HAL
open science

Toward Automated Schema-directed Code Revision

Raquel Oliveira, Pierre Genevès, Nabil Layaïda

To cite this version:

Raquel Oliveira, Pierre Genevès, Nabil Layaïda. Toward Automated Schema-directed Code Revision. DocEng 2012, Sep 2012, Paris, France. 10.1145/2361354.2361377 . hal-00734678

HAL Id: hal-00734678

https://inria.hal.science/hal-00734678

Submitted on 24 Sep 2012

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.

Toward Automated Schema-Directed Code Revision

Raquel Oliveira
Inria/LIG
655 avenue de l'Europe,
38334 Saint Ismier, France
raraujo.oliveira@gmail.com

Pierre Genevès
CNRS and Inria/LIG
655 avenue de l'Europe,
38334 Saint Ismier, France
pierre.geneves@inria.fr

Nabil Layaïda
Inria/LIG
655 avenue de l'Europe,
38334 Saint Ismier, France
nabil.layaida@inria.fr

ABSTRACT

Updating XQuery programs in accordance with a change of the input XML schema is known to be a time-consuming and error-prone task. We propose an automatic method aimed at helping developers realign the XQuery program with the new schema. First, we introduce a taxonomy of possible problems induced by a schema change. This allows to differentiate problems according to their severity levels, e.g. errors that require code revision, and semantic changes that should be brought to the developer's attention. Second, we provide the necessary algorithms to detect such problems using a solver that checks satisfiability of XPath expressions.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*Query languages*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*

Keywords

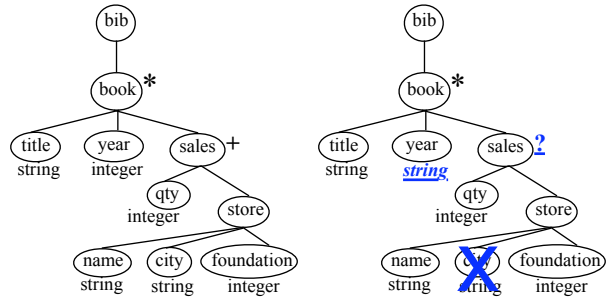
XML, Schemas, XQuery, Schema evolution

1. INTRODUCTION

In document management systems, documents usually encoded in XML are often rendered into output formats (e.g. HTML, SVG, PDF) using transformations, typically written in XSLT or XQuery.[1] XML documents conform to constraints expressed with schemas that continuously change in order to cope with the natural evolution of the entities they describe. However, these changes may break transformations for documents whose structure was described by the original schema.

A frequent scenario consists in an XQuery transformation executed over an XML document (conforming to some schema S_{in}), generating an XML document that is valid against another schema $\mathsf{S}_{\mathsf{out}}$. For instance, consider a schema describing bibliographical data for a writer (as illustrated in Figure 1a). The schema allows zero or more books, each book having a title of type string and a year of publication of type integer, and some information about its sales.

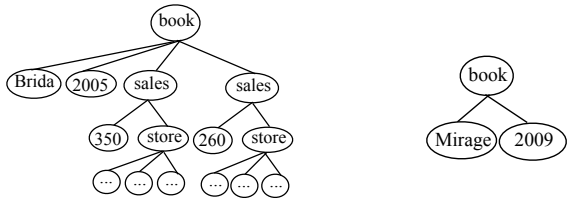
Now consider that this schema S_{in} evolved into a newer version $\mathsf{S}'_{\mathsf{in}}$ (as illustrated in Figure 1b), with three changes: (I) the type of the year is now a string (II) the book is distributed at most once and (III) the *city* element is removed. Figure 2 gives two document instances. Instance (a) is valid



(a) S_{in}=bibliography data (b) $\mathsf{S}'_{\mathsf{in}}$=evolved schema

Figure 1: Example of a schema evolution.

against S_{in}, but not valid against $\mathsf{S}'_{\mathsf{in}}$ (because at most one sale is expected in $\mathsf{S}'_{\mathsf{in}}$). Instance (b) is allowed by $\mathsf{S}'_{\mathsf{in}}$ whereas it was not allowed by S_{in}, since S_{in} prevented *book* elements to occur without a least one *sale* child.



(a) instance valid againt S_{in} (b) instance valid against $\mathsf{S}'_{\mathsf{in}}$

Figure 2: Instances of documents

The XQuery transformation shown below extracts the sales information for book structures valid against S_{in}. If the book was published in 2012 and was sold in New York, it considers only "new" stores (founded in 2010 or later). For other cases, it considers "old" stores too. Finally it returns the sales only for books that were distributed at most once.

[1] It is known that XSLT can be compiled into XQuery.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DocEng'12, September 4–7, 2012, Paris, France.
Copyright 2012 ACM 978-1-4503-1116-8/12/09 ...$15.00.

```
for $s in doc("bib.xml")//book
let $y as xs:integer := xs:integer($s/year)
where
  if (($s/sales/store/city = "New York")
        and ($y = 2012)) then
        $s/sales/store/foundation >= 2010
  else
        $s/sales/store/foundation >= 1960
return
        <book>{$s[count(sales)<=1]/sales}</book>
```

In our scenario, the output document is validated against the schema shown in Figure 3.

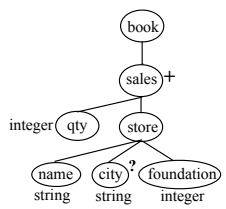


Figure 3: Output schema $\mathsf{S}_{\mathsf{out}}$

Contribution

Our goal is to propose a framework to detect impacts of schema changes on transformations automatically.

2. TAXONOMY OF ERRORS

We now introduce a taxonomy of impacts on transformations according to various changes brought to a given schema. The taxonomy is divided into two main groups. The first group gathers different errors that can be detected by a joint analysis of the transformations and the input schemas. The second group consists in extending the analysis by taking advantage of the availability of the output schema. Basically, the goal is to check whether the transformation results would still be valid against the output schema. The taxonomy categories are summarized in Table 1. In our setting, the transformation analysis results in one of the two following conclusions: either no problem was detected among the ones we cover in the taxonomy, or a problem is detected, in which case it is properly categorized.

Schema considered	Acronym	Category
Input Schemas	TYPECAST	Type casting
	DEADCODE	Dead code
	REDUNDCODE	Redundant code
	INVQUERY	Invalid query
	DIFFRES	Different results
Output Schema	INVRES	Invalid results

Table 1: Summary of the taxonomy

We detail each category in the following subsections.

2.1 Analysis w.r.t. input schema versions

Type casting problems. Type casting problems can occur in XQuery clauses that bind variables (e.g. *for* and *let* clauses). Specifically, a type casting error is detected whenever the schema change results in an assignment of an element with type t_1 to a variable with type $t_2 \neq t_1$. For instance, in our motivating example, this problem occurs in the binding between the XPath expression *//book/year* and the integer variable *$y*, since the *year* element is of type string in the new schema $\mathsf{S}'_{\mathsf{in}}$.

Dead code. A given code portion is dead whenever it is never reached by any execution of the program, for example, the *else* part of a condition when it is known that the *if* part always evaluates to true (or the *then* part whenever the *if* part is found unsatisfiable). In a XQuery program, there are several places where a dead code can potentially occur, as studied in [3]. We detect errors reported in [3] as well as other categories not covered in [3]. In the motivating example, dead code can be detected in the *if* expression. Once we detected that this condition always return an empty set (because the *city* element was removed from the schema), the *then* part will never be executed.

Redundant code. A code portion is redundant if its execution does not affect the result of the program. It can be safely removed from the program, like a condition that is always evaluated to true. The relevance of the detection of this problem resides on the possibility to propose query rewriting, with the purpose of optimization and/or size reduction of a program. In our motivating example, the XPath qualifier in the *return* clause becomes redundant following the schema change. In the new schema, the *sales* element can now occur at most once in the document. As a result, the condition of the qualifier is always true under this schema, making it removable.

Invalid query. We identify as invalid the XPath expressions that are unsatisfiable under the considered schema. Using the satisfiability algorithm proposed in [5], we check statically (at compile time) whether an XPath is satisfiable against schema $\mathsf{S}'_{\mathsf{in}}$. Otherwise, we deduce that the XPath (and subsequently the expression that uses it) is invalid. The XPath used in the *if* condition of the motivating example is invalid. It is never satisfied given that the *city* element was removed from the schema.

Different query results. In this category, we are interested in highlighting to the user at compile time the fact that, due to the schema evolution, the result of the query will change unexpectedly. The query will not raise type casting problems, does not contain neither dead nor redundant code, but it will select unexpectedly more (or less) nodes in the input XML document, due to, for example, insertion (or removal) of elements in the XML schema, or changes in the constraints of the elements, or in its content model. In our motivating example, once we remove the *city* element from the schema, the content model of the *store* element changes (it is somehow "decreased"). Since *store* is part of the content model of the element that will be returned, we can conclude that the result of this XQuery is affected by the schema change (knowing that the *return* clause creates a document consisting of the selected element of the XPath and its content model).

2.2 Analysis w.r.t. the output schema

Invalid query results. When the output schema is available ($\mathsf{S}_{\mathsf{out}}$), we can go further and detect at compile time that the changes in the input schema will produce an invalid document with respect to the output schema $\mathsf{S}_{\mathsf{out}}$. First, we review the validation process of the output document before considering the input schema changes. The XQuery of our example returns the sales information of books which were

sold at most once. This XQuery is launched on instances of the schema S_{in} of Figure 1a. This schema allows instances of books with at least one sale ($< + >$ constraint at the element *sales*). When the transformation is executed, only books with **exactly** one sale will be returned, which is a valid result considering $\mathsf{S}_{\mathsf{out}}$ ($< + >$ constraint at the element *sales*). Now consider the input schema change (II): the book will be distributed at most once (*sales?*). Now this schema allows instances of books with no sale. When we run the XQuery transformation, the condition of the *return* clause will also allow such kind of documents. When considering the output schema of Figure 3 we observe that each book should have at least one sale ($< + >$ constraint at the element *sales*). So, the XQuery transformation of our example may generate invalid results against $\mathsf{S}_{\mathsf{out}}$.

3. ERROR DETECTION

We present the analysis technique for detecting errors for each taxonomy category. For this purpose, we develop inference rules as a detection mechanism for a fragment of XQuery defined below.

Considered XQuery fragment

Specifically, we consider the fragment of XQuery whose syntax is defined in Table 2, that uses XPath expressions whose syntax is described in Table 3. In these grammars, n stands for any integer number, v for any variable's name, *tag* for any string corresponding to the name of a tag and *cst* for any constant value. For *FLWOR* expressions, we consider *for*, *let*, *where* and *return* clauses. The considered fragment captures the most important features for extracting and generating information.

$$
\begin{array}{rl}
e ::= & () \\
 & \mid \$v \\
 & \mid xpath \\
 & \mid \$v/xpath \\
 & \mid e,e \\
 & \mid <tag>\ e\ </tag> \\
 & \mid \text{element}\{e\}\{e\} \\
 & \mid \text{if } e \text{ then } e \text{ else } e \\
 & \mid \text{let } \$v \text{ as } t := e \text{ (where } e\text{)? return } e \\
 & \mid \text{for } \$v \text{ (as } t\text{)? in } e \text{ (where } e\text{)? return } e \\
t & ::= \text{xs:integer} \mid \text{xs:string} \mid \text{xs:boolean}
\end{array}
$$

Table 2: Syntax of XQuery Programs

$$
\begin{array}{rl}
xpath & ::= step \mid xpath/xpath \mid xpath[qualifier] \\
step & ::= axis::nameTest \\
qualifier & ::= xpath \mid xpath\ op\ cst \mid \text{count}(xpath)\ op\ n \\
op & ::= < \mid > \mid \leq \mid \geq \mid = \mid ! = \mid \text{eq}|\text{ne}|\text{lt}|\text{le}|\text{gt}|\text{ge} \\
axis & ::= \text{self} \mid \text{child} \mid \text{parent} \mid \text{descendant} \mid \text{ancestor} \\
 & \mid \text{following} \mid \text{preceding} \mid \text{following-sibling} \\
 & \mid \text{preceding-sibling} \\
nameTest & ::= tag \mid *
\end{array}
$$

Table 3: Syntax of XPath Expressions

Inference Rules

For each category of the taxonomy, we develop a set of inference rules that are applied recursively. Specifically, we design one inference rule per construct of our XQuery fragment shown on Table 2. A given rule tests for the presence of an error by invoking logical predicates in the premises. A logical predicate corresponds to a property involving XPath queries, schemas and more generally constraints over XML document trees. The truth status of these logical predicates is evaluated using calls to an external XML reasoner such as the one proposed in [5]. Depending on the truth status of the properties described in the premises, the conclusion of a rule attaches the detected error to the corresponding XQuery subexpression. We only detail rules for cases where an error is detected, other obvious cases are omitted.

Type casting problems. In order to detect type casting errors in variable bindings of the *for* and *let* clauses, we use an environment Γ' that keeps track of the **type** of variable *$v*. The main rule is shown in the table below.

TYPECAST

for	$\dfrac{\mathsf{S}_{\mathsf{in}},\mathsf{S}'_{\mathsf{in}},\Gamma,\Gamma' \cup (t,\$v) \vdash has_different_type(xpath,\mathsf{S}'_{\mathsf{in}},t)}{\mathsf{S}_{\mathsf{in}},\mathsf{S}'_{\mathsf{in}},\Gamma,\Gamma' \vdash for\ \$v\ as\ t\ in\ xpath\ return\ e_3 \rightarrow TYPECAST}$

This rule invokes the predicate *has_different_type*, that checks if an element selected by an XPath is of type t in the considered schema. This predicate formulates the property that the type of an element returned by the XPath is different from the one expected by the schema.

Redundant code. For checking redundant code, we introduce the predicate *removable_qualifier*. It successively removes qualifiers from the XPath expression one by one and checks for the equivalence with the original one (meaning that such a qualifier is redundant). To check for equivalence, we use the predicate *non_equivalence(xpath1, xpath2)* that is satisfiable iff there exists an element which is selected by one of the XPaths and not by the other one.

REDUNDCODE

xpath	$\dfrac{\mathsf{S}_{\mathsf{in}},\mathsf{S}'_{\mathsf{in}},\Gamma \vdash removable_qualifier(xpath)}{\mathsf{S}_{\mathsf{in}},\mathsf{S}'_{\mathsf{in}},\Gamma \vdash xpath \rightarrow REDUNDCODE}$

Invalid query. The rules for detecting invalid queries are based on satisfiability tests for each XPath, directly available in any XML reasoner, that we denote here through the predicates *select* and *type*. The rule concludes that a given XQuery expression is invalid with respect to an input schema iff it contains XPath expressions that are unsatisfiable in the presence of the schema (i.e. always empty).

INVQUERY

xpath	$\dfrac{\mathsf{S}_{\mathsf{in}},\mathsf{S}'_{\mathsf{in}},\Gamma \vdash \neg select(xpath,type(\mathsf{S}'_{\mathsf{in}}))}{\mathsf{S}_{\mathsf{in}},\mathsf{S}'_{\mathsf{in}},\Gamma \vdash xpath \rightarrow INVQUERY}$

Different query results. For this category, the rules expressed in the table bellow focus on analyzing parts of XQuery expressions that, combined with elements and document constructors, generate content dynamically. For example, in a $< if\ e_1\ then\ e_2\ else\ e_3 >$ expression, only e_2 and e_3 are evaluated, since they can generate new content depending on the boolean condition. In the inference rules, we use the predicates *new_regions* and *new_contents* introduced in [4]. The first one returns true when an XPath selects nodes that were already present in the old schema, but that now appear in different regions of the document, due to the schema changes. The second predicate returns true when the XPath selects elements that already occurred

in the old schema, but whose content model has changed. The main rules are shown in the table below.

DIFFRES

xpath	$\frac{\mathsf{S_{in}},\mathsf{S'_{in}},\Gamma \vdash new_regions(xpath,\mathsf{S_{in}},\mathsf{S'_{in}})}{\mathsf{S_{in}},\mathsf{S'_{in}},\Gamma \vdash xpath \rightarrow DIFFRES}$
xpath	$\frac{\mathsf{S_{in}},\mathsf{S'_{in}},\Gamma \vdash new_contents(xpath,\mathsf{S_{in}},\mathsf{S'_{in}})}{\mathsf{S_{in}},\mathsf{S'_{in}},\Gamma \vdash xpath \rightarrow DIFFRES}$
for	$\frac{\mathsf{S_{in}},\mathsf{S'_{in}},\Gamma\cup(\$v,e_1) \vdash e_3 \rightarrow DIFFRES}{\mathsf{S_{in}},\mathsf{S'_{in}},\Gamma \vdash for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3 \rightarrow DIFFRES}$

Invalid query results. To detect invalid query results, it is necessary to statically analyze whether the document produced by the transformation validates against the output schema $\mathsf{S_{out}}$. We know that the structure of the output document is influenced by changes brought to input schema. For example, assume that we launch our XQuery on some document that follows $\mathsf{S_{in}}$ of Figure 1a. In this case, the output document will follow the structure described by $\mathsf{S_{out}}$ of the Figure 3.

In order to check if the generated document remains valid against $\mathsf{S_{out}}$ whenever $\mathsf{S_{in}}$ changes, we proceed in two steps. First, we abstract over the structure generated by the XQuery transformation using a set of XPath expressions that describe the location of nodes created in the output document. Those XPath expressions are collected during the application of the inference rules. As a second step, we check whether these XPath expressions are unsatisfiable against the output schema (using the "select" predicate), in which case we raise the INVQUERY error.

INVRES

xpath	$\frac{}{\Gamma \vdash xpath \mapsto /selected_node(xpath)}$
seq	$\frac{\Gamma \vdash e_1 \mapsto p \quad \Gamma \vdash e_2 \mapsto p'}{\Gamma \vdash e_1,e_2 \mapsto p/following_simbling::*[p']}$
tag	$\frac{\Gamma \vdash e \mapsto p}{\Gamma \vdash <tag>e</tag> \mapsto /tag/p}$
elem	$\frac{\Gamma \vdash e_1 \mapsto p \quad \Gamma \vdash e_2 \mapsto p'}{\Gamma \vdash element\{e_1\}\{e_2\} \mapsto /p/p'}$
if	$\frac{\Gamma \vdash e_2 \mapsto p \quad \Gamma \vdash e_3 \mapsto p'}{\Gamma \vdash if\ e_1\ then\ e_2\ else\ e_3 \mapsto (p\|p')}$
for	$\frac{\Gamma\cup(\$v,e_1) \vdash e_3 \mapsto p}{\Gamma \vdash for\ \$v\ in\ e_1\ where\ e_2\ return\ e_3 \mapsto p}$
e	$\frac{\mathsf{S_{out}},\Gamma \vdash e \mapsto p \quad \mathsf{S_{out}},\Gamma \vdash \neg select(p,type(\mathsf{S_{out}}))}{\mathsf{S_{out}},\Gamma \vdash e \rightarrow INVRES}$

We introduce the predicate *selected_node* that takes an XPath as argument and returns the selected node name (for example, *selected_node(/a/b/c)=c*). Consider the *return* clause of our motivating example:

return <book>{$s[count(sales)<=1]/*sales*}</book>

The XPath that abstracts over the generated structure for this *return* clause would be: */book/sales*. In the inference rules above, the construction of an XPath p from the XQuery expression e is denoted as $e \mapsto p$ (see table INVRES).

4. RELATED WORK

Impacts of schema evolutions have been recently investigated in the literature by several authors [8, 7, 1, 2, 6, 4].

In [8] the authors address the problem of schema evolution assuming that a mapping between schemas is provided. In [7] the authors propose a five-level framework (called XCase) to manage XML evolution, but they do not deal with propagation of schema changes at query level. [1] is concerned with a structural similarity measure as a step for the automatic inference of a transformation from one schema to another. However the approach is approximate and does not deal with XQuery. The work found in [2] addresses evolution in schemas but the process is not fully automatic in the sense that the user must indicate which queries are potentially affected by schema changes. The work found in [6] deals with revalidation of documents whenever their schema evolves but does not deal with queries. Finally, the present work extends our previous results on evolution restricted to XPath expressions and schemas analysis [4]. The extension consists in considering a transformation language such as XQuery that can not only select nodes in input documents but also generate output documents. In addition, we analyze the impact of input schema changes over the output generated by the transformation.

5. CONCLUSION

We highlighted various consequences that schema changes may have on transformations. We presented a new taxonomy of possible problems induced by schema changes. We proposed automated static analysis techniques to detect each category. The analyses are presented through inference rules that operate on a core fragment of XQuery. We believe this is a step toward automated schema-directed code revision techniques for modern document management systems.

6. REFERENCES

[1] A. Boukottaya and C. Vanoirbeek. Schema matching for transforming structured documents. In *Proceedings of the 2005 ACM symposium on Document engineering*, pages 101–110, 2005.

[2] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: the PRISM workbench. *Proc. VLDB Endow.*, 1(1):761–772, Aug. 2008.

[3] P. Genevès and N. Layaïda. Eliminating dead-code from XQuery programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, pages 305–306, 2010.

[4] P. Genevès, N. Layaïda, and V. Quint. Impact of XML schema evolution. *ACM Transactions on Internet Technology*, 11(1):4:1–4:27, July 2011.

[5] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. *SIGPLAN Not.*, 42(6):342–351, June 2007.

[6] G. Guerrini, M. Mesiti, and D. Rossi. Impact of XML schema evolution on valid documents. In *Proceedings of the 7th annual ACM international workshop on Web information and data management*, pages 39–44, 2005.

[7] M. Necaský and I. Mlýnková. Five-level multi -application schema evolution. In *DATESO*, pages 90–104, 2009.

[8] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping adaptation under evolving schemas. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, pages 584–595, 2003.