Impact of XML Schema Evolution

Pierre Genevès, Nabil Layaïda, Vincent Quint

Impact of XML Schema Evolution

Pierre Genevès
CNRS
Nabil Layaïda
INRIA
and
Vincent Quint
INRIA

We consider the problem of XML Schema evolution. In the ever-changing context of the web, XML schemas continuously change in order to cope with the natural evolution of entities they describe. Schema changes have important consequences. First, existing documents valid with respect to the original schema are no longer guaranteed to fulfill the constraints described by the evolved schema. Second, the evolution also impacts programs manipulating documents whose structure is described by the original schema.

We propose a unifying framework for determining the effects of XML Schema evolution both on the validity of documents and on queries. The system is very powerful in analyzing various scenarios in which forward/backward compatibility of schemas is broken, and in which the result of a query may not be anymore what was expected. Specifically, the system offers a predicate language which allows one to formulate properties related to schema evolution. The system then relies on exact reasoning techniques to perform a fine-grained analysis. This yields either a formal proof of the property or a counter-example that can be used for debugging purposes. The system has been fully implemented and tested with real-world use cases, in particular with the main standard document formats used on the web, as defined by W3C. The system identifies precisely compatibility relations between document formats. In case these relations do not hold, the system can identify queries that must be reformulated in order to produce the expected results across successive schema versions.

Categories and Subject Descriptors: D.3.4 [**Software**]: Programming Languages—*Processors*; D.2.4 [**Software**]: Engineering—*Software/Program Verification*

General Terms: Languages, Standardization, Verification

Additional Key Words and Phrases: XML, Schemas, Queries, Web Document Formats, Schema Evolution

1. INTRODUCTION

XML is now commonplace on the web and in many information systems where it is used for representing all kinds of information resources, ranging from simple text documents such as RSS or Atom feeds to highly structured databases. In these dynamic environments, not only data are changing steadily but their schemas also get modified to cope with the evolution of the real world entities they describe.

Schema changes raise the issue of data consistency. Existing documents and data that were valid with a certain version of a schema may become invalid on a new version of the schema (forward incompatibility). Conversely, new documents created with the latest version of a schema may be invalid on some previous versions (backward incompatibility).

In addition, schemas may be written in different languages, such as DTD, XML Schema, or Relax-NG, to name only the most popular ones. And it is common practice to describe the same structure, or new versions of a structure, in different schema languages. Document formats developed by W3C provide a variety of examples: XHTML 1.0 has both DTDs and XML Schemas, while XHTML 2.0 has a Relax-NG definition; the schema for SVG Tiny 1.1 is a DTD, while version 1.2 is written in Relax-NG; MathML 1.01 has a DTD, MathML 2.0 has both a DTD and an XML Schema, and MathML 3.0 is developed with a Relax-NG schema and also published with a DTD and an XML Schema. An issue then is to make sure that schemas written in different languages are equivalent, *i.e.* they describe the same structure, possibly with some differences due to the expressivity of the language [Murata et al. 2005]. Another issue is to clearly identify the differences between two versions of the same schema expressed in different languages. Moreover, the issues of forward and backward compatibility of instances obviously remain when schema languages change from a version to another.

Validation, and then compatibility, is not the only purpose of a schema. Validation is usually the first step for safe processing of documents and data. It makes sure that documents and data are structured as expected and can then be processed safely. The next step is to actually access and select the various parts to be handled in each phase of an application. For this, query languages play a key role. As an example, when transforming a document with XSL, XPath queries are paramount to locate in the original document the data to be produced in the transformed document.

Queries are affected by schema evolutions. The structures they return may change depending on the version of the schema used by a document. When changing schema, a query may return nothing, or something different from what was expected, and obviously further processing based on this query is at risk.

These observations highlight the need for evaluating precisely and safely the impact of schema evolutions on existing and future instances of documents and data. They also show that it is important for software engineers to precisely know what parts of a processing chain have to be updated when schemas change. In this paper we focus on the XPath query language which is used in many situations while processing XML documents and data. The XSL transformation language was already mentioned, but XPath is also present in XLink and XQuery for instance.

A part of this work concerning the impact of schema changes on XPath queries was presented at the ACM International Conference on Functional Programming (ICFP), 2009, [Genevès et al. 2009]. The present article aims at covering the more general issue of schema evolution by taking into account the impact on the validity of documents as well. In particular, we identify criteria for the evolution of standard XML Schemas. We present a framework for checking these criteria with the schemas specifying the main standard documents formats used on the web, as defined by W3C (see Section 5).

Outline

We first introduce the framework from a high-level perspective in Section 2: we describe how the whole system is assembled, and which XML schemas and queries are supported. In Section 3, we provide a more in-depth understanding of the

underlying logic on which the system is built; in particular we explain how XML constructs are mapped to this logical representation. Based on this logical encodings, Section 4 introduces a predicate language specifically designed for assessing the impact of schema evolutions. The following sections respectively focus on applying the framework for studying the impact of schema evolutions on the validity of documents (Section 5) and on queries (Section 6). The full implementation of the system is presented in Section 7. Finally, we discuss related work in Section 8 before concluding in Section 9.

2. ANALYSIS FRAMEWORK

The main contribution of this paper is a unifying framework that allows the automatic verification of properties related to XML schema evolution and its impact on the validity of documents and on queries. In particular, it offers the possibility of checking fine-grained properties of the behavior of queries with respect to successive versions of a given schema. The system can be used for checking relations between schemas and whether schema evolutions require a particular query to be updated. Whenever schema evolutions may induce query malfunctions, the system is able to generate annotated XML documents that exemplify bugs, with the goal of helping the programmer to understand and properly overcome undesired effects of schema evolutions.

The system relies on a predicate language (presented in Section 4) specifically designed for studying schema and query compatibility issues when schemas evolve. Specifically, predicates allow characterizing in a precise manner nodes subject to evolution. For instance, predicates allow to distinguish new nodes selected by the query after a schema change from new nodes that appear in the modified schema. Predicates also allow to describe nodes that appear in new regions of a schema compared to its original version, or even in a new context described by a particular XPath expression. Predicates, together with the composition language provided in the system allow to express and analyze complex settings.

The system has been fully implemented [Genevès and Layaïda 2009] and is outlined in Figure 1. It is composed of a parser for reading the text file description of the problem (which in turn uses specific parsers for schemas, queries, logical formulas, and predicates), compilers for translating schemas and queries into their logical representations, a solver for checking satisfiability of logical formulas, and a counter example XML tree generator (described in [Genevès et al. 2008]).

We first introduce the data model we consider for XML documents, schemas and queries.

2.1 XML Trees with Attributes

An XML document is considered as a finite tree of unbounded depth and arity, with two kinds of nodes respectively named elements and attributes. In such a tree, an element may have any number of children elements, and may carry zero, one or more attributes. Attributes are leaves. Elements are ordered whereas attributes are not, as illustrated on Figure 4. In this paper, we focus on the nested structure of elements and attributes, and ignore XML data values.

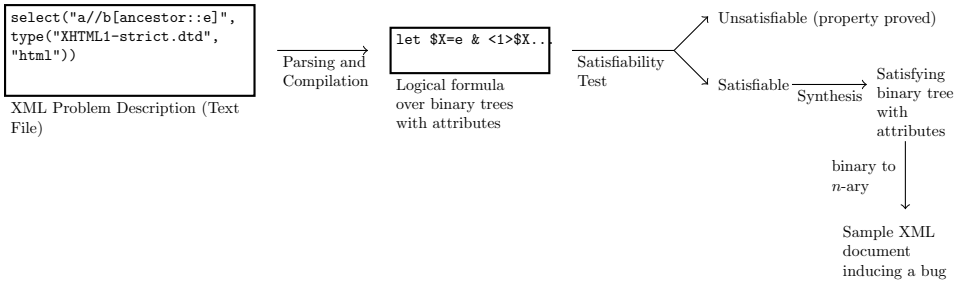


Fig. 1. Framework Overview.

.

2.2 Type Constraints

Our tree type expressions capture most of the schemas in use today either written using DTD, XML Schema, Relax NG, etc. Users may thus define constraints over XML documents with the language of their choice, and, more importantly, they may refer to most existing schemas for use with the system. Instead of having one parser/compiler per schema language, we rely on a common intermediate language in which all these languages are compiled. For the intermediate language we consider the standard class of regular tree grammars, commonly found in the literature [Hosoya et al. 2005], to which we have added the support of constraints over XML attributes (whose efficiency is further discussed in section 3.3). In terms of expressive power, regular tree grammars support constraints over trees which are more expressive than local tree grammars (DTDs) and single-type tree grammars (XML schemas), capturing exactly the class of Relax NG schemas, and, more fundamentally finite tree automata (see [Murata et al. 2005] for a formal characterization of the respective expressive power of these languages). In practice, we have implemented parsers that produce this intermediate representation from a given DTD, XML Schema, or Relax NG schema. We have implemented one compiler from this representation into the logic. An advantage of this approach is that it is extensible: it is easy to know the supported features since (1) the intermediate language is well-characterized and made explicit, and (2) extending the system with new schema languages is easy since one does not need to implement new compilers into the logic (and prove soundness, completeness and polynomial-time translation), but rather simply express the new considered constraints in the intermediate language.

Specifically, our unifying internal representation for tree grammars is made of regular tree type expressions, extended with constraints over attributes. Assuming

a set of variables ranged over by x, we define a tree type expression as follows:

τ ::=		tree type expression
	$\emptyset$	empty set
	`()`	empty sequence
	$\tau \mid \tau$	disjunction
	τ, τ	concatenation
	l`(`a`)[`τ`]`	element definition
	x	variable
	`let` $\overline{x = \tau}$ `in` τ	binder

The `let` construct allows binding one or more variables to associated formulas. Since several variables can be bound at a time, the notation $\overline{x = \tau}$ is used for denoting a vector of variable bindings (possibly with mutual recursion).

We impose a usual restriction on the recursive use of variables: we allow unguarded (*i.e.* not enclosed by a label) recursive uses of variables, but restrict them to tail positions[1]. With that restriction, tree types expressions define regular tree languages. In addition, an element definition may involve simple attribute expressions that describe which attributes the defined element may (or may not) carry:

a ::=		attribute expression
	`()`	empty list
	$list \mid a$	disjunction
$list$ ::=		attribute list
	$list, list$	commutative concatenation
	l`?`	optional attribute
	l	required attribute
	$\neg l$	prohibited attribute

We use the usual semantics of regular tree types found in [Hosoya et al. 2005] and [Genevès et al. 2008].

2.3 Queries

The set of XPath expressions we consider is given by the syntax shown on Figure 2. The semantics of XPath expressions is described in [Clark and DeRose 1999], and more formally in [Wadler 2000]. We observed that, in practice, many XPath expressions contain syntactic sugars that can also fit into this fragment. Figure 3 presents how our XPath parser rewrites some commonly found XPath patterns into the fragment of Figure 2, where the notation $(axis{::}nt)^k$ stands for the composition of k successive path steps of the same form: $\underbrace{axis{::}nt/.../axis{::}nt}_{k \text{ steps}}$.

The next Section presents the logic underlying the predicate language.

3. LOGICAL SETTING

It is well-known that there exist bijective encodings between unranked trees (trees of unbounded arity) and binary trees [Thomas 1990]. Owing to these encodings

[1]For instance, "`let` $x = l$`(`a`)[`τ`],`x `|` `()` `in` x" is allowed.

query ::=		
	$/path$	absolute path
	$path$	relative path
	$query \mid query$	union
	$query \cap query$	intersection
path ::=		
	$path/path$	path composition
	$path[qualifier]$	qualified path
	$axis::nt$	step
qualifier ::=		
	$qualifier$ and $qualifier$	conjunction
	$qualifier$ or $qualifier$	disjunction
	not($qualifier$)	negation
	$path$	path
	$path/@nt$	attribute path
	$@nt$	attribute step
nt ::=		node test
	σ	node label
	$*$	any node label
axis ::=		tree navigation axis
	self | child | parent	
	descendant | ancestor	
	descendant-or-self	
	ancestor-or-self	
	following-sibling	
	preceding-sibling	
	following | preceding	

Fig. 2. XPath Expressions.

$$
\begin{aligned}
nt[\text{position}() = 1] &\rightsquigarrow nt[\text{not}(\text{preceding-sibling}::nt)] \\
nt[\text{position}() = \text{last}()] &\rightsquigarrow nt[\text{not}(\text{following-sibling}::nt)] \\
nt[\text{position}() = \underbrace{k}_{k>1}] &\rightsquigarrow nt[(\text{preceding-sibling}::nt)^{k-1}] \\
\text{count}(path) = 0 &\rightsquigarrow \text{not}(path) \\
\text{count}(path) > 0 &\rightsquigarrow path \\
\text{count}(nt) > \underbrace{k}_{k>0} &\rightsquigarrow nt/(\text{following-sibling}::nt)^{k}
\end{aligned}
$$

$$
\begin{aligned}
&\text{preceding-sibling}::*[\text{position}() = \text{last}() \text{ and } qualifier] \\
&\qquad \rightsquigarrow \text{preceding-sibling}::*[\text{not}(\text{preceding-sibling}::*) \text{ and } qualifier]
\end{aligned}
$$

Fig. 3. Syntactic Sugars and their Rewritings.

binary trees may be used instead of unranked trees without loss of generality. In the sequel, we rely on a simple "first-child & next-sibling" encoding of unranked trees. In this encoding, the first child of an element node is preserved in the binary tree representation, whereas siblings of this node are appended as right successors in the binary representation. Attributes are left unchanged by this encoding. For instance, Figure 5 presents how the sample tree of Figure 4 is mapped.

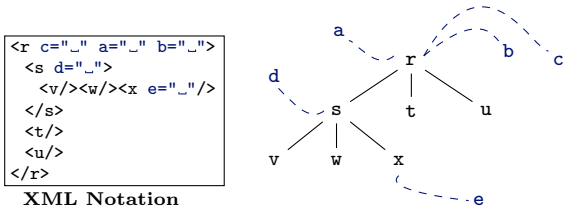


Fig. 4. Sample XML Tree with Attributes.

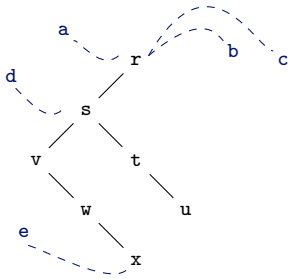


Fig. 5. Binary Encoding of Tree of Figure 4.

The logic we introduce below, used as the core of our framework, operates on such binary trees with attributes.

3.1 Logical Formulas

The concrete syntax of logical formulas is shown on Figure 6, where the meta-syntax $\langle X \rangle^{\oplus}$ means one or more occurences of X separated by commas. The user can directly encode formulas with this syntax in text files to be used with the system [Genevès and Layaïda 2009]. This concrete syntax is used as a single unifying notation throughout all the paper.

The semantics of logical formulas corresponds to the classical semantics of a μ-calculus interpreted over finite tree structures. A formula is satisfiable iff there exists a finite binary tree with attributes for which the formula holds at some node. This is formally defined in [Genevès et al. 2007], and we review it informally below through a series of examples.

There is a difference between an element name and an atomic proposition[2]: an element has one and only one element name, whereas it can satisfy multiple atomic propositions. We use atomic propositions to attach specific information to tree nodes, not related to their XML labeling. For example, the start context (a reserved atomic proposition) is used to mark the starting context nodes for evaluating XPath

[2]In practice, an atomic proposition must start with a "_".

φ ::=		formula
	`T`	true
	`F`	false
	l	element name
	p	atomic proposition
	`#`	start context
	φ `|` φ	disjunction
	φ `&` φ	conjunction
	φ `=>` φ	implication
	φ `<=>` φ	equivalence
	`(`φ`)`	parenthesized formula
	$\tilde{}\varphi$	negation
	`<`p`>`φ	existential modality
	`<`l`>T`	attribute named l
	`$X`	variable
	`let` $\langle$`$X` $= \varphi\rangle^{\oplus}$ `in` φ	binder for recursion
	predicate	predicate (See Section 4)
p ::=		program inside modalities
	`1`	first child
	`2`	next sibling
	`-1`	parent
	`-2`	previous sibling

Fig. 6. Concrete Syntax of Formulas.

expressions.

The logic uses modalities for navigating in binary trees. A modality `<`p`>`φ can be read as follows: "there exists a successor node by program p such that φ holds at this successor". As shown on Figure 6, a program p is simply one of the four basic programs $\{1, 2, -1, -2\}$. Program `1` allows navigating from a node down to its first successor, and program `2` allows navigating from a node down to its second successor. The logic also features converse programs `-1` and `-2` for navigating upward in binary trees, respectively from the first successor to its parent and from the second successor to its previous sibling. Table I gives some simple formulas using modalities for navigating in binary trees, together with sample satisfying trees, in binary and unranked tree representations.

The logic allows expressing recursion in trees through the recursive binder. For example the recursive formula:

```
let $X = b | <2>$X in $X
```

means that either the current node is named `b` or there is a sibling of the current node which is named `b`. For this purpose, the variable `$X` is bound to the subformula `b | <2>$X` which contains an occurence of `$X` (therefore defining the recursion). The scope of this binding is the subformula that follows the "`in`" symbol of the formula, that is `$X`. The entire formula can thus be seen as a compact recursive notation for a infinitely nested formula of the form:

```
b | <2>(b | <2>(b | <2>(...)))
```

Recursion allows expressing global properties. For instance, the recursive formula:

```
~ let $X = a | <1>$X | <2>$X in $X
```

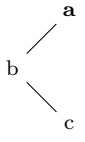
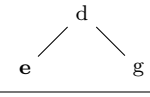
Sample Formula	Tree	XML
a & <1>b		<a><b/>
a & <1>(b & <2>c)		<a><b/><c/>
e & <-1>(d & <2>g)		<d><e/></d><g/>
f & <-2>(g & ~<2>T)	none	none

Table I. Sample Formulas and Satisfying Trees.

expresses the absence of nodes named `a` in the whole subtree of the current node (including the current node). Furthermore, the fixpoint operator makes possible to bind several variables at a time, which is specifically useful for expressing mutual recursion. For example, the mutually recursive formula:

```
let
  $X = (a & <2>$Y) | <1>$X | <2>$X,
  $Y = b | <2>$Y
in $X
```

asserts that there is a node somewhere in the subtree such that this node is named `a` and it has at least one sibling which is named `b`. Binding several variables at a time provides a very expressive yet succinct notation for expressing mutually recursive structural patterns (that are common in XML Schemas, for instance).

From a theoretical perspective, the recursive binder `let $X` $= \varphi$ `in` φ corresponds to the fixpoint operators of the μ-calculus. It is shown in [Genevès et al. 2007] that the least fixpoint and the greatest fixpoint operators of the μ-calculus coincide over finite tree structures, for a restricted class of formulas called *cycle-free* formulas.

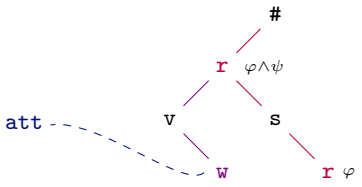
3.2 Queries

The logic is expressive enough to capture the set of XPath expressions presented in Section 2.3. For example, Figure 7 illustrates how the sample XPath expression:

```
child::r[child::w/@att]
```

is expressed in the logic. From a given context in an XML document, this expression selects all `r` child nodes which have at least one `w` child with an attribute `att`. Figure 7 shows how it is expressed in the logic, on the binary tree representation. The formula holds for `r` nodes which are selected by the expression. The first part of the formula, φ, corresponds to the step `child::r` which selects candidates `r` nodes. The second part, ψ, navigates downward in the subtrees of these candidate nodes to verify that they have at least one immediate `w` child with an attribute `att`.

This example illustrates the need for converse programs inside modalities. The translated XPath expression only uses forward axes (child and attribute), neverthe-



Translated Query: `child::r[child::w/@att]`

Translation:

$$\underbrace{\texttt{r \& (let \$X=<-1>\# | <-2>\$X)}}_{\varphi}\ \underbrace{\texttt{\& <1>let \$Y=w \& <att>T | <2>\$Y}}_{\psi}$$

Fig. 7. XPath Translation Example.

less both forward and backward modalities are required for its logical translation. Without converse programs we would have been unable to differentiate selected nodes from nodes whose existence is simply tested. More generally, properties must often be stated on both the ancestors and the descendants of the selected node. Equipping the logic with both forward and converse programs is therefore crucial. Logics without converse programs may only be used for solving XPath emptiness but cannot be used for solving other decision problems such as containment efficiently.

A systematic translation of XPath expressions into the logic is given in [Genevès et al. 2007]. In this paper, we extended it to deal with attributes. We implemented a compiler that takes any expression of the fragment of Figure 2 and computes its logical translation. With the help of this compiler, we extend the syntax of logical formulas with a logical predicate `select("`*query*`"`, φ). This predicate compiles the XPath expression *query* given as parameter into the logic, starting from a context that satisfies φ. The XPath expression to be given as parameter must match the syntax of the XPath fragment shown on Figure 2 (or Figure 3). In a similar manner, we introduce the predicate `exists("`*query*`"`, φ) which tests the existence of *query* from a context satisfying φ, in a qualifier-like manner (without moving to its result). Additionally, the predicate `select("`*query*`")` is introduced as a shortcut for `select("`*query*`",#)`, where `#` simply marks the initial context node of the XPath expression[3]. The predicate `exists("`*query*`")` is a shortcut for `exists("`*query*`",T)`. These syntactic extensions of the logic allow the user to easily embed XPath expressions and formulate decision problems out of them (like *e.g.* containment or any other boolean combination). In the next sections we explain how the framework allows combining queries with schema information for formulating problems.

3.3 Tree Types

Tree type expressions are compiled into the logic in two steps: the first stage translates them into binary tree type expressions, and the second step actually

[3]This mark is especially useful for comparing two or more XPath expressions from the same context.

compiles this intermediate representation into the logic. The translation procedure from tree type expressions to binary tree type expressions is well-known and detailed in [Genevès 2006]. The syntax of output expressions follows:

τ ::=		binary tree type expression
	$\emptyset$	empty set
	`()`	empty tree
	$\tau \mid \tau$	disjunction
	l`(`a`)[`x, x`]`	element definition
	`let` $\overline{x = \tau}$ `in` τ	binder

Attribute expressions are not concerned by this transformation to binary form: they are simply attached, unchanged, to new (binary) element definitions. Finally, binary tree type expressions are compiled into the logic. This translation step was introduced and proven correct in [Genevès et al. 2007]. Originally, the translation takes a tree type expression τ and returns the corresponding logical formula. Here, we extend it slightly but crucially: the logical translation of an expression τ is given by the function $\mathrm{tr}(\tau)^{\psi}_{\varphi}$ defined below, that takes additional arguments φ and ψ:

$$\begin{aligned}
\mathrm{tr}(\tau)^{\psi}_{\varphi} &\stackrel{\text{def}}{=} \mathtt{F} \quad \text{for} \quad \tau = \emptyset, \mathtt{()} \\
\mathrm{tr}(\tau_1 \mid \tau_2)^{\psi}_{\varphi} &\stackrel{\text{def}}{=} \mathrm{tr}(\tau_1)^{\psi}_{\varphi} \ \mathtt{|}\ \mathrm{tr}(\tau_2)^{\psi}_{\varphi} \\
\mathrm{tr}(l\mathtt{(}a\mathtt{)[}x_1, x_2\mathtt{]})^{\psi}_{\varphi} &\stackrel{\text{def}}{=} (l \ \mathtt{\&}\ \varphi \ \mathtt{\&}\ \mathrm{tra}(a) \ \mathtt{\&}\ s_1(x_1) \ \mathtt{\&}\ s_2(x_2)) \ \mathtt{|}\ \psi \\
\mathrm{tr}(\mathtt{let}\ \overline{x_i = \tau_i}\ \mathtt{in}\ \tau)^{\psi}_{\varphi} &\stackrel{\text{def}}{=} \mathtt{let}\ \overline{\mathtt{\$X}_i = \mathrm{tr}(\tau_i)^{\psi}_{\varphi}}\ \mathtt{in}\ \mathrm{tr}(\tau)^{\psi}_{\varphi}
\end{aligned}$$

The addition of φ and ψ (respectively in a new conjunction and a new disjunction) is a key element for the definition of predicates in Section 4. More precisely, this allows marking type sub-expressions so that they can be distinguished in predicates, as explained in Section 3.4. In addition, φ and ψ are either true, false, or simple atomic propositions. Thus, it is worth noticing that their addition does not affect the linear complexity of tree type translation. The function $s_{\cdot}(\cdot)$ describes the type for each successor:

$$s_p(x) \ = \ \begin{cases} \tilde{}\texttt{<}p\texttt{>T} & \textit{if } x \textit{ is bound to } \texttt{()} \\ \tilde{}\texttt{<}p\texttt{>T | <}p\texttt{>\$X} & \textit{if } \textit{nullable}(x) \\ \texttt{<}p\texttt{>\$X} & \textit{if not } \textit{nullable}(x) \end{cases}$$

according to the predicate $\mathit{nullable}(x)$ which indicates whether the type $T \neq \texttt{()}$ bound to x contains the empty tree.

The function $\mathrm{tra}(a)$ compiles attribute expressions associated with element defi-

nitions as follows:

$$\begin{aligned}
\mathrm{tra}(\texttt{()}) &\stackrel{\text{def}}{=} \mathrm{notothers}(\texttt{()}) \\
\mathrm{tra}(\mathit{list} \mid a) &\stackrel{\text{def}}{=} \mathrm{tra}(\mathit{list})\ \texttt{\&}\ \mathrm{notothers}(\mathit{list}) \\
\mathrm{tra}(\mathit{list}, \mathit{list}') &\stackrel{\text{def}}{=} \mathrm{tra}(\mathit{list})\ \texttt{\&}\ \mathrm{tra}(\mathit{list}') \\
\mathrm{tra}(l?) &\stackrel{\text{def}}{=} l\ \texttt{|}\ \tilde{}\, l \\
\mathrm{tra}(l) &\stackrel{\text{def}}{=} l \\
\mathrm{tra}(\neg l) &\stackrel{\text{def}}{=} \tilde{}\, l
\end{aligned}$$

In usual schemas (*e.g.* DTDs, XML Schemas) when no attribute is specified for a given element, it simply means no attribute is allowed for the defined element. This convention must be explicitly stated in the logic. This is the role of the function "notothers(*list*)" which returns the negated disjunction of all attributes not present in *list*. As a result, taking attributes into account comes at an extra-cost. The above translation appends a (potentially very large) formula in which all attributes occur, for each element definition. In practice, a placeholder atomic proposition is inserted until the full set of attributes involved in the problem formulation is known. When the whole formula has been parsed, placeholders are replaced by the conjunction of negated attributes they denote. This extra-cost can be observed in practice, and the system allows two modes of operations: with or without attributes[4]. Nevertheless the system is still capable of handling real world DTDs (such as the DTD of XHTML 1.0 Strict) with attributes. This is due to (1) the limited expressive power of languages such as DTD that do not allow for disjunction over attribute expressions (like "*list* | *a*"); and, more importantly, (2) the satisfiability-testing algorithm which is implemented using symbolic techniques [Genevès et al. 2008].

Tree type expressions form the common internal representation for a variety of XML schema definition languages. In practice, the logical translation of a tree type expression τ are obtained directly from a variety of formalisms for defining schemas, including DTD, XML Schema, and Relax NG. For this purpose, the syntax of logical formulas is extended with a predicate $\texttt{type}(\texttt{"}\cdot\texttt{"}, \cdot)$. The logical translation of an existing schema is returned by $\texttt{type}(\texttt{"}f\texttt{"}, l)$ where f is a file path to the schema file and l is the element name to be considered as the entry point (root) of the given schema. Any occurence of this predicate will parse the given schema, extract its internal tree type representation τ, compile it into the logic and return the logical formula $\mathrm{tr}(\tau)^{\mathtt{F}}_{\mathtt{T}}$.

3.4 Type Tagging

A tag (or "color") is introduced in the compilation of schemas with the purpose of marking all node types of a specific schema. A tag is simply a fresh atomic proposition passed as a parameter to the translation of a tree type expression. For example: $\mathrm{tr}(\tau)^{\mathtt{F}}_{\mathrm{xhtml}}$ is the logical translation of τ where each element definition is annotated with the atomic proposition "xhtml". With the help of tags, it becomes possible to refer to the element types in any context. For instance, one may formu-

[4]The optional argument "-attributes" must be supplied for attributes to be considered.

late $\mathrm{tr}(\tau)^{\mathtt{F}}_{\mathrm{xhtml}}$ | $\mathrm{tr}(\tau')^{\mathtt{F}}_{\mathrm{smil}}$ for denoting the union of all τ and τ' documents, while keeping a way to distinguish element types; even if some element names are shared by the two type expressions.

Tagging becomes even more useful for characterizing evolutions between successive versions of a single schema. In this setting, we need a way to distinguish nodes allowed by a newer schema version from nodes allowed by an older version. This distinction must not be based only on element names, but also on content models. Assume for instance that τ' is a newer version of schema τ. If we are interested in the set of trees allowed by τ' but not allowed by τ then we may formulate:

$$\mathrm{tr}(\tau')^{\mathtt{F}}_{\mathtt{T}}\ \mathtt{\&\tilde{}}\ \mathrm{tr}(\tau)^{\mathtt{F}}_{\mathtt{T}}$$

If we now want to check more fine-grained properties, we may rather be interested in the following (tagged) formulation:

$$\mathrm{tr}(\tau')^{\mathtt{F}}_{\mathrm{all}}\ \mathtt{\&\tilde{}}\ \mathrm{tr}(\tau)^{\tilde{}\,\mathrm{old\_complement}}_{\mathtt{T}}$$

In this manner, we can distinguish elements that were added in τ' and whose names did not occur in τ, from elements whose names already occured in τ but whose content model changed in τ', for instance.

In practice, a type is tagged using the predicate $\mathtt{type}(\texttt{"}f\texttt{"}, l, \varphi, \varphi')$ which parses the specified schema, converts it into its logical representation τ and returns the formula $\mathrm{tr}(\tau)^{\varphi'}_{\varphi}$. This kind of type tagging is useful for studying the consequences of schema updates over queries, as presented in the next sections.

4. ANALYSIS PREDICATES

This section introduces the basic analysis tasks offered to XML application designers for assessing the impact of schema evolutions. In particular, we propose a means for identifying the precise reasons for type mismatches or changes in query results under type constraints.

For this purpose, we build on our query and type expression compilers, and define additional predicates that facilitate the formulation of decision problems at a higher level of abstraction. Specifically, these predicates are introduced as logical macros with the goal of allowing system usage while focusing (only) on the XML-side properties, and keeping underlying logical issues transparent for the user. Ultimately, we regard the set of basic logical formulas (such as modalities and recursive binders) as an assembly language, into which predicates are translated.

We illustrate this principle with two simple predicates designed for checking backward-compatibility of schemas, and query satisfiability in the presence of a schema.

—The predicate $\mathtt{backward\_incompatible}(\tau, \tau')$ takes two type expressions as parameters, and assumes τ' is an altered version of τ. This predicate is unsatisfiable iff all instances of τ' are also valid against τ. Any occurrence of this predicate in the input formula will automatically be compiled as $\mathrm{tr}(\tau')^{\mathtt{F}}_{\mathtt{T}}\ \mathtt{\&\tilde{}}\ \mathrm{tr}(\tau)^{\mathtt{F}}_{\mathtt{T}}$.

—The predicate $\mathtt{non\_empty}(\texttt{"}\mathit{query}\texttt{"}, \tau)$ takes an XPath expression (with the syntax defined on Figure 2) and a type expression as parameters, and is unsatisfiable iff the query always returns an empty set of nodes when evaluated on an XML document valid against τ. This predicate compiles into $\mathtt{select}(\texttt{"}\mathit{query}\texttt{"}, \mathrm{tr}(\tau)^{\mathtt{F}}_{\mathtt{T}}\ \mathtt{\&\ \#})$

where the top-level predicate `select`("*query*", φ) compiles the XPath expression *query* into the logic, starting from a context that satisfies φ, as explained in Section 3.2. This can be used to check whether the modification of the schema does not contradict any part of the query.

Notice that the predicate `non_empty`("*query*", τ) can be used for checking whether a query that is valid[5] against a schema remains valid with an updated version of a schema. In other terms, this predicate allows determining whether a query that must always return a non-empty result (whatever the tree on which it is evaluated) keeps verifying the same property with a new version of a schema.

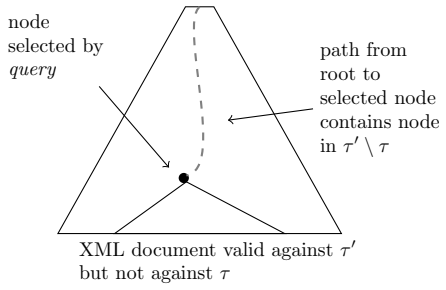
A second, more-elaborate, class of predicates allows formulating problems that combine both a query *query* and two type expressions τ, τ' (where τ' is assumed to be a evolved version of τ):

—`new_element_name`("*query*", τ, τ') is satisfied iff the query *query* selects elements whose names did not occur at all in τ. This is especially useful for queries whose last navigation step contains a "`*`" node test and may thus select unexpected elements. This predicate is compiled into:

$$\tilde{}\texttt{element}(\tau)\ \texttt{\&}\ \texttt{select}(\texttt{"}query\texttt{"}, \mathrm{tr}(\tau')^{\mathtt{F}}_{\mathtt{T}})$$

where `element`(τ) is another predicate that builds the disjunction of all element names occuring in τ. In a similar manner, the predicate `attribute`(φ) builds the logical disjunction of all attribute names used in φ.

—`new_region`("*query*", τ, τ') is satisfied iff the query *query* selects elements whose names already occurred in τ, but such that these nodes now occur in a new context in τ'. In this setting, the path from the root of the document to a node selected by the XPath expression *query* contains a node whose type is defined in τ' but not in τ as illustrated below:



[5]We say that a query is *valid* iff its negation is unsatisfiable.

The predicate $\texttt{new\_region}(\texttt{"}query\texttt{"}, \tau, \tau')$ is logically defined as follows:

$$\begin{aligned}\texttt{new\_region}(\texttt{"}query\texttt{"}, \tau, \tau') \stackrel{\text{def}}{=} \\ \texttt{select}(\texttt{"}query\texttt{"}, \mathrm{tr}(\tau')^{\texttt{F}}_{\texttt{\_all}} \mathbin{\texttt{\&}\tilde{}} \mathrm{tr}(\tau)^{\tilde{}\texttt{\_old\_complement}}_{\texttt{T}}) \\ \mathbin{\texttt{\&}\tilde{}}\, \texttt{added\_element}(\tau, \tau') \\ \texttt{\&}\, \texttt{ancestor}(\texttt{\_old\_complement}) \\ \mathbin{\texttt{\&}\tilde{}}\texttt{descendant}(\texttt{\_old\_complement}) \\ \mathbin{\texttt{\&}\tilde{}}\texttt{following}(\texttt{\_old\_complement}) \\ \mathbin{\texttt{\&}\tilde{}}\texttt{preceding}(\texttt{\_old\_complement})\end{aligned}$$

The previous definition heavily relies on the partition of tree nodes defined by XPath axes, as illustrated by Figure 8. The definition of $\texttt{new\_region}(\texttt{"}query\texttt{"}, \tau, \tau')$ uses an auxiliary predicate $\texttt{added\_element}(\tau, \tau')$ that builds the disjunction of all element names defined in τ' but not in τ (or in other terms, elements that were added in τ'). In a similar manner, the predicate $\texttt{added\_attribute}(\varphi, \varphi')$ builds the disjunction of all attribute names defined in τ' but not in τ. The pred-

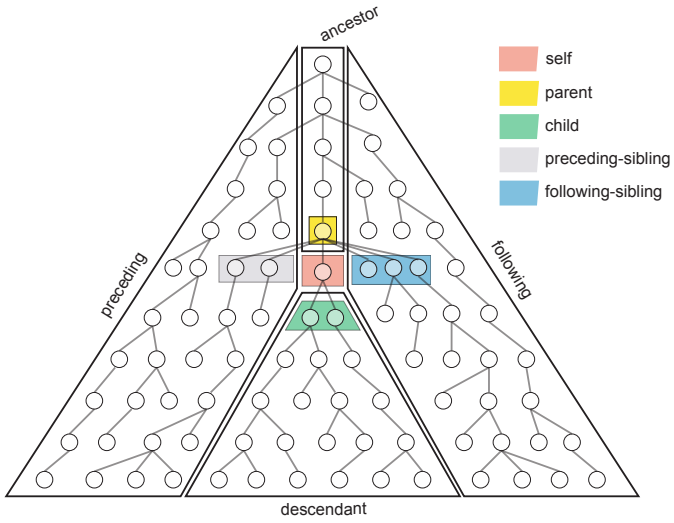
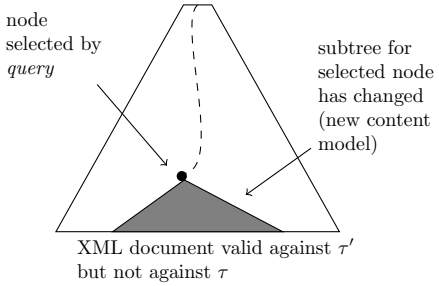


Fig. 8. XPath axes: partition of tree nodes.

icate $\texttt{new\_region}(\texttt{"}query\texttt{"}, \tau, \tau')$ is useful for checking whether a query selects a different set of nodes with τ' than with τ because selected elements may occur in new regions of the document due to changes brought by τ'.

—$\texttt{new\_content}(\texttt{"}query\texttt{"}, \tau, \tau')$ is satisfied iff the query *query* selects elements whose names were already defined in τ, but whose content model has changed due to evolutions brought by τ', as illustrated below:



The definition of $\texttt{new\_content}(\texttt{"}query\texttt{"}, \tau, \tau')$ follows:

$$
\begin{aligned}
&\texttt{new\_content}(\texttt{"}query\texttt{"}, \tau, \tau') \stackrel{\text{def}}{=} \\
&\qquad \texttt{select}(\texttt{"}query\texttt{"}, \mathrm{tr}(\tau')^{\texttt{F}}_{\texttt{\_all}} \mathbin{\&\tilde{}} \widetilde{\mathrm{tr}(\tau)}^{\texttt{\_old\_complement}}_{\texttt{T}}) \\
&\qquad\qquad \mathbin{\&\tilde{}}\, \texttt{added\_element}(\tau, \tau') \\
&\qquad\qquad \mathbin{\&\tilde{}}\, \texttt{ancestor}(\texttt{added\_element}(\tau, \tau')) \\
&\qquad\qquad \mathbin{\&}\, \texttt{descendant}(\texttt{\_old\_complement}) \\
&\qquad\qquad \mathbin{\&\tilde{}}\, \texttt{following}(\texttt{\_old\_complement}) \\
&\qquad\qquad \mathbin{\&\tilde{}}\, \texttt{preceding}(\texttt{\_old\_complement})
\end{aligned}
$$

The predicate $\texttt{new\_content}(\texttt{"}query\texttt{"}, \tau, \tau')$ can be used for ensuring that XPath expressions will not return nodes with a possibly new content model that may cause problems. For instance, this allows checking whether an XPath expression whose resulting node set is converted to a string value (as in, *e.g.* XPath expressions used in XSLT "value-of" instructions) is affected by the changes from τ to τ'.

—$\texttt{new\_sibling}(\texttt{"}query\texttt{"}, \tau, \tau')$ is satisfied iff the query *query* selects elements whose names already occurred in τ, but such that they now occur with new potential siblings due to τ'. The notion of context, here, is extended to be not only the chain of ancestors from the selected node to the root but also the set of previous and following siblings of the selected node.

The previously defined predicates can be used to help the programmer identify precisely how type constraint evolutions affect queries. They can even be combined with usual logical connectives to formulate even more sophisticated problems. For example, let us define the predicate $\texttt{exclude}(\varphi)$ which is satisfiable iff there is no node that satisfies φ in the whole tree. This predicate can be used for excluding specific element names or even nodes selected by a given XPath expression. It is defined as follows:

$$\texttt{exclude}(\varphi) \stackrel{\text{def}}{=} \mathbin{\tilde{}}\, \texttt{ancestor-or-self}(\texttt{descendant-or-self}(\varphi))$$

This predicate can also be used for checking properties in an iterative manner, refining the property to be tested at each step. It can also be used for verifying fine-grained properties. For instance, one may check whether τ' defines the same set of trees as τ modulo new element names that were added in τ' with the following

formulation:

$$\tilde{}(\tau \texttt{ <=> } \tau') \texttt{ \& exclude(added\_element}(\tau, \tau'))$$

This allows identifying that, during the type evolution from τ to τ', the query results change has not been caused by the type extension but by new compositions of nodes from the older type.

In practice, instead of taking internal tree type representations (as defined in Section 2.2) as parameters, most predicates do actually take any logical formula as parameter, or even schema paths as parameters. We believe this facilitates predicates usage and, most notably, how they can be composed together. Figure 9 gives the syntax of built-in predicates as they are implemented in the system, where f is a file path to a DTD (.dtd), XML Schema (.xsd), or Relax NG (.rng). In addition of aforementioned predicates, the predicate `descendant`(φ) forces the

predicate ::=

- `select("`*query*`")`
- `select("`*query*`",` φ`)`
- `exists("`*query*`")`
- `exists("`*query*`",` φ`)`
- `type("`f`",` l`)`
- `type("`f`",` l, φ, φ'`)`
- `forward_incompatible(`φ, φ'`)`
- `backward_incompatible(`φ, φ'`)`
- `element(`φ`)`
- `attribute(`φ`)`
- `descendant(`φ`)`
- `exclude(`φ`)`
- `added_element(`φ, φ'`)`
- `added_attribute(`φ, φ'`)`
- `non_empty("`*query*`",` φ`)`
- `new_element_name("`*query*`", "`f`", "`f'`",` l`)`
- `new_region("`*query*`", "`f`", "`f'`",` l`)`
- `new_sibling("`*query*`", "`f`", "`f'`",` l`)`
- `new_content("`*query*`", "`f`", "`f'`",` l`)`
- *predicate-name*$(\langle\varphi\rangle^{\oplus})$

Fig. 9. Syntax of Predicates for XML Reasoning.

existence of a node satisfying φ in the subtree, and *predicate-name*$(\langle\varphi\rangle^{\oplus})$ is a call to a custom predicate, as explained in the next section.

4.1 Custom Predicates

Following the spirit of predicates presented in the previous section, users may also define their own custom predicates. The full syntax of XML logical specifications to be used with the system is defined on Figure 10, where the meta-syntax $\langle X\rangle^{\oplus}$ means one or more occurrence of X separated by commas. A global problem specification can be any formula (as defined on Figure 6), or a list of custom predicate

definitions separated by semicolons and followed by a formula. A custom predicate may have parameters that are instanciated with actual formulas when the custom predicate is called (as shown on Figure 9). A formula bound to a custom predicate may include calls to other predicates, but not to the currently defined predicate (recursive definitions must be made through the let binder shown on Figure 6).

$spec$ ::=		
	φ	formula (see Fig. 6)
	$def; \varphi$	
def ::=		
	$predicate\text{-}name(\langle l \rangle^{\oplus}) = \varphi'$	custom definition
	$def; def$	list of definitions

Fig. 10. Global Syntax for Specifying Problems.

5. IMPACT OF STANDARD SCHEMA EVOLUTION ON VALID DOCUMENTS

As depicted on Fig. 1, the whole system relies on a satisfiability solver for the underlying logic. The main principe of the satisfiability-solver is an exhaustive search for a tree that satisfies the formula. The search relies on a least fixpoint computation that starts from all possible leaves and attempt to plug every possible parent node at each further step. The algorithm terminates once the initial formula has been found to hold in a given node of the tree. Otherwise, the algorithm terminates when no more parent nodes can be added. The algorithm, as well as proofs of its soundness and completeness, optimal complexity, and implementation techniques are detailed in [Genevès et al. 2007].

We have carried out extensive experiments of the system in real world settings, e.g. with popular web schemas such as XHTML, MathML, SVG, SMIL (Table II gives details related to their respective sizes). In this section, we show how the tool can be used to analyze different situations where schemas changes have important consequences on the validity of existing documents.

Schema	Variables	Elements	Attributes
XHTML 1.0 basic DTD	71	52	57
XHTML 1.1 basic DTD	89	67	83
MathML 1.01 DTD	137	127	72
MathML 2.0 DTD	194	181	97

Table II. Sizes of (Some) Considered Schemas.

One major role of organizations such as W3C is to contribute to the standardization effort leading to a unique widely accepted set of constraints for a given class of documents. Designing a normative specification is a complex process, which is made even harder by a few important considerations. For example, when a language is designed, one need to take into account how future versions of that language can evolve. For a particular version of a language, not only the schema constraints allowed by that version need to be considered but also how they can be modified

in future versions. This allows to address how an implementation of this version should process document variants added by future schema versions.

Specifically, we identify three different properties for a specification:

—*Forward compatibility*: All instances of an older specification should be valid with respect to newer specifications. This ensures that a document can still be processed properly with applications implementing newer specifications.

—*Backward compatibility without added elements/attributes*: New combinations of old elements are not supposed to be introduced in later specifications. Otherwise, an application implementing an older specification will not able to process a document that conforms to some future specification, even if this document does not contain any element or attribute introduced as extensions.

—*Equivalence between schema versions*: A given specification can be expressed in a variety of schema definition languages like DTD, XML Schema, Relax NG. We expect the different schema versions of the same specification to define the same set of documents modulo the expressivity of the schema language [Murata et al. 2005].

An XML schema definition (whether normative or not) often evolves over time, as new needs often result in new features usually introduced as new elements and attributes. However we believe that this normal evolution should not break the three previous properties.

We report below on using the framework for characterizing the evolution of the main standard document formats used on the web, including W3C XHTML, SMIL, SVG and MathML, based on the criteria identified above. This kind of analyses yield important observations on the validity of, potentially, billions of documents.

XHTML Basic

The first test consists in analyzing the relationship (forward and backward compatibility) between XHTML basic 1.0 and XHTML basic 1.1 schemas. In particular, backward compatibility can be checked by the following command:

```
backward_incompatible("xhtml-basic10.dtd",
                      "xhtml-basic11.dtd", "html")
```

Executing the test yields a counter example as the new schema contains new element names. The counter example (shown below) contains a `style` element occurring as a child of `head`, which is not permitted in XHTML basic 1.0:

```
<html>
  <head>
    <title/>
    <style type="_otherV"/>
  </head>
  <body/>
</html>
```

The next step consists in focusing on the relationship between both schemas excluding these new elements. This can be formulated by the following command:

```
backward_incompatible("xhtml-basic10.dtd",
                      "xhtml-basic11.dtd", "html")
& exclude(added_element(
      type("xhtml-basic10.dtd","html"),
      type("xhtml-basic11.dtd", "html")))
```

The result of the test shows a counter example document that proves that XHTML basic 1.1 is not backward compatible with XHTML basic 1.0 even if new elements are not considered. In particular, the content model of the `label` element cannot have an `a` element in XHTML basic 1.0 while it can in XHTML basic 1.1. The counter example produced by the solver is shown below:

```
<html>
  <head>
    <object>
      <label>
        <a href="...">
          <img/>
        </a>
        <img/>
      </label>
      <param/>
    </object>
    <meta/>
    <title/>
    <base/>
  </head>
  <body/>
</html>

XTML basic 1.0 validity error: element a is not
declared in label list of possible children
```

SMIL

The second test consists in analyzing the relationship (forward and backward compatibility) between several versions of the SMIL standard[6], namely versions 1.0, 2.0, and 3.0. In particular, forward compatibility between 1.0 and 2.0 can be checked by the following command:

```
forward_incompatible("SMIL10.dtd", "SMIL20.dtd", "smil")
```

The result of the test shows a counter example document that proves that there exist valid SMIL 1.0 documents that are not valid anymore with respect to SMIL 2.0. In fact that is because the content model of the `layout` element is defined as `any` in SMIL 1.0, whereas it is more restricted in SMIL 2.0. We observe that

[6]The first author was a member of the W3C SMIL working group and a co-author of SMIL 2.0 and 2.1.

introducing `any` is a choice that has important consequences. Indeed, a document that was playable with 1.0 implementations may no longer be playable using 2.0 implementations. The counter example produced by the solver is shown below:

```
<smil>
  <head>
    <layout>
      <meta content="_otherV" name="_otherV"/>
    </layout>
  </head>
</smil>

SMIL 2.0 validity error:
Element layout content does not follow the DTD,
expecting (region|topLayout|root-layout|regPoint)*,
got (meta)
```

The lesson here is that introducing very permissive content models (like `any`) has to be considered very seriously. Indeed, that means that all future versions of the standard should be at least as permissive. Otherwise, all content produced with earlier (more permissive) versions becomes at risk. Therefore, the initial content model has to be carefully designed in order to avoid such situations.

The following example is even worse. We check forward compatibility between SMIL 2.0 and 3.0:

```
forward_incompatible("SMIL20.dtd",
                          "SMIL30Language.dtd", "smil")
```

We obtain the following counter-example:

```
<smil xmlns="http://www.w3.org/2001/SMIL20/Language">
  <body>
    <switch>
      <animateMotion/>
    </switch>
    <a href="..."/>
  </body>
</smil>
```

This document is valid with respect to SMIL 2.0. However it does not validate with respect to SMIL 3.0. That is because the content model for the `switch` element was set to a more restrictive pattern in version 3.0 compared to 2.0, as the following validation error message suggests:

```
SMIL 3.0 validity error :
Element switch content does not follow the DTD,
expecting ((metadata | switch)* , ((((animate | set |
animateMotion | animateColor) , (metadata | switch)*)* ,
(((par | seq | excl | audio | video | animation | text |
 ... switch)*)+)) | (layout , (metadata | switch)*)*)),
got (animateMotion)
```

Now we would like to know if the bug is limited to the occurrence of the `animateMotion` element or whether it is more general. To this end, we progressively exclude elements named `animateMotion`, `set`, `animateColor`, and `animate`, as follows:

```
forward_incompatible("SMIL20.dtd",
                          "SMIL30Language.dtd", "smil")
  & exclude(animateMotion) & exclude(set)
  & exclude(animateColor) & exclude(animate)
```

We still obtain the following counter-example (valid w.r.t SMIL 2.0 but not w.r.t SMIL 3.0), which shows that the forward incompatibility is not limited to the occurence of the previous elements, but rather, to severe limitations of the `switch` content model introduced in 3.0. In other words, `switch` is an element which undermines SMIL forward compatibility.

```
<smil xmlns="http://www.w3.org/2001/SMIL20/Language">
  <body>
    <switch>
      <seq/>
      <area/>
    </switch>
    <switch/>
    <a href="..."/>
  </body>
</smil>
```

SVG

The SVG test consists in analyzing the relationship (forward and backward compatibility) between SVG 1.0 et 1.1. In particular, we examine the different profiles (tiny, basic and full) from 1.0 and compare them to 1.1 schemas. Backward compatibility can be checked by the following command:

```
forward_incompatible("svg10.dtd",
              "svg11-flat-20030114.dtd", "svg")
```

The test is unsatisfiable meaning that SVG 1.1 is formally proven to be forward compatible with SVG 1.0. This is good news as it means that all 1.0 documents will be supported with 1.1 conforming implementations, without any exception. In the case where a 1.0 document does not play with a 1.1 implementation, this indicates a bug in the implementation and not in the SVG specification.

We observe here that the common practice of including a single doctype declaration within a document is questionable, since a document is not only valid w.r.t a given schema but also w.r.t to all future forward-compatible versions. Keeping track of this mapping between a document and several schemas allows the document to be supported by a larger set of implementations.

Similar tests on the SVG 1.1 tiny, basic and full also exhibit good results. This corresponds to the definition of these three profiles as strict subsets of each other. Furthermore, we believe that the use of a modularized version of a schema (as opposed to a complete redefinition) has helped in avoiding compatibility problems.

We now focus on testing the backward compatibility between the SVG basic 1.1 profile and SVG 1.0 profile. The test fails even if new features are left aside:

```
backward_incompatible("svg10.dtd",
            "svg11-basic.dtd", "svg")
& exclude( added_element(type("svg10.dtd", "svg"),
                         type("svg11-basic.dtd","svg")))
& exclude(switch)
```

This test yields the following counter-example which confirms that there is actually a flaw in the 1.1 specification:

```
<svg>
  <image href="..." width="..." height="...">
    <title/>
    <title/>
  </image>
</svg>
```

as it allows two `title` elements to occur inside an `image` element, which was not allowed in the 1.0.

MathML

We apply a similar investigation approach to MathML 1.0 and its newer version 2.0. We formulate a backward compatibility test without elements that were added in version 2.0. Furthermore, we want to exclude immediate trivial counter-examples involving the use of the `declare` element as well as of the `math` element occuring within the `annotation-xml` element. For this purpose, we use the following formulation:

```
backward_incompatible("mathml.dtd","mathml2.dtd","math")
& exclude( added_element( type("mathml.dtd","math"),
                          type("mathml2.dtd","math")))
& exclude(declare)
& (~descendant(math))
```

that bans the `declare` element from occuring in the whole tree (achieved with the use of the `exclude(declare)` predicate), and prevents the `math` element from ocurring in the root's subtree (owing to the use of the `(~descendant(math))` predicate) The following counter-example is produced:

```
<math>
  <apply>
    <annotation-xml>
      <mprescripts/>
    </annotation-xml>
  </apply>
</math>
```

Such backward incompatibilities suggest that applications cannot simply ignore new elements from newer schemas, as the combination of older elements may evolve significantly from one version to another.

6. IMPACT OF SCHEMA EVOLUTION ON QUERIES

In this section, we report on using the framework in order to evaluate the consequences of schema changes on XPath queries such as the ones found in transformations like the MathML content to presentation conversion [Pietriga 2005].

MathML Content to Presentation Conversion

MathML is an XML format for describing mathematical notations and capturing both its mathematical structure and graphical rendering, also known as Content MathML and Presentation MathML respectively. The structure of a given equation is kept separate from the presentation and the rendering part can be generated from the structure description. This operation is usually carried out using an XSLT transformation that achieves the conversion. In this test series, we focus on the analysis of the queries contained in such a transformation sheet and evaluate the impact of the schema change from MathML 1.0 to MathML 2.0 on these queries.

Most of the queries contained in the transformation represent only a few patterns very similar up to element names. The following three patterns are the most frequently used:

```
Q1:  //apply[*[1][self::eq]]
Q2:  //apply[*[1][self::apply]/inverse]
Q3:  //sin[preceding-sibling::*[position()=last()
              and (self::compose or self::inverse)]]
```

The first test is formulated by the following command:

```
new_region("Q1","mathml.dtd","mathml2.dtd","math")
```

The result of the test shows a counter example document that proves that the query may select nodes in new contexts in MathML 2.0 compared to MathML 1.0. In particular, the query `Q1` selects `apply` elements whose ancestors can be `declare` elements, as indicated on the document produced by the solver[7]:

```
<math xmlns:solver="http://wam.inrialpes.fr/xml"
            solver:context="true">
  <declare>
    <apply solver:target="true">
      <eq/>
    </apply>
    <condition/>
  </declare>
</math>
```

[7]Notice that the solver automatically annotates a pair of nodes related by the query: when the query is evaluated from a node marked with the attribute `solver:context`, the node marked with `solver:target` is selected.

To evaluate the effect of this change, the counter example is filled with content and passed as an input parameter to the transformation. This shows immediately a bug in the transformation as the resulting document is not a MathML 2.0 presentation document. Based on this analysis, we know that the XSLT template associated with the match pattern `Q1` must be updated to cope with MathML evolution from version 1.0 to version 2.0.

The next test consists in evaluating the impact of the MathML type evolution for the query `Q2` while excluding all new elements added in MathML 2.0 from the test. This identifies whether old elements of MathML 1.0 can be composed in MathML 2.0 in a different manner. This can be performed with the following command:

```
new_content("Q2","mathml.dtd","mathml2.dtd","math")
& exclude(added_element(type("mathml.dtd","math"),
                        type("mathml2.dtd", "math")))
```

The test result shows an example document that effectively combines MathML 1.0 elements in a way that was not allowed in MathML 1.0 but permitted in MathML 2.0.

```
<math xmlns:solver="http://wam.inrialpes.fr/xml"
            solver:context="true">
  <apply solver:target="true">
    <apply>
      <inverse/>
    </apply>
    <annotation-xml>
      <math/>
    </annotation-xml>
    <condition/>
  </apply>
</math>
```

Similarly, the last test consists in evaluating the impact of the MathML type evolution for the query `Q3`, excluding all new elements added in MathML 2.0 and counter example documents containing `declare` elements (to avoid trivial counter examples):

```
new_region("Q3","mathml.dtd","mathml2.dtd","math")
& exclude(added_element(type("mathml.dtd","math"),
                        type("mathml2.dtd","math")))
                        & exclude(declare)
```

The counter example document shown below illustrates a case where the `sin` element occurs in a new context.

```
<math xmlns:solver="http://wam.inrialpes.fr/xml"
            solver:context="true">
  <apply>
    <annotation-xml>
      <math>
```

```
        <apply>
          <inverse/>
          <sin solver:target="true"/>
        </apply>
      </math>
    </annotation-xml>
  </apply>
</math>
```

Applying the transformation on previous examples yields documents which are neither MathML 1.0 nor MathML 2.0 valid. As a result, the stylesheet cannot be used safely over documents of the new type without modifications. In addition, the required changes to the stylesheet are not limited to the addition of new templates for MathML 2.0 elements. The templates that deal with the composition of MathML 1.0 elements should be revised as well.

7. SYSTEM IMPLEMENTATION

We have implemented the whole software architecture described in Section 2 and illustrated on Figure 1. The tool [Genevès and Layaïda 2009] is available online from:

`http://wam.inrialpes.fr/xml`

Interaction with the system is offered through a web user interface in a web browser. Figure 11 presents a screenshot of the user interface. The user can either enter an analysis problem using predicates through area (1) of Figure 11 or select from pre-loaded analysis tasks offered in area (4) of Figure 11. The level of details displayed by the analyzer can be adjusted in area (2) of Figure 11 and allows to inspect logical translations and statistics on problem size and the different operation costs. The results of the analysis are displayed in area (3) of Figure 11 together with XML counter-examples.

All the previous tests were processed in less than 30 seconds on an ordinary laptop computer running Mac OS X. The 30s correspond to the most complex use cases. Most complex means analyzing recursive forward/backward and qualified queries such as `Q3`, under evolution of large and heavily recursive schemas such as XHTML and MathML (large number of type variables, elements and attributes: see Table II). These are the hardest cases measured in practice with the implementation. Most of other schemas and queries usually found in applications are much simpler than the ones presented in this paper and will obviously be solved much faster. Given the variety of schemas occurring in practice, we focused on the most complex W3C standard schemas. The full online implementation [Genevès and Layaïda 2009] allows to run all the tests described in the paper as well as user-supplied ones. It shows intermediate compilation stages, generated formulae (in particular the translation of schemas into the logic), and reports on the performance of each step of the analysis.

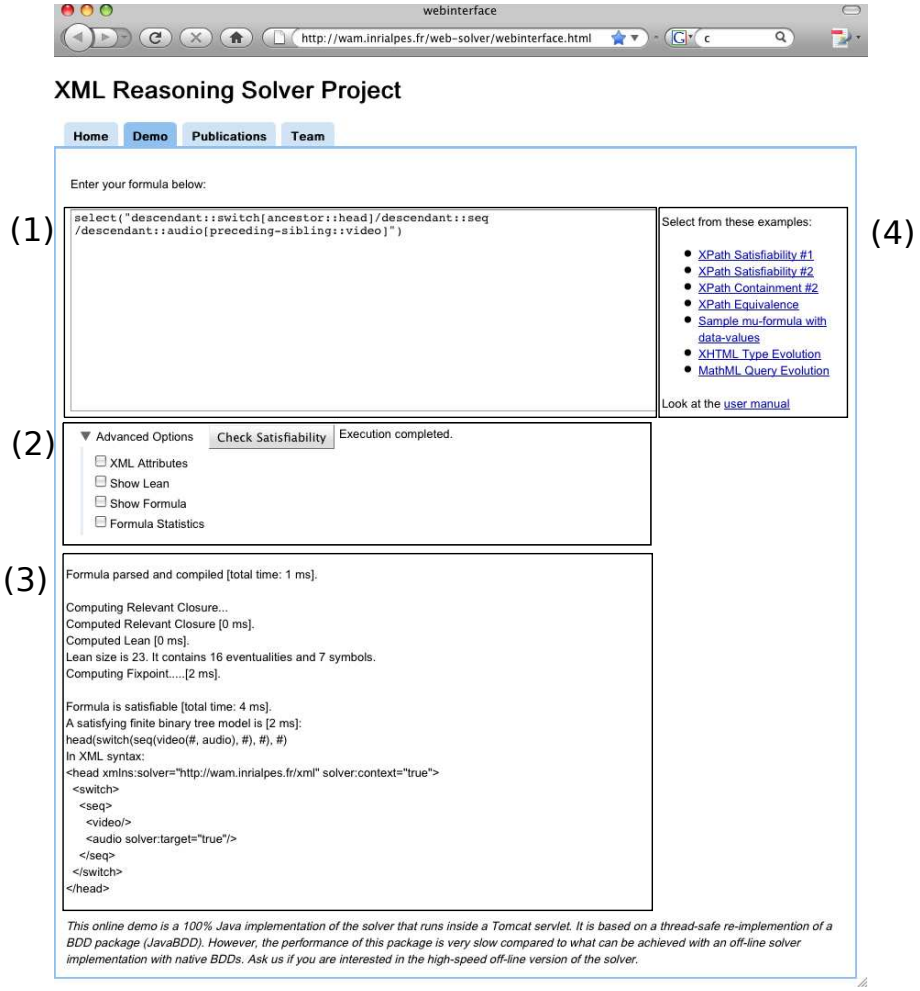


Fig. 11. Screenshot of the Web-Based Solver Interface.

8. RELATED WORK

Schema evolution is an important topic and has been extensively explored in the context of relational, object-oriented, and XML databases. Most of the previous work for XML query reformulation is approached through reductions to relational problems [Beyer et al. 2005]. This is because schema evolution was considered as a storage problem where the priority consists in ensuring data consistency across multiple relational schema versions. In such settings, two distinct schemas and an explicit description of the mapping between them are assumed as input. The problem then consists in reformulating a query expressed in terms of one schema into a semantically equivalent query in terms of the other schema: see [Yu and Popa 2005] and more recently [Moon et al. 2008] with references thereof.

In addition to the fundamental differences between XML and the relational data

model, in the more general case of XML processing, schemas constantly evolve in a distributed, independent, and unpredictable environment. The relations between different schemas are not only unknown but hard to track. In this context, one priority is to help maintaining query consistency during these evolutions, which is still considered as a challenging problem [Sedlar 2005; Rose 2004]. The absence of evolution analysis tools for XML/XPath contrasts with the abundance of tools and methods routinely used in relational databases.

The work found in [Moro et al. 2007] discusses the impact of evolving XML schemas on query reformulation. Based on a taxonomy of XML schema changes during their evolution, the authors provide informal – not exact nor systematic – guidelines for writing queries which are less sensitive to schema evolution. In fact, studying query reformulation requires at least the ability to analyze the relationship between queries. For this reason, a closely related work is the problem of determining query containment and satisfiability under type constraints [Benedikt et al. 2005; Colazzo et al. 2006; Genevès et al. 2007]. These static analysis tasks are also notably useful for performing query optimization [Groppe et al. 2006].

The works found in [Benedikt et al. 2005; Groppe and Groppe 2008] study the complexity of XPath emptiness and containment for various fragments with or without type constraints (see [Benedikt and Koch 2009] and references thereof for a survey). In [Colazzo et al. 2004; 2006], a technique is presented for statically ensuring correctness of paths. The approach deals with emptiness of XPath expressions without reverse axes. The work presented in [Genevès et al. 2007] solves the more general problem of containment, including reverse axes.

The main distinctive idea pursued in this paper is to develop a logical approach for guiding schema and query evolution. In contrast to the previous use of logics for proving properties such as query emptiness or equivalence, the goal here is different in that we seek to provide the necessary tools to produce relevant knowledge when such relations do not hold. From a complexity point-of-view, it is worth noticing that the addition of predicates does not increase complexity for the underlying logic shown in [Genevès et al. 2007].

We would also like to emphasize that, to the best of our knowledge, this work is the first to provide precise analyses of XML evolution, that was tested on real life use cases (such as XHTML and MathML types) and complex queries (involving recursive and backward navigation). As a consequence, in this context, analysis tools such as type-checkers [Hosoya and Pierce 2003; Benzaken et al. 2003; Møller and Schwartzbach 2005; Gapeyev et al. 2006; Castagna and Nguyen 2008] do no match the expressiveness, typing precision, and analysis capabilities of the work presented here.

9. CONCLUSION

In this article, we present an application of a unifying logical framework for verifying forward/backward compatibility issues caused by schemas evolution. We provide evidence that such a framework can be successfully used to overcome the obstacles of the analysis of XML schema evolution. This kind of analyses is widely considered as a challenging problem in XML programming. As mentioned earlier, the difficulty is twofold: first it requires dealing with large and complex language constructions such

as XML types and XPath queries, and second, it requires modeling and reasoning about evolution of such constructions.

We presented the logical foundations of the framework. We then applied the framework for analyzing two major issues due to schema evolution: first, the consequence on the validity of documents and, second, the impact on queries. The presented system detected several compatibility problems in the main document formats used on the web. The same tool also allows XML designers to identify queries that need reformulation in order to produce the expected results across successive schema versions. With this tool designers can examine precisely the impact of schema changes over queries, therefore facilitating their reformulation.

We gave illustrations of how to use the tool for schema evolution on realistic examples. In particular, we considered typical situations in applications involving evolution of W3C schemas used on the web such as XHTML and MathML. We believe that the tool can be very useful for standard schema writers and maintainers in order to assist them enforce some level of quality assurance on compatibility between versions.

One direction for future work is to search for techniques giving suggestions on how to rewrite the query into an equivalent one to accommodate schema changes.

REFERENCES

Benedikt, M., Fan, W., and Geerts, F. 2005. XPath satisfiability in the presence of DTDs. In *PODS '05*. ACM Press, 25–36.

Benedikt, M. and Koch, C. 2009. XPath leashed. *ACM Comput. Surv. 41*, 3:1–3:54.

Benzaken, V., Castagna, G., and Frisch, A. 2003. CDuce: An XML-centric general-purpose language. In *ICFP '03: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*. ACM Press, New York, NY, USA, 51–63.

Beyer, K., Özcan, F., Saiprasad, S., and der Linden, B. V. 2005. DB2/XML: designing for evolution. In *SIGMOD '05*. ACM, 948–952.

Castagna, G. and Nguyen, K. 2008. Typed iterators for XML. In *ICFP*. 15–26.

Clark, J. and DeRose, S. 1999. XML path language (XPath) version 1.0, W3C recommendation. http://www.w3.org/TR/ 1999/REC-xpath-19991116.

Colazzo, D., Ghelli, G., Manghi, P., and Sartiani, C. 2004. Types for path correctness of XML queries. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*. ACM Press, New York, NY, USA, 126–137.

Colazzo, D., Ghelli, G., Manghi, P., and Sartiani, C. 2006. Static analysis for path correctness of XML queries. *J. Funct. Program. 16,* 4-5, 621–661.

Gapeyev, V., Garillot, F., and Pierce, B. C. 2006. Statically typed document transformation: An Xtatic experience. In *PLAN-X 2006: Proceedings of the International Workshop on Programming Language Technologies for XML*. BRICS Notes Series, vol. NS-05-6. BRICS, Aarhus, Denmark, 2–13.

Genevès, P. 2006. Logics for XML. Ph.D. thesis, Institut National Polytechnique de Grenoble. http://www.pierresoft.com/pierre.geneves/phd.htm.

Genevès, P. and Layaïda, N. 2009. The XML reasoning solver project. http://wam.inrialpes.fr/xml.

Genevès, P., Layaïda, N., and Quint, V. 2009. Identifying query incompatibilities with evolving XML schemas. In *ICFP '09: Proceedings of the ACM SIGPLAN international conference on Functional programming*. 221–230.

Genevès, P., Layaïda, N., and Schmitt, A. 2007. Efficient static analysis of XML paths and types. In *PLDI '07*. ACM Press, 342–351.

Genevès, P., Layaïda, N., and Schmitt, A. 2008. Efficient static analysis of XML paths and types. Long version of [Genevès et al. 2007], Research Report 6590, INRIA. July.

Groppe, J. and Groppe, S. 2008. Filtering unsatisfiable XPath queries. *Data Knowl. Eng. 64,* 1, 134–169.

Groppe, S., Bottcher, S., and Groppe, J. 2006. XPath query simplification with regard to the elimination of intersect and except operators. In *ICDEW '06: Proceedings of the 22nd International Conference on Data Engineering Workshops*. IEEE Computer Society, Washington, DC, USA, 86.

Hosoya, H. and Pierce, B. C. 2003. XDuce: A statically typed XML processing language. *ACM Trans. Inter. Tech. 3,* 2, 117–148.

Hosoya, H., Vouillon, J., and Pierce, B. C. 2005. Regular expression types for XML. *ACM TOPLAS 27,* 1, 46–90.

Møller, A. and Schwartzbach, M. I. 2005. The design space of type checkers for XML transformation languages. In *Proc. Tenth International Conference on Database Theory, ICDT '05*. LNCS, vol. 3363. Springer-Verlag, London, UK, 17–36.

Moon, H. J., Curino, C. A., Deutsch, A., and Hou, C.-Y. 2008. Managing and querying transaction-time databases under schema evolution. In *VLDB '08*. VLDB Endowment, 882–895.

Moro, M. M., Malaika, S., and Lim, L. 2007. Preserving xml queries during schema evolution. In *WWW '07*. ACM, 1341–1342.

Murata, M., Lee, D., Mani, M., and Kawaguchi, K. 2005. Taxonomy of XML schema languages using formal language theory. *ACM TOIT 5,* 4, 660–704.

Pietriga, E. 2005. MathML content2presentation transformation. http://www.lri.fr/~pietriga/mathmlc2p/mathmlc2p.html.

Rose, K. H. 2004. The XML world view. In *DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering*. ACM, New York, NY, USA, 34–34.

Sedlar, E. 2005. Managing structure in bits & pieces: the killer use case for XML. In *SIGMOD '05*. ACM, 818–821.

Thomas, W. 1990. Automata on infinite objects. In *Handbook of theoretical computer science (vol. B): formal models and semantics*. MIT Press, Cambridge, MA, USA, 133–191.

Wadler, P. 2000. Two semantics for XPath. Internal Technical Note of the W3C XSL Working Group, http://homepages.inf.ed.ac.uk/wadler/papers/xpath-semantics/xpath-semantics.pdf.

Yu, C. and Popa, L. 2005. Semantic adaptation of schema mappings when schemas evolve. In *VLDB '05*. VLDB Endowment, 1006–1017.