HAL
open science

Ensuring Query Compatibility with Evolving XML Schemas

Pierre Genevès, Nabil Layaïda, Vincent Quint

To cite this version:

Pierre Genevès, Nabil Layaïda, Vincent Quint. Ensuring Query Compatibility with Evolving XML Schemas. [Research Report] RR-6711, INRIA. 2008. inria-00336360

HAL Id: inria-00336360

https://inria.hal.science/inria-00336360v1

Submitted on 12 Nov 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Ensuring Query Compatibility with Evolving XML Schemas

Pierre Genevès — Nabil Layaïda — Vincent Quint

N° 6711

Novembre 2008

Thème SYM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6711--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

INRIA

centre de recherche GRENOBLE - RHÔNE-ALPES

Ensuring Query Compatibility with Evolving XML Schemas

Pierre Genevès*, Nabil Layaïda , Vincent Quint

Thème SYM — Systèmes symboliques
Équipes-Projets Wam

Rapport de recherche n° 6711 — Novembre 2008 — 24 pages

Abstract: During the life cycle of an XML application, both schemas and queries may change from one version to another. Schema evolutions may affect query results and potentially the validity of produced data. Nowadays, a challenge is to assess and accommodate the impact of theses changes in rapidly evolving XML applications.

This article proposes a logical framework and tool for verifying forward/backward compatibility issues involving schemas and queries. First, it allows analyzing relations between schemas. Second, it allows XML designers to identify queries that must be reformulated in order to produce the expected results across successive schema versions. Third, it allows examining more precisely the impact of schema changes over queries, therefore facilitating their reformulation.

Key-words: XML, Schema, Queries, XPath, Evolution, Compatibility, Analysis

* CNRS

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Ensuring Query Compatibility with Evolving XML Schemas

Résumé : Durant le cycle de vie d'une application XML, à la fois les schémas et les requêtes sont amenés à évoluer d'une version à une autre. Les évolutions de schémas peuvent affecter les résultats des requêtes et potentiellement la validité des données produites. De nos jours, un vrai défi consiste à évaluer et à prendre en compte l'impact de ces changements dans des applications XML qui évoluent rapidement.

Cet article propose un cadre logique et un outil pour la vérification des compatibilités ascendante et descendante des schémas et des requêtes. Tout d'abord, il permet d'analyser les relations entre les schémas. Ensuite, il permet au concepteur XML d'identifier les requêtes qui doivent être reformulées afin de produire les résultats attendus à travers les versions successives des schémas. Enfin, il permet d'examiner de manière plus précise l'impact des changements des schémas sur les requêtes, facitilitant de ce fait leur formulation.

Mots-clés : XML, Schema, Requêtes, XPath, Evolution, Compatibilité, Analyse

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 is expected to have also 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 [14]. 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.

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 [4]. 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 [6, 18] and more recently [12] 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 [16].

The work found in [13] 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 [1, 9]. The work found in [1] studies the complexity of XPath emptiness and containment for various fragments (see [2] and references thereof for a survey).

The main distinctive idea pursued in this paper is to develop a logical approach for guiding schema and query evolution. In contrast to the classical use of logics for proving properties such as query emptiness or equivalence [1, 9], the goal here is different in that we seek to provide the necessary tools to produce relevant knowledge when such relations do not hold.

Outline

The rest of this paper is organized as follows: the next section introduces our framework, Section 3 presents its underlying logic, and Section 4 presents predicates for characterizing the impact of schema changes. We report on experiments on realistic scenarios in Section 5 before we conclude in Section 6.

2 Analysis Framework

Our framework allows the automatic verification of properties related to XML schema and query evolution. In particular, it offers the possibility of checking fine-grained properties on the behavior of queries with respect to successive versions of a given schema. The system can be used for checking 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.

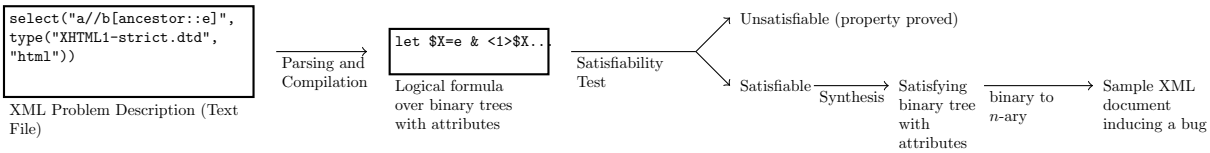


Figure 1: Framework Overview.

.

For these purposes, our framework relies on the combination and joint use of several contributions:

- an extension of the logic introduced in [9] to deal with XML attributes (Sections 2 and 3);
- a set of logical features and high-level predicates specifically designed for studying and characterizing schema and query compatibility issues when schemas evolve (Section 4);
- a range of applications and procedures to cope with schema and query evolution (Section 5);
- a full implementation of the whole system, including:
  - a parser for reading the problem description (text file), which in turn use specific parsers for schemas (Section 2.2), queries (Section 2.3), logical formulas (Section 3.2), and predicates (Section 4);
  - compilers for translating schemas and queries into their logical representations (Sections 3.3 and 3.4);
  - an optimized solver first described in [9, 10] for checking satisfiability of logical formulas in time $2^{O(n)}$ where n is the formula size;
  - and a counter example XML tree generator (described in [10]).

Figure 1 illustrates how the previous software components are combined and used together, in a simplified overview of the global framework. We next 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.

2.2 Type Constraints

As an internal representation for tree grammars, we consider regular tree type expressions (in the manner of [11]), 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
		$\texttt{()}$	empty sequence
		$\tau \mid \tau$	disjunction
		τ, τ	concatenation
		$l\texttt{(}a\texttt{)[}\tau\texttt{]}$	element definition
		x	variable
		$\texttt{let}\ \overline{x.\tau}\ \texttt{in}\ \tau$	binder

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

Our tree type expressions capture most of the schemas in use today [14, 3]. In practice, our system provides parsers that convert DTDs, XML Schemas, and Relax NGs to this internal tree type representation. 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.

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 [5], and more formally in [17]. 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: $\underbrace{axis{::}nt/.../axis{::}nt}_{k \text{ steps}}$.

3 Logical Setting

3.1 Logical Data Model

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

[1] For instance, "$\texttt{let}\ x.l\texttt{(}a\texttt{)[}\tau\texttt{]}, x \mid \texttt{()}\ \texttt{in}\ x$" is allowed.

query	::=		
		/*path*	absolute path
		path	relative path
		query | *query*	union
		query ∩ *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	

Figure 2: XPath Expressions.

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

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

Figure 3: Syntactic Sugars and their Rewritings.

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.

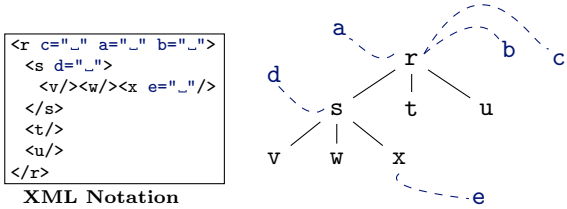


Figure 4: Sample XML Tree with Attributes.

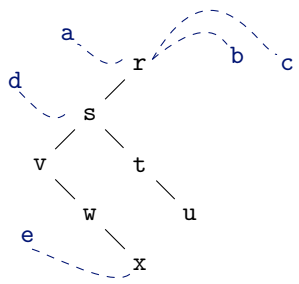


Figure 5: Binary Encoding of Tree of Figure 4.

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

3.2 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 reader can directly use this syntax for encoding formulas as text files to be used with the system described in Section 2 [8]. 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 [9], and we review it informally below through a series of examples.

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

Figure 6: Syntax of Logical Formulas.

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 expressions.

The logic uses programs for navigating in binary trees: the program `1` allows to navigate from a node down to its first successor and the program `2` for 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 1 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

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

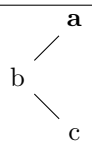
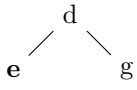
Sample Formula	Tree	XML
`a & <1>b`		`<a><b/></a>`
`a & <1>(b & <2>c)`		`<a><b/><c/></a>`
`e & <-1>(d & <2>g)`		`<d><e/></d><g/>`
`f & <-2>(g & ~<2>T)`	none	none

Table 1: Sample Formulas and Satisfying Trees.

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
```

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).

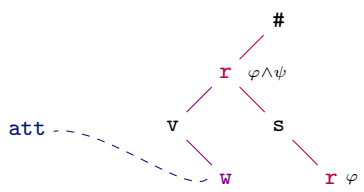
From a theoretical perspective, the recursive binder `let $X` $= \varphi$ `in` φ corresponds to the fixpoint operators of the μ-calculus. It is shown in [9] that the least fixpoint and the greatest fixpoint operators of the μ-calculus coincide over finite tree structures, for a restricted class of formulas called *cycle-free* formulas. Translations of XPath expressions and schemas presented in this paper always yield cycle-free formulas (see [10] for more details).

3.3 Compilation of 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`.



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

Translation:

```
r & (let $X=<-1># | <-2>$X) & <1>let $Y=w & <att>T | <2>$Y
```

φ ψ

Figure 7: XPath Translation Example.

This example illustrates the need for converse programs inside modalities. The translated XPath expression only uses forward axes (child and attribute), nevertheless 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 [9]. 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.

3.4 Compilation of 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 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 [7]. 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. The logical translation of an expression τ is given by the function $\mathrm{tr}(\tau)^{\mathtt{F}}_{\mathtt{T}}$ defined below:

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

where the function $s_{\cdot}(\cdot)$ sets the type frontier:

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

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

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

The function tra(a) compiles attribute expressions associated with element definitions as follows:

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

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 into 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 [10].

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 $\mathrm{tr}(\tau)^{\mathtt{F}}_{\mathtt{T}}$.

3.5 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,

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

one may formulate $\mathrm{tr}(\tau)^{\mathtt{F}}_{\mathrm{xhtml}}$ | $\mathrm{tr}(\tau')^{\mathtt{F}}_{\mathrm{smil}}$ 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:

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

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

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

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 $\mathrm{tr}(\tau)^{\varphi'}_{\varphi}$. Such 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 mean 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, to 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 $\mathrm{tr}(\tau')^{\mathtt{F}}_{\mathtt{T}} \ \&\tilde{}\ \mathrm{tr}(\tau)^{\mathtt{F}}_{\mathtt{T}}$.
- 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 `select("`*query*`",` $\mathrm{tr}(\tau)^{\mathtt{F}}_{\mathtt{T}}$ `& #)` where the predicate `select("`*query*`",` φ`)` compiles the XPath expression *query* into the logic, starting from a context that satisfies φ, as explained in Section 3.3. 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.

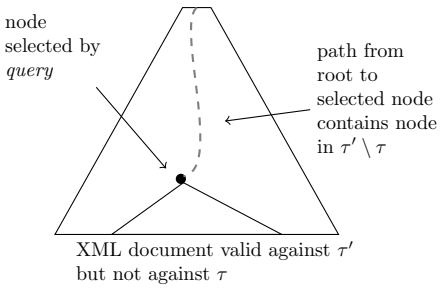
A second, more-elaborated, class of predicates allows formulating problems that combine both a query *query* and two type expressions τ, τ' (where τ' is assumed to be a evolved version of τ):

- `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:

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

  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 `new_region("query"`$,\tau,\tau'$) is logically defined as follows:

$$
\begin{aligned}
&\texttt{new_region("}\mathit{query}\texttt{"},\tau,\tau') \stackrel{\text{def}}{=}\\
&\qquad \texttt{select("}\mathit{query}\texttt{"},\mathrm{tr}(\tau)^{\texttt{F}}_{\texttt{_all}}\ \texttt{\&}\tilde{}\,\mathrm{tr}(\tau')^{\tilde{}\texttt{_old_complement}}_{\texttt{T}})\\
&\qquad\qquad \texttt{\&}\tilde{}\,\texttt{added_element}(\tau,\tau')\\
&\qquad\qquad \texttt{\&}\ \texttt{ancestor(_old_complement)}\\
&\qquad\qquad \texttt{\&}\tilde{}\,\texttt{descendant(_old_complement)}\\
&\qquad\qquad \texttt{\&}\tilde{}\,\texttt{following(_old_complement)}\\
&\qquad\qquad \texttt{\&}\tilde{}\,\texttt{preceding(_old_complement)}
\end{aligned}
$$

The previous definition heavily relies on the partition of tree nodes defined by XPath axes, as illustrated by Figure 8. The definition of `new_region("query"`$,\tau,\tau'$) uses an auxiliary predicate `added_element`(τ,τ') 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 `added_attribute`(φ,φ') builds the disjunction of all attribute names defined in τ' but not in τ.

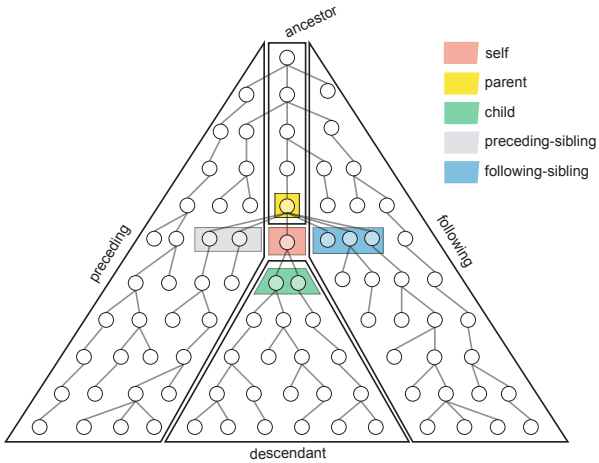
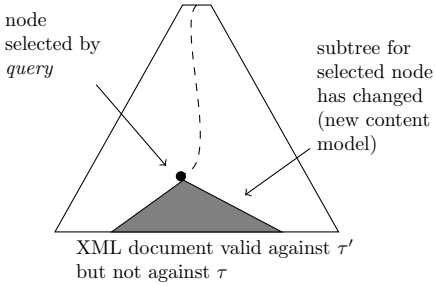


Figure 8: XPath axes: partition of tree nodes.

The predicate `new_region("query"`$,\tau,\tau'$) 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'.

- `new_content("query"`$,\tau,\tau'$) 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:

$$
\begin{aligned}
&\texttt{new_content}(\texttt{"}query\texttt{"}, \tau, \tau') \stackrel{\text{def}}{=} \\
&\qquad \texttt{select}(\texttt{"}query\texttt{"}, \mathrm{tr}(\tau)^{\texttt{F}}_{\texttt{_all}} \ \texttt{\&}\tilde{}\ \mathrm{tr}(\tau')^{\tilde{}\texttt{_old_complement}}_{\texttt{T}}) \\
&\qquad\qquad \texttt{\&}\tilde{}\ \texttt{added_element}(\tau, \tau') \\
&\qquad\qquad \texttt{\&}\tilde{}\ \texttt{ancestor}(\texttt{added_element}(\tau, \tau')) \\
&\qquad\qquad \texttt{\&}\ \texttt{descendant}(\texttt{_old_complement}) \\
&\qquad\qquad \texttt{\&}\tilde{}\ \texttt{following}(\texttt{_old_complement}) \\
&\qquad\qquad \texttt{\&}\tilde{}\ \texttt{preceding}(\texttt{_old_complement})
\end{aligned}
$$

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'.

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:

$$
\begin{aligned}
&\texttt{exclude}(\varphi) \stackrel{\text{def}}{=} \\
&\qquad\qquad \tilde{}\ \texttt{ancestor-or-self}(\texttt{descendant-or-self}(\varphi))
\end{aligned}
$$

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:

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

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

$$
\begin{array}{ll}
\textit{predicate} & ::= \\
& \quad \texttt{select}(\texttt{"}\textit{query}\texttt{"}) \\
& \quad \texttt{select}(\texttt{"}\textit{query}\texttt{"}, \varphi) \\
& \quad \texttt{exists}(\texttt{"}\textit{query}\texttt{"}) \\
& \quad \texttt{exists}(\texttt{"}\textit{query}\texttt{"}, \varphi) \\
& \\
& \quad \texttt{type}(\texttt{"}f\texttt{"}, l) \\
& \quad \texttt{type}(\texttt{"}f\texttt{"}, l, \varphi, \varphi') \\
& \quad \texttt{forward_incompatible}(\varphi, \varphi') \\
& \quad \texttt{backward_incompatible}(\varphi, \varphi') \\
& \\
& \quad \texttt{element}(\varphi) \\
& \quad \texttt{attribute}(\varphi) \\
& \quad \texttt{descendant}(\varphi) \\
& \quad \texttt{exclude}(\varphi) \\
& \quad \texttt{added_element}(\varphi, \varphi') \\
& \quad \texttt{added_attribute}(\varphi, \varphi') \\
& \\
& \quad \texttt{non_empty}(\texttt{"}\textit{query}\texttt{"}, \varphi) \\
& \quad \texttt{new_element_name}(\texttt{"}\textit{query}\texttt{"}, \texttt{"}f\texttt{"}, \texttt{"}f'\texttt{"}, l) \\
& \quad \texttt{new_region}(\texttt{"}\textit{query}\texttt{"}, \texttt{"}f\texttt{"}, \texttt{"}f'\texttt{"}, l) \\
& \quad \texttt{new_content}(\texttt{"}\textit{query}\texttt{"}, \texttt{"}f\texttt{"}, \texttt{"}f'\texttt{"}, l) \\
& \quad \textit{predicate-name}(\langle\varphi\rangle^{\oplus})
\end{array}
$$

Figure 9: Syntax of Predicates for XML Reasoning.

`descendant`(φ) forces the 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,

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 2: Sizes of (Some) Considered Schemas.

but not to the currently defined predicate (recursive definitions must be made through the let binder shown on Figure 6).

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

Figure 10: Global Syntax for Specifying Problems.

5 Framework in Action

We have implemented the whole software architecture described in Section 2 and illustrated on Figure 1 [8]. We have carried out extensive experiments of the system with real world schemas such as XHTML, MathML, SVG, SMIL (Table 2 gives details related to their respective sizes) and queries found in transformations such MathML content to presentation [15]. We present two of them that show how the tool can be used to analyze different situations where schemas and queries evolve.

Evolution of 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")
```

The test immediately 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>
          <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
```

Notice that we observed similar forward and backward compatibility issues with several other W3C normative schemas (in particular for the different versions of SMIL and SVG). Such backward incompatibilities suggests that applications cannot simply ignore new elements from newer schemas, as the combination of older elements may evolve significantly from one version to another.

MathML Content to Presentation Conversion

MathML is an XML format for describing mathematical notations and capturing both its structure and graphical structure, 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:

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

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_regions("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.

All the previous tests were processed in less than 30 seconds on an ordinary laptop computer running Java under Mac OS X.

6 Conclusion

In this article, we present a logical framework and a tool for verifying forward/backward compatibility issues caused by schemas and queries evolution. The tool allows XML designers to identify queries that must be reformulated 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 both schema and query evolution on realistic examples. In particular, we considered typical situations in applications involving W3C schemas evolution such as XHTML and MathML. 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.

There are a number of interesting extensions to the proposed system. First, the set of predicates can be easily enriched to detect more precisely the impact on queries. For example, one can extend the tagging to identify separately every navigation step and qualifier in a query expression. This will help greatly in the identification and reformulation of the navigation steps or qualifiers affected by schemas evolution.

References

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

[2] M. Benedikt and C. Koch. XPath leashed. submitted, 2006.

[3] G. J. Bex, W. Martens, F. Neven, and T. Schwentick. Expressiveness of XSDs: from practice to theory, there and back again. In *WWW '05*, pages 712–721, 2005.

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

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

[6] A. Deutsch and V. Tannen. XML queries and constraints, containment and reformulation. *Theor. Comput. Sci.*, 336(1):57–87, 2005.

[7] P. Genevès. *Logics for XML*. PhD thesis, Institut National Polytechnique de Grenoble, December 2006. http://www.pierresoft.com/pierre.geneves/phd.htm.

[8] P. Genevès and N. Layaïda. A satisfiability solver for XML and XPath, June 2006. http://wam.inrialpes.fr/xml.

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

[10] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. Long version of [9], Research Report 6590, INRIA, July 2008.

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

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

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

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

[15] E. Pietriga. MathML content2presentation transformation, May 2005. http://www.lri.fr/˜pietriga/mathmlc2p/mathmlc2p.html.

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

[17] P. Wadler. 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, January 2000.

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

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399