tau)\subseteq L(\tau') \\ \mathcal{MAYBE} & \text{if } L(\tau)\not\subseteq L(\tau') \wedge L(\tau)\cap L(\tau')\neq\emptyset \\ \mathcal{KO} & \text{if } L(\tau)\cap L(\tau')=\emptyset \end{cases}$$

We want to detect whether an update does not compromise validity by simply taking into account the invoked primitives, its parameters, and, if needed, by a quick analysis of the updated type. We first investigate which evolution primitives are known not to compromise validity. Then, we propose a type labelling process that allows us to keep track of the document portions whose validity might have been compromised.

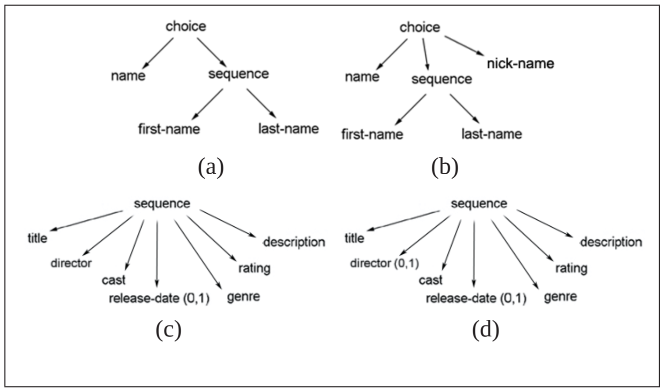


Figure 2: Validity preserving $insert\_local\_element$ (a,b) and $change\_cardinality$ (c,d)

5.1 Validity Preserving Primitives

Primitives inserting a new type definition or a new global element cannot compromise document validity. Similarly, the removal of a type/global element does not impact validity. The inserted/removed types/elements are indeed not yet/no more used in the schema (otherwise the update would not have been applicable). Primitives like $local\_to\_global$, $global\_to\_local$, $rename\_global\_type$, that reorganize the schema but do not alter the constraints imposed on documents, cannot affect validity. Thus, primitives marked with a star in Table 2 are validity preserving.

Consider now changes in simple type definitions. A relationship $\tau_1 \subseteq^s \tau_2$ exists among simple types modelling the fact that type τ_1 legal values are also values of type τ_2. Primitive types can be considered as restrictions of type `anySimpleType` and any other simple type is obtained through `restriction`, `list`, or `union` from a simple type. Derivation through `restriction` restricts the base type legal value set, while `list` and `union` extends it. Changes in a simple type definition can lead to modifications in the $\subseteq^s$ relationship, easily detected by keeping this relationship stored as a graph. Moreover, we can easily check whether changing the type associated with an element in the schema from a simple type τ to a new simple type τ' may impact document validity: if $\tau \subseteq^s \tau'$ no check is needed since the validity set is being extended.

We now turn to discuss some primitives which are known to have no impact on validity under certain conditions. We refer the reader to [9] for the discussion of primitives we do not consider here.

Subelement Insertion. This update is realized by the primitive invocation $insert\_local\_element(l,(min,max),t,(p,j),t_s,S)$. If the inserted element is optional (i.e., $min = 0$), document validity is not compromised. Moreover, if the element is added in a complex type as a child of a `choice` operator, then, independently from the cardinality, there is no impact on document validity. As an example, Figure 2(a,b) shows the effect of the primitive $insert\_local\_element(nick\text{-}name,(1,1),t,(0,3),t_s,S)$ inserting a new subelement in type t_s. Since the element is added as a subelement in a `choice`, document validity is not impacted.

Cardinality Change. This update is realized through the primitive invocation $change\_cardinality((min',max'),p,t_s,S)$ Let (min,max) be the cardinality constraints associated with the node whose position is p in type t_s structure. If the update extends the allowed cardinalities, that is, if $min' \leq min$ and $max \leq max'$,

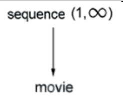
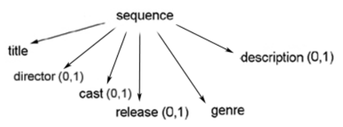
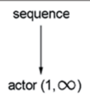
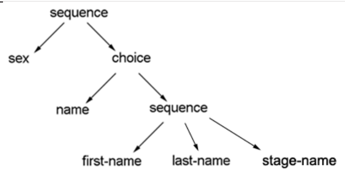
$\mathcal{EN}_G = \{description, movies\}, \mathcal{T} = \mathcal{TT} \cup \mathcal{AT}, \{personType, ratingType\} \subseteq \mathcal{TT}, \mathcal{AT} = \{t1, t2, t3\}$ $\mathcal{R}_G(movies) = t3, \mathcal{R}_G(description) = string$			
$\rho(t3)$	$movie$		$movie \mapsto t2$
$\rho(t2)$	$title$ $director$ $cast$ $release$ $genre$ $description$	 sequence title director (0,1) cast (0,1) release (0,1) genre description (0,1)	$title \mapsto string$ $director \mapsto personType$ $cast \mapsto t1$ $release \mapsto date$ $genre \mapsto string$ $description \mapsto string$
$\rho(t1)$	$actor$		$actor \mapsto personType$
$\rho(personType)$	sex $name$ $first\text{-}name$ $last\text{-}name$ $stage\text{-}name$	 sequence sex choice name sequence first-name last-name stage-name	$sex \mapsto string$ $name \mapsto string$ $first\text{-}name \mapsto string$ $last\text{-}name \mapsto string$ $stage\text{-}name \mapsto string$

Table 3: Evolved movie schema

then document validity is not compromised. As an example, Figure 2(c,d) shows the effect of the primitive $change\_cardinality$ $((0,1), 2, t_s, S)$ that makes optional the `director` subelement in type t_s. Since the cardinality constraints are weakened, all valid documents remain valid.

Substructure Insertion. This update is realized through the high-level primitive invocation $insert\_substruct(T, (p,j), t_s, S)$. If the substructure T is inserted under a specified node (the denoted position is (p,j) with $j > 1$), the case is analogous to subelement insertion. Thus, if T is optional (i.e., $\varphi_{|_2}(root(T)) = (min, max)$ and $min \leq 1$), it is inserted under a `choice`, or it can validate an `emptyContent` element, the primitive is validity preserving. By contrast, if the substructure T is inserted over a specified node (the denoted position is (p,j) with $j = 0$) the primitive is validity preserving if T is optional, if its root is labelled by `choice` (i.e., $\varphi_{|_1}(root(T)) =$ `CHOICE`), or if all T subtrees either are optional or validate `emptyContent` elements. Referring to the update of Example 3, the call of the $insert\_substruct$ primitive is validity preserving, since the entire inserted substructure is optional (its root cardinality is (0,1)). Note that not all the atomic primitives corresponding to that primitive are validity preserving, as stated by the following proposition, motivating the need to devise validity preservation conditions also for high level primitives.

PROPOSITION 2. *Let P be a high level primitive defined as the sequence of atomic primitives $AP_1, \ldots, AP_n$. If $\forall i = 1, \ldots, n$, AP_i is validity preserving then P is guaranteed to be validity preserving but the converse is not true.* □

5.2 Type Graph Labelling

Relying on the analysis of the impact on validity of each single primitive, we develop a process allowing the identification of the types τ in schema S' (the evolved schema obtained through a sequence of evolution primitives applied on the original schema) that do not describe any more instances valid for S (the original schema). The process consists of 3 phases.

In the *setup phase* the whole schema is represented through a directed graph, named *type graph*, by adding as child of each element the tree corresponding to the structure of its type. The type graph of our reference `movie` schema is depicted in Figure 3.

In the *annotation phase*, a label in $\{\mathcal{OK}, \mathcal{KO}, \mathcal{MAYBE}\}$ is associated with each node corresponding to a type in the graph for each evolution primitive of the sequence. For each single primitive, we have devised the labels to associate with the corresponding types depending on the primitive parameters, as an extension of what discussed in the previous section. Initially, all types are labelled $\mathcal{OK}$. Each primitive is considered in turn. Depending on the applied primitive, the label of some types can be changed to $\mathcal{MAYBE}$ or $\mathcal{KO}$. When considering subsequent primitives, a $\mathcal{MAYBE}$ label can be changed to $\mathcal{KO}$.

The last *propagation phase*, which is executed at the end of the schema evolution process, consists in propagating the annotation labels to ancestor types as follows. If a subelement of a type definition τ is associated with a type τ' labelled with $\mathcal{KO}$, if the element is not optional and going up in the graph neither a `choice` nor an operator with min cardinality equal to zero is encountered, type τ is labelled $\mathcal{KO}$ as well, otherwise type τ is labelled $\mathcal{MAYBE}$.

Annotations are then exploited for document revalidation as follows. First of all, we need to identify which elements in the documents are associated with τ through function $get\_path(\tau)$. This function is defined on each type τ present in the schema and returns the XPath expressions of elements whose type is τ. Referring to the `movie` schema in Figure 3, for instance, $get\_path(personType) = \{/movies/movie/cast/actor, /movies/movie/director\}$. Information expressed through type labels are then used as follows.

- If a type is labelled $\mathcal{OK}$, this means that it has not been modified or the update is validity preserving; in both cases $L(\tau) \subseteq L(\tau')$ and no checks are needed.
- If a type τ is labelled $\mathcal{KO}$, this means that its update compromised the validity of all documents valid for the previous corresponding type τ', that is, $L(\tau) \cap L(\tau') = \emptyset$; we can thus simply check if the document contains elements associated with type τ by inspecting in the documents the contents of paths in $get\_path(\tau)$. If one of such elements exists, the document is not valid.

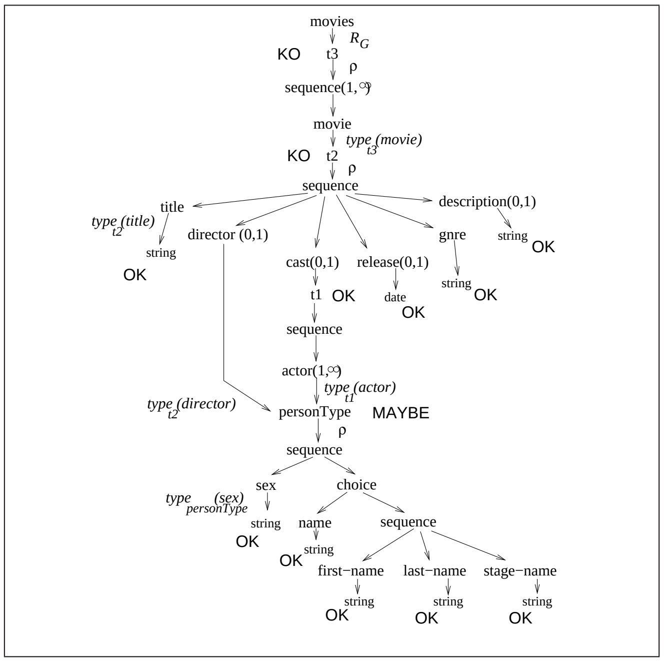


Figure 3: Labelled type graph

- If a type τ is labelled $\mathcal{MAYBE}$, this means that the update may have compromised the validity of some documents, that is, $L(\tau) \not\subseteq L(\tau') \wedge L(\tau) \cap L(\tau') \neq \emptyset$. In this case, again through function get_path, the elements associated with the type must be retrieved and their subtrees must be revalidated.

EXAMPLE 4. *We discuss the impact on validity of the evolution operations of Example 2 (consider the graph in Figure 3).*

- *The $change\_cardinality$ update is recognized as an extension of the set of valid instances for type $t2$. The corresponding $t2$ label is thus not changed.*
- *The $insert\_local\_element$ update is the insertion of a mandatory subelement in a `sequence`. The document validity with respect to the `sequence` is compromised, however the `sequence` is under a `choice` thus some documents may not be affected by this update. Type `personType` is thus labelled $\mathcal{MAYBE}$.*
- *The $remove\_element$ primitive removes a mandatory element from a `sequence` and no `choice` ancestor nor an ancestor operator with `minOccurs`=0 exist. Thus, type $t2$ is labelled $\mathcal{KO}$.*

Type labels are now propagated. The only type labelled by $\mathcal{KO}$ is $t2$ which is associated only with the `movie` subelement of type $t3$. Since element `movie` is not optional and going up the type definition neither `choice` nor operators with `minOccurs`=0 are encountered, type $t3$ is labelled $\mathcal{KO}$ as well. ○

PROPOSITION 3. *The annotation process is sound, that is, all the document portions that can have been invalidated by the schema updates are detected.* □

Note that the annotation process allows to identify the minimal document portions potentially invalidated by a schema update, if single update primitives are considered. When processing update sequences, however, minimality is lost since the *net effect* of different updates should be considered (e.g., the insertion of a mandatory element and its subsequent deletion, which have a null net effect).

6. CONCLUSIONS AND FUTURE WORK

In this paper we have investigated the problem of XML Schema evolution, by proposing a set of evolution primitives and analyzing the impact of such primitives on the validity of XML documents known to be valid for the original schema.

We are extending the work reported in this paper along several directions. First of all, we are finalizing the implementation of a graphical interface providing the devised schema evolution primitives. The minimal revalidation approach based on graph labelling is being implemented as well, and the performance gain over the brute-force revalidation approach would be evaluated over real XML document collections. The tool under development offers support for document adaptation as well by proposing the user a restructuring structure when it can be devised and helping her in specifying such a structure when user intervention is required. It will finally allow the specification of customized high-level primitives as composition of atomic ones.

The problem of adaptation of no more valid documents to the new schema has being addressed as well, which involves subtleties related both to the kind of update performed and to the structure of the updated type. Several updates require the detection of the *minimal substructure* for an element whose insertion/deletion is required to make the documents valid for the evolved schema. Our approach to document adaptation is based on the use of *restructuring structures*, that are structures like the ones introduced in Section 3 in which labels can also be Δ^{ϵ}_{l}, Δ^{l}_{ϵ}, and $\Delta^{l_o}_{l_n}$, with $l, l_n, l_o \in \mathcal{EN}$. These structures allow the specification of the minimal modifications to be performed on documents invalidated by a schema update. These structures are automatically inferred from the schema update whenever possible, they are provided by the user otherwise. The adaptation process will occur during the revalidation process and the idea is that to validate the special subelement Δ^{ϵ}_{l} element l should be inserted. Similarly, to validate the special subelements Δ^{l}_{ϵ} and $\Delta^{l_o}_{l_n}$ element l should be deleted and element l_o should be renamed to l_n, respectively.

Finally, as a longer term goal, we are also interested in investigating the impact of schema evolution on access control policies, indexing structures, and programs working on the XML documents whose schema is being evolved.

7. REFERENCES

[1] A. Balmin, et al. Incremental Validation of XML Documents. *ACM TODS* 29(4): 710–751, 2004.
[2] J. Banerjee, W. Kim, H. Kim, and H. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD*, 311–322, 1987.
[3] D. Barbosa, et al. Efficient Incremental Validation of XML Documents. *ICDE*, 671–682, 2004.
[4] E. Bertino, et al. Evolving a Set of DTDs according to a Dynamic Set of XML Documents. *EDBT Workshops*, LNCS 2490, 45–66, 2002.
[5] G.J. Bex, F. Neven, and J. Van den Bussche. DTDs versus XML Schema: A Practical Study. *WebDB*, 79–84, 2004.
[6] B. Bouchou and M.H. Ferrari Alves. Updates and Incremental Validation of XML Documents. *DBPL*, 216–232, 2003.
[7] B. Choi. What are Real DTDs Like? *WebDB*, 43–48, 2002.
[8] F. Ferrandina, et al. Schema and Database Evolution in the O2 Object Database System. *VLDB*, 170–181, 1995.
[9] G. Guerrini, M. Mesiti, and D. Rossi. XML Schema Evolution, TR Universit'a di Genova, 2005.
[10] D. K. Kramer and E. A. Rundensteiner. Xem: XML Evolution Management. *RIDE-DM*, 103–110, 2001.
[11] M. B. L. Tan and A. Goh. Keeping Pace with Evolving XML-Based Specifications. *EDBT Workshops*, LNCS 3268, 280–288, 2004.
[12] M. Raghavachari and O. Shmueli. Efficient Schema-Based Revalidation of XML. *EDBT*, 639–657, 2004.
[13] W3C. Extensible Markup Language 1.0, 1998.
[14] W3C. XML Schema Part 0: Primer, 2001.