# An Algorithm for Transforming XPath Expressions According to Schema Evolution

Kazuma Hasegawa
Graduate School of Library, Information and Media Studies
University of Tsukuba
s1221595@u.tsukuba.ac.jp

Kosetsu Ikeda
Graduate School of Library, Information and Media Studies
University of Tsukuba
lumely@slis.tsukuba.ac.jp

Nobutaka Suzuki
Faculty of Library, Information and Media Science
University of Tsukuba
nsuzuki@slis.tsukuba.ac.jp

## ABSTRACT

XML is a de-fact standard format on the Web. In general, schemas of XML documents are continuously updated according to changes in real world. If a schema is updated, then query expressions have to be transformed so that they are "valid" under the updated schema, since the expressions are no longer valid under the updated schema due to the schema update. However, this is not an easy task since many of recent schemas are large and complex and thus it is becoming difficult to know how to update the query expressions correctly. In this paper, we propose an algorithm for transforming XPath expressions according to schema evolution. For an XPath expression $p$ and a schema $S$, our algorithm treats both $p$ and $S$ as tree automata $TA_p$ and $TA_S$, respectively. Our algorithm first takes the product automaton $TA_R$ of $TA_p$ and $TA_S$, then analyze $TA_R$ to find the correspondence between the states of $TA_p$ and $TA_S$. Based on this correspondence, the algorithm transforms $TA_p$ according to an update operation applied to $TA_S$. We also show some preliminary experimental results.

## Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design—*Data models*

## General Terms

Algorithms

## Keywords

XML, XPath, schema evolution, tree automaton

## 1. INTRODUCTION

XML[5] is a de-fact standard format on the Web. An XML document is usually stored with its schema so that the structural consistency of the document is ensured. In general, schemas are continuously updated according to changes in real world, which is called *schema evolution*. If a schema is updated, then query expressions have to be transformed so that they are "valid" under the updated schema, since the expressions are no longer valid against the updated schema due to the schema update. However, this is not an easy task since many of recent schemas are large and complex, and thus it is becoming very difficult to know how to update the query expressions correctly.

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License (CC BY-SA 3.0). To view a copy of the license, visit http://creativecommons.org/licenses/by-sa/3.0/.

*DChanges 2013,* September 10th, 2013, Florence, Italy.
ceur-ws.org Volume 1008, http://ceur-ws.org/Vol-1008/paper4.pdf .

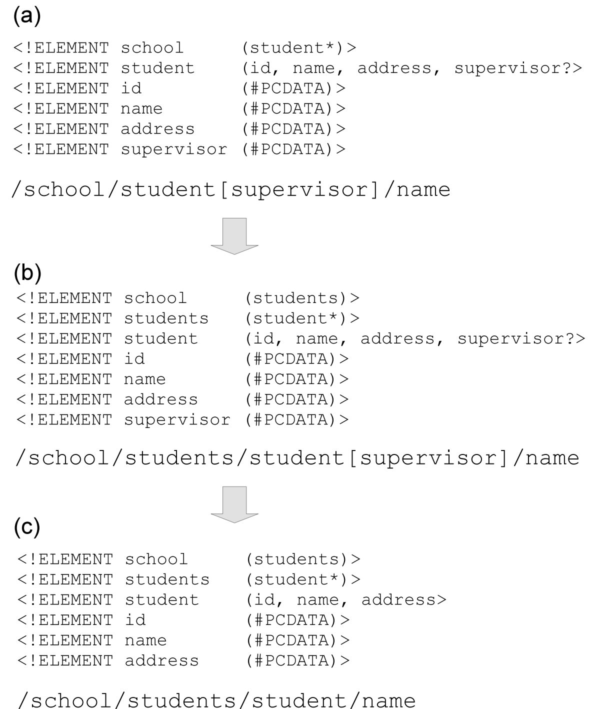


**Figure 1: An example of XPath transformations**

In this paper, we propose an algorithm for transforming XPath expressions according to schema evolution. Here, XPath[4] is the most popular query language for XML. For a given schema $S$, an edit operation *op* to $S$, and an XPath expression $p$ under $S$, our algorithm transforms $p$ into an XPath expression $p'$ "equivalent" to $p$ whenever possible, that is, the result of $p'$ under $op(S)$ coincides with that of $p$ under $S$, where $op(S)$ is the updated schema obtained by applying *op* to $S$.

To illustrate our algorithm, let us show a simple example. Let $D$ be the DTD in Fig. 1(a), and suppose that element

`students` is inserted as the child of `school` (Fig. 1(b)). According to this schema update, our algorithm transforms the following XPath expression

```
/school/student[supervisor]/name
```

into the following:

```
/school/students/student[supervisor]/name.
```

Then, suppose that element `supervisor` is deleted from the DTD in Fig. 1(b). In this case, it is impossible to transform the above XPath expression into an equivalent one. Thus, as an alternative answer our algorithm deletes `supervisor` from the above expression and returns the following:

```
/school/students/student/name.
```

Although these DTDs are very small, schemas used in practice are much larger and complex [6]. Therefore, a user tends not to understand the entire structure of a schema exactly, and thus our algorithm is helpful to transform XPath expressions appropriately according to schema evolutions.

In this paper, schema is modeled as (unranked) tree automaton, which is equivalent to regular tree grammar. Tree automaton is the formal model of RELAX NG, and XML Schema and DTD can also be modeled by tree automaton [14]. Moreover, tree automaton is equivalent to specialized DTD [17]. Thus, our algorithm is applicable to most of formal and practical XML schema languages. As for XPath, we focus on an XPath fragment using child and descendant-or-self axes with predicates. Although our XPath fragment supports no upward axes, this gives usually little problem since the majority of XPath queries uses only downward axes[10]. Thus, we believe that our algorithm is useful to correct a large number of XPath queries.

## Related Work

The study most related to this paper is [13]. This study proposes an algorithm for transforming XPath expressions according to schema evolution, assuming that a schema monotonically increases (no element can be deleted from a schema). On the other hand, this paper has no such an assumption and allows more general schema updates. Since actual schema evolutions usually involve element deletions or some similar updates, our algorithm is more practical in real world situations. To the best of our knowledge, there is no study on transforming XPath expressions according to schema evolution, except [13]. However, several studies deal with update operations to schemas. For example, Ref. [18] proposes a "complete" set of update operations to DTDs. Refs. [12, 9] propose update operations schemas and algorithms for extracting "diff" between two schemas. Refs. [7, 8, 19] propose update operations that assures any updated schema contains its original schema. Recently, Ref. [16] introduces a taxonomy of possible problems induced by a schema change, and gives an algorithm to detect such problems. Ref. [11] studies query-update independence analysis, and shows that the performance of [3] can be drastically enhanced in the use of $\mu$-calculus.

# 2. DEFINITIONS

In this section, we define tree automaton and the product of tree automata.

An XML document is modeled as a labeled ordered tree, and a schema is modeled as a tree automaton. Formally, a *tree automaton* is a quadruple $TA = (N, \Sigma, s, P)$, where

- $N$ is a set of *states* (element types),
- $\Sigma$ is a finite set of *element names*,
- $s \in N$ is the *start state*,
- $P$ is a set of *transition rules* of the form $X \to a(reg)$ or $X \to Y$, where $X, Y \in N$ and $reg$ is a regular expression over $N$.

For a transition rule $X \to a(reg)$, we say that $X$ is the *left-hand side*, $a(reg)$ is the *right-hand side*, $a$ is the *label*, and $reg$ is the *content model* of the rule. For example, consider the tree automaton $TA_a$ shown in Fig. 2. This is equivalent to the DTD in Fig. 1(a). "School" in $N$ is the type of element "school" , and so on. We assume that element "pcdata" represents an arbitrary string. By $L(TA)$ we mean the *language* of tree automaton $TA$, i.e., the set of trees "valid" against $TA$.

Following [15], we define the *product* of tree automata. Let $TA_1 = (N_1, \Sigma_1, s_1, P_1)$ and $TA_2 = (N_2, \Sigma_2, s_2, P_2)$ be tree automata. Without loss of generality, we assume that $\Sigma_1 = \Sigma_2 = \Sigma$. First, we define the product $reg_1 \oplus reg_2$, where $reg_1$ is a regular expression over $N_1$ and $reg_2$ is a regular expression over $N_2$. Then $reg_1 \oplus reg_2$ is a regular expression over $N_1 \times N_2$ such that $n_1^1 n_1^2 \cdots n_1^i$ matches $reg_1$ and $n_2^1 n_2^2 \cdots n_2^i$ matches $reg_2$ if and only if $[n_1^1, n_2^1][n_1^2, n_2^2] \cdots [n_1^i, n_2^i]$ matches $reg_1 \oplus reg_2$, where $n_1^1 n_1^2 \cdots n_1^i$ is a sequence of states in $N_1$ and $n_2^1 n_2^2 \cdots n_2^i$ is a sequence of states in $N_2$. $reg_1 \oplus reg_2$ is constructed as follows.

1. $reg_1'$ is obtained from $reg_1$ by replacing each state $n$ by $[n_1, n_2^1]|[n_1, n_2^2]| \cdots |[n_1, n_2^k]$, where $n_2^1, n_2^2, \cdots, n_2^k$ is an enumeration of $N_2$. Similarly, $reg_2'$ is obtained from $reg_2$.
2. Construct two automata $A_1$ and $A_2$ from $reg_1'$ and $reg_2'$, respectively.
3. Construct the product automaton of $A_1$ and $A_2$.
4. Construct a regular expression, $reg_1 \oplus reg_2$, from the product automaton.

The product automaton of $TA_1$ and $TA_2$ is $TA_3 = (N_1 \times N_2, \Sigma, s_1 \times s_2, P_3)$, where

$$
\begin{aligned}
P_3 \quad = \quad & \{[n_1, n_2] \to a(reg_1 \oplus reg_2) \mid (n_1 \to a(reg_1) \in P_1, \\
& n_2 \to a(reg_2) \in P_2) \\
& \cup \{[n_1, n_2] \to [n_1, n_2'] \mid (n_1 \in N_1, n_2 \to n_2' \in P_2)\} \\
& \cup \{[n_1, n_2] \to [n_1', n_2] \mid (n_1 \to n_1' \in P_1, n_2 \in N_2)\}\}.
\end{aligned}
$$

By definition, it is immediate that for any tree $t$, $t \in L(TA_3)$ if and only if $t \in L(TA_1)$ and $t \in L(TA_2)$.

Let $\Sigma$ be a set of element names. We define XPath expression $p$ as follows.

- $p ::= /p'$
- $p' ::= \chi :: l \mid p'/p' \mid p'[q]$, where $l \in \Sigma$
- $\chi$ ::=child | descendant-or-self
- $q ::= p'$

We call each $\chi \in \{\text{child}, \text{descendant-or-self}\}$ *axis*, and $q$ *predicate*, respectively.

$$N = \{\text{School}, \text{Student}, \text{ID}, \text{Name}, \text{Address}, \text{Supervisor}, \text{Pcdata}\},$$
$$\Sigma = \{\text{school}, \text{student}, \text{id}, \text{name}, \text{address}, \text{supervisor}, \text{pcdata}\},$$
$$P = \{\text{School} \to \text{school}(\text{Student}^*), \text{Student} \to \text{student}(\text{ID Name Address Supervisor?}), \text{ID} \to \text{id}(\text{Pcdata}), \text{Name} \to \text{name}(\text{Pcdata}), \text{Address} \to \text{address}(\text{Pcdata}), \text{Supervisor} \to \text{supervisor}(\text{Pcdata}), \text{Pcdata} \to \text{pcdata}(\epsilon)\}.$$

**Figure 2: Tree automaton** $TA_a = (N, \Sigma, \text{School}, P)$

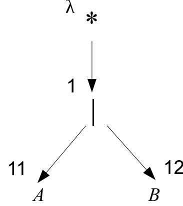


**Figure 3: Tree representation of** $(A|B)^*$

# 3. UPDATE OPERATION TO TREE AUTOMATON

In this section, we define update operations to tree automaton.

Let *reg* be a regular expression. To define update operations to a tree automaton, we need to identify the positions of states and operators in *reg*. Thus we define the set $pos(reg)$ of *positions* in a *reg*, as follows.

- If $reg = \epsilon$ or $reg = a$ $(a \in \Sigma)$, then $pos(reg) = \{\lambda\}$.
- Otherwise, $pos(reg) = \{\lambda\} \cup \{u \mid u = vw, 1 \leq v \leq n, w \in pos(reg_v)\}$, where $reg_v$ is a subexpression consisting of the descendants of $v$ in *reg*.

For example, let $reg = (A|B)^*$. Then $pos(reg) = \{\lambda, 1, 11, 12\}$. As shown in Fig. 3, $\lambda$ is the position of $*$, 1 is the position of $|$, 11 is the position of $A$, and 12 is the position of $B$.

In the following, without loss of generality we assume that for any state $A$ and any element name $a$, there is at most one transition rule whose left-hand side is $A$ and label is $a$. If there are transition rules $A \to a(reg_1)$ and $A \to a(reg_2)$, then these can be merged into one transition rule $A \to a(reg_1|reg_2)$.

We now define update operations to a tree automaton $TA = (N, \Sigma, s, P)$ as follows.

- $ins_state(A, a, B, i)$: Inserts new state $B$ at position $i$ in the content model of $r$, where $r$ is the transition rule in $P$ such that the left-hand side of $r$ is $A$ and that the label of $r$ is $a$.
- $ins_opr(A, a, opr, i)$: Inserts an operator *opr* $(*, +, ?)$ at position $i$ in the content model of $r$, where $r$ is the transition rule in $P$ such that the left-hand side of $r$ is $A$ and that the label of $r$ is $a$.
- $del_state(A, a, i)$: Deletes the state at position $i$ in the content model of $r$, where $r$ is the transition rule in $P$ such that the left-hand side of $r$ is $A$ and that the label of $r$ is $a$.
- $del_opr(A, a, i)$: Deletes the operator at position $i$ in the content model of $r$, where $r$ is the transition rule in $P$ such that the left-hand side of $r$ is $A$ and that the label of $r$ is $a$.
- $nest_state(A, a, B, b, i)$: Replaces the subexpression $E$ at position $i$ (i.e., subtree rooted at position $i$) in content model of $r$ by state $B$, where $r$ is the transition rule in $P$ such that the left-hand side of $r$ is $A$ and that the label of $r$ is $a$. Moreover, a transition rule $B \to b(E)$ is added to $P$.
- $unnest_state(A, a, b, i)$: This is the inverse operation of *nest_state*, and replaces the state $B$ at position $i$ in the content model of $r$ by *reg*, where (1) $r$ is the transition rule in $P$ such that the left-hand side of $r$ is $A$ and that the label of $r$ is $a$ and (2) *reg* is the content model of the transition rule whose left-hand side is $B$ and label is $b$.
- $replace_state(A, a, B, i)$: Replaces the state at position $i$ in the content model of $r$ by $B$. In this paper, this operation is treated as a pair of $unnest_state(A, a, i)$ and $nest_state(A, a, B, b, i)$. This operation is used in order to "rename" element names.

For example, consider the tree automaton $TA_a$ in Fig. 2. Applying $nest_state(\text{School}, \text{school}, \text{Students}, \text{students}, \lambda)$ to $TA_a$, we obtain the tree automaton $TA_b$ in Fig. 4. Then applying $del_opr(\text{Student}, \text{student}, 4)$ and $del_state(\text{Student}, \text{student}, 4)$ to $T_b$, we obtain the tree automaton $TA_c$ in Fig. 5.

# 4. TRANSFORMATION FROM XPATH EXPRESSION INTO TREE AUTOMATON

Our algorithm treats an XPath expression as a tree automaton. Thus, in this section we define a transformation from an XPath expression into a tree automaton. This transformation is based on [15].

An XPath expression is transformed into a tree automaton as follows. First, we transform a location step $\chi :: l$ into a tree automaton. Each transformed tree automaton has *input* and *output* states, corresponding to the "context node" for $\chi :: l$ and the "result node" selected by $\chi :: l$, respectively. For an XPath expression $p$ of the form $p_1/p_2$, $p'[q]$, or $/p'$, we transform $p$ into a tree automaton in a bottom-up manner, according to the structure of $p$.

We first define a tree automaton $(N, \Sigma, s, P)$ corresponding to location step $\chi :: l$. In the following, $A \to \sigma(\cdots)$ denotes $\{A \to \sigma(\cdots) \mid \sigma \in \Sigma\}$.

- If $\chi = \text{child}$, then
  - The initial state $s$ is $B$.

$$
\begin{aligned}
N &= \{\text{School}, \text{Students}, \text{Student}, \text{ID}, \text{Name}, \text{Address}, \text{Supervisor}, \text{Pcdata}\},\\
\Sigma &= \{\text{school}, \text{students}, \text{student}, \text{id}, \text{name}, \text{address}, \text{supervisor}, \text{pcdata}\},\\
P &= \{\text{School} \to \text{school}(\text{Students}), \text{Students} \to \text{students}(\text{Student}^*), \text{Student} \to \text{student}(\text{ID Name Address Supervisor?}),\\
&\quad \text{ID} \to \text{id}(\text{Pcdata}), \text{Name} \to \text{name}(\text{Pcdata}), \text{Address} \to \text{address}(\text{Pcdata}), \text{Supervisor} \to \text{supervisor}(\text{Pcdata}), \text{Pcdata} \to \text{pcdata}(\epsilon)\}.
\end{aligned}
$$

**Figure 4: Tree automaton** $TA_b = (N, \Sigma, \text{School}, P)$

$$
\begin{aligned}
N &= \{\text{School}, \text{Students}, \text{Student}, \text{ID}, \text{Name}, \text{Address}, \text{Pcdata}\},\\
\Sigma &= \{\text{school}, \text{students}, \text{student}, \text{id}, \text{name}, \text{address}, \text{pcdata}\},\\
P &= \{\text{School} \to \text{school}(\text{Students}), \text{Students} \to \text{students}(\text{Student}^*), \text{Student} \to \text{student}(\text{ID Name Address}), \text{ID} \to \text{id}(\text{Pcdata}),\\
&\quad \text{Name} \to \text{name}(\text{Pcdata}), \text{Address} \to \text{address}(\text{Pcdata}), \text{Pcdata} \to \text{pcdata}(\epsilon)\}.
\end{aligned}
$$

**Figure 5: Tree automaton** $TA_c = (N, \Sigma, \text{School}, P)$

- $P$ consists of the following transition rules.
  - $A \to \sigma(A^*)$
  - $B \to \sigma(A^*(B|C)A^*)$
  - $B \to C$
  - $C \to \sigma(A^*DA^*)$
  - $D \to l(A^*)$

- If $\chi =$ descendant-or-self, then
  - Initial state $s$ is $B_1$.
  - $P$ consists of the following transition rules.
    - $A \to \sigma(A^*)$
    - $B_1 \to \sigma(A^*(B_1|C)A^*)$
    - $B_1 \to C$
    - $C \to \sigma(A^*(B_2|D)A^*)$
    - $C \to D$
    - $B_2 \to \sigma(A^*(B_2|D)A^*)$
    - $D \to l(A^*)$

The input state and the finite set of output states of a finite tree automata are $\{C\}$ and $\{D\}$, respectively. $C$ represents the context node and $D$ represents the node selected by the location step $\chi :: l$.

We next define the transformation from an XPath expression $p$ into a tree automaton. We have to consider the case where $p = p_1/p_2$, $p = p[q]$, and $p = /p$. In the following, for a tree automaton $TA_p$ of $p$, $N_p$ denotes the set of states of $TA_p$, $NI_p \subset N_p$ denotes the set of input states of $TA_p$, and $NO_p \subset N_p$ denotes the set of output states of $TA_p$.

### *The case of* $p_1/p_2$

Let $TA_{p_1} = (N_{p_1}, \Sigma, s_{p_1}, P_{p_1})$ be the tree automaton of $p_1$ and $TA_{p_2} = (N_{p_2}, \Sigma, s_{p_2}, P_{p_2})$ be the tree automaton of $p_2$. Moreover, let $TA = (N, \Sigma, s, P)$ be the product automaton of $TA_{p_1}$ and $TA_{p_2}$. Then the automaton $TA_{p_1/p_2}$ of $p_1/p_2$ is defined as $TA_{p_1/p_2} = (N_{p_1/p_2}, \Sigma, s, P)$, where

$$
\begin{aligned}
N_{p_1/p_2} &= \{[n_{p_1}, n_{p_2}] \mid (n_{p_1} \in NO_{p_1}, n_{p_2} \in NI_{p_2}) \vee\\
&\quad (n_{p_1} \in (N_{p_1} - NO_{p_1}), n_{p_2} \in (N_{p_2} - NI_{p_2}))\}.
\end{aligned}
$$

The set $NI_{p_1/p_2}$ of input states and the set $NO_{p_1/p_2}$ of output states of $TA_{p_1/p_2}$ are defined as follows.

$$
\begin{aligned}
NI_{p_1/p_2} &= \{[n_{p_1}, n_{p_2}] \mid n_{p_1} \in NI_{p_1}, n_{p_2} \in (N_{p_2} - NI_{p_2})\},\\
NO_{p_1/p_2} &= \{[n_{p_1}, n_{p_2}] \mid n_{p_1} \in (N_{p_1} - NO_{p_1}), n_{p_2} \in NO_{p_2}\}.
\end{aligned}
$$

### *The case of* $p[q]$

Let $TA_p = (N_p, \Sigma, s_p, P_p)$ be the tree automaton of $p$ and $TA_q = (N_q, \Sigma, s_q, P_q)$ be the tree automaton of $q$. Moreover, let $TA = (N, \Sigma, s, P)$ be the product automaton $TA_p$ and $TA_q$. Then the tree automaton $TA_{p[q]}$ of $p[q]$ is defined as $TA_{p[q]} = (N_{p[q]}, \Sigma, s, P)$, where

$$
\begin{aligned}
N_{p[q]} &= \{[n_p, n_q] \mid (n_p \in NO_p, n_q \in NI_q) \vee\\
&\quad (n_p \in (N_p - NO_p), n_q \in (N_q - NI_q))\}.
\end{aligned}
$$

The set $NI_{p[q]}$ of input states and the set $NO_{p[q]}$ of output states of $TA_{p[q]}$ are defined as follows.

$$
\begin{aligned}
NI_{p[q]} &= \{[n_p, n_q] \mid n_p \in NI_p, n_q \in (N_q - NI_q)\},\\
NO_{p[q]} &= \{[n_p, n_q] \mid (n_p \in NO_p, n_q \in NI_q)\}.
\end{aligned}
$$

### *The case of* $/p$

Let $\chi :: l$ be the first location step in $p$ and let $TA_p = (N_p, \Sigma, s_p, P_p)$ be the tree automaton of $p$. Then the tree automaton $TA_{/p}$ of $/p$ is defined as $TA_{/p} = (N_{/p}, \Sigma, R, P_{/p})$, where

$$
\begin{aligned}
N_{/p} &= N_p \cup \{R\},\\
P_{/p} &= P_p \cup \{R \to root(D)\},
\end{aligned}
$$

where $R$ is the initial state of $TA_{/p}$ and $root$ is the element name corresponding to the root node.

The set $NI_{/p}$ of input states and the set $NO_{/p}$ of output states of $TA_{/p}$ are defined as follows.

$$
\begin{aligned}
NI_{/p} &= \{R\},\\
NO_{/p} &= NO_p,
\end{aligned}
$$

where $NO_p$ is the set of output states of $TA_p$.

## 5. ALGORITHM FOR TRANSFORMING XPATH EXPRESSION ACCORDING TO SCHEMA EVOLUTION

In this section, we show an algorithm for transforming a given XPath expression according to schema evolution.

To describe our algorithm, we need some definitions. For an XPath expression $p$, the *selection path* of $p$ is the XPath expression obtained by dropping every predicate from $p$. Let $p$ be an XPath expression of the form $/p_1[q]/p_2$. Then $/p_1/q$ is called a *predicate path* of $p$ (the predicate path for

a predicate in $q$ can be defined similarly). For example, Let $p = /a[f]/b[c/e]/d$. Then $/a/b/d$ is the selection path of $p$, and $/a/f$ and $/a/b/c/e$ are the predicate paths of $p$.

Let us first show the "main" algorithm (Fig. 6). Let $op$ be the update operation applied to tree automaton $TA$. Moreover, let $p$ be the input XPath expression, $p_0$ be the selection path of $p$ and $p_1, \cdots, p_k$ be the predicate paths of $p$. We transform $p_i$ to $p'_i$ according to $op$, for each $i = 0, 1, \cdots, k$, and merge the resulting $k$ expressions $p'_0, p'_1, \cdots, p'_k$ as the result. More concretely, the algorithm first partition $p$ into $p_0, p_1, \cdots, p_k$ (step 1). Then $p_0$ is transformed into $p'_0$ according to $op$ by function Transform (shown later). If the result of $p_0$ becomes empty due to $del_state$, then $p'_0 = nil$ and the transformation of $p$ is terminated (steps 3 and 4). Otherwise, we transform $p_i$ for each $i = 1, \cdots, k$ (steps 6 to 8), by Transform. Finally, $p_0, p_1, \cdots, p_k$ are merged and returned as the result. If a predicate path $p'_i$ is $nil$, then $p'_i$ is just ignored when merged. For example, let $p = /a[f]/b[c/e]/d$. Then $p_0 = /a/b/c$, $p_1 = /a/f$, and $p_2 = /a/b/c/e$. Suppose that $c$ is deleted by $unnest_state$. Then we obtain $p'_0 = /a/b/d$, $p'_1 = /a/f$, and $p'_2 = /a/b/e$ due to Transform, and merging these three expressions we have $p' = /a[f]/b[e]/d$, which is the result of the algorithm.

Let us next consider function Transform. Let $TA = (N, \Sigma, s, P)$ be a tree automaton, $op$ be an edit operation to $TA$, and $p$ be an XPath expression having no predicate. Our objective is to transform $p$ into an XPath expression $p'$ so that the result of $p'$ under $op(TA)$ coincides with that of $p$ under $TA$. However, this is sometimes impossible if a state (and its corresponding element) is deleted from $TA$. Thus, Transform is constructed as follows.

- If $op$ is $ins_state$, $ins_opr$, or $del_opr$, then $p$ is unchanged.
- If $op$ is $nest_state(A, a, B, b, i)$, then location step child :: $b$ is inserted to $p$ at the position that nesting is occurred, if necessary (i.e., unless $p$ contains a descendant-or-self axis that "masks" the $nest_state$ operation).
- If $op$ is $unnest_state(A, a, b, i)$, then a location step whose node test is $b$ is deleted from $p$, if it is not "masked" by a descendant-or-self axis.
- If $op$ is $del_state$, then $p$ is modified as follows.
  - If all the result elements retrieved by $p$ disappear from $TA$ due to $op$, the result of $p$ becomes empty. Then our algorithm returns $nil$.
  - Otherwise, some element of $op(TA)$ can still be retrieved by $p$. Thus our algorithm transforms $p$ so that such elements are retrieved.

**Input** : XPath expression $p$, tree automaton $TA$, update operation $op$ to $TA$.
**Output** : XPath expression or $nil$

1. Partition $p$ into the selection path $p_0$ and the predicate path $p_1, p_2, \cdots, p_k$.
2. $p'_0 \leftarrow$ Transform$(p_0, TA, op)$;
3. **if** $p'_0 = nil$ **then**
4.   **return** $nil$;
5. **else**
6.   **for** $i \leftarrow 1$ **to** $k$ **do**
7.     $p'_i \leftarrow$ Transform$(p_i, TA, op)$;
8.   **end**
9. Merge $p'_0, p'_1, \cdots, p'_k$ into $p'$.
10. **return** $p'$;

**Figure 6: Main algorithm**

Now we present function Transform (Fig.7). $state(A, a, i, P)$ in steps 16, 23, and 29 denotes the state at position $i$ in the content model of $r$, where $r$ is the transition rule in $P$ such that its left-hand side is $A$ and its label is $a$. If $op$ is $ins_state$, $ins_opr$, or $del_opr$, $p$ is unchanged (steps 4 to 7). If $op$ is $del_state$, we examine whether the result of $p$ becomes empty under $op(TA)$ (steps 8 to 11). If the result of $p$ does not become empty, $p$ is unchanged (steps 11 to 12). If the result of $p$ becomes empty and $p$ is the selection path, the algorithm returns $nil$ (steps 13 to 14). Otherwise (i.e., $p$ is a predicate path), we delete the suffix of $p$ that becomes invalid due to $op$ (steps 15 to 20). In step 17, $P''$ is obtained in step 2, and we say that $r$ is a transition rule from $A$ to $C$ if the left-hand side of $r$ contains $A$ and a state in the content model of $r$ contains $C$. In step 18, we say that $ls_j$ *corresponds* to $r \in P''$ if $r$ is the intersection of (i) the transition rule corresponding to $ls_j$ in $P_p$ and (ii) some transition rule in $P$. If $op$ is $nest_state(A, a, B, b, i)$, we examine whether we need to transform $p$ (steps 23 to 24). If we need to do, we insert a new location step child :: $b$ to $p$ (steps 25 and 26). If $op$ is $unnest_state(A, a, b, i)$, then we also examine whether we need to transform $p$ (steps 30 to 31). Moreover, if the location step $ls_{j+1}$ to be deleted has a predicate $q$ and the axis of $ls_{j+1}$ is descendant-or-self, we delete $q$ (steps 32 to 33). Otherwise, we delete $ls_{j+1}$ from $p$ (steps 34 to 38).

Finally, let us present the time complexity of the algorithm. Let $TA = (N, T, s, P)$ be a tree automaton (schema) $p$ be an XPath expression that is partitioned into $p_0, p_1, p_2, \cdots, p_k$. Then the algorithm runs in $O(k \cdot |p|^2 \cdot |TA|)$, where $|p|$ is the number of location steps in $p$ and $|TA|$ is the size of $TA$.

## 6. EXPERIMENTAL RESULTS

To verify if our algorithm transforms XPath expressions appropriately under real world schemas, we implemented our algorithm (in Ruby) and made a few experimentations. We use two pairs of schemas, MSRMEDOC DTDs (version 2.1.1 and 2.2.2)[2] and the NLM Journal Publishing Tag Set Tag Library DTDs (version 2.3 and 3.0)[1].

First, we give the evaluation of our algorithm on MSRMEDOC DTDs. Let $D_{211}$ be the version 2.1.1 MSRMEDOC DTD and $D_{222}$ be the version 2.2.2 MSRMEDOC DTD. The number of elements of $D_{211}$ is 185 and that of $D_{222}$ is 205. Table 1 shows the number of update operations between $D_{211}$ and $D_{222}$, where "others" are attributes insertions (not supported by our algorithm).

We generate 90 XPath expressions under $D_{211}$ by XQgen[20]. The average size (i.e., number of location steps) of the XPath expressions is 5, where the minimum size is 4 and the maximum size is 7. Each XPath expression uses child axes and at most one descendant-or-self axes (78 XPath expressions use a descendant-or-self axis). There are 5 XPath expressions whose result becomes empty under $D_{222}$. Since there is no predicate in these 90 XPath expressions, we

**Table 1: Update operations between $D_{211}$ and $D_{222}$ and between $D_{23}$ and $D_{30}$**

| | *ins_state* | *del_state* | *nest_state* | *unnest_state* | *replace_state* | *ins_opr*/*del_opr* | others | total |
|---|---|---|---|---|---|---|---|---|
| $D_{211} \to D_{222}$ | 61 | 11 | 27 | 0 | 0 | 60 | 32 | 191 |
| $D_{23} \to D_{30}$ | 504 | 82 | 3 | 0 | 24 | 0 | 120 | 733 |

Function Transform($p, TA, op$)

**Input** : XPath expression $p = /ls_1/\cdots/ls_m$, tree automaton $TA = (N, \Sigma, s, P)$, update operation $op$ to $TA$.
**Output** : XPath expression or *nil*

1. Construct the tree automaton $TA_p = (N_p, \Sigma, s_p, P_p)$ of $p$
2. Construct the product automaton $TA'' = (N_D \times N_p, \Sigma, s_D \times s_p, P'')$ of $TA$ and $TA_p$
3. **switch** $op$
4. **case** $ins_state(A, a, B, i)$
5. **case** $ins_opr(A, a, opr, i)$
6. **case** $del_opr(A, a, i)$
7.   **return** $p$;
8. **case** $del_state(A, a, i)$
9.   Construct the tree automaton $TA' = (N', \Sigma, s', P')$ of $op(TA)$
10.   Construct the product automaton $TA'_p = (N' \times N_p, \Sigma, s' \times s_p, P'_p)$ of $TA'$ and $TA_p$
11.   **if** $L(TA'_p) \neq \emptyset$ **then**
12.     **return** $p$;
13.   **else if** $p$ is the selection path **then**
14.     **return** *nil*;
15.   **else**
16.     $C \leftarrow state(A, a, i, P)$    // $C$ is deleted
17.     **if** $P''$ contains a transition rule $r$ from $A$ to $C$ **then**
18.       Let $ls_j$ be the location step in $p$ corresponding to $r$;
19.       $p \leftarrow /ls_1/\cdots/ls_{j-1}$;
20.     **end**
21.   **end**
22. **case** $nest_state(A, a, B, b, i)$
23.   $C \leftarrow state(A, a, i, P)$    // $C$ is nested
24.   **if** $P''$ contains a transition rule $r$ from $A$ to $C$ **then**
25.     Let $ls_j$ be the location step in $p$ corresponding to $r$;
26.     $p \leftarrow /ls_1/\cdots/ls_j/\text{child} :: b/ls_{j+1}/\cdots/ls_m$;
27.   **end**
28. **case** $unnest_state(A, a, b, i)$
29.   $C \leftarrow state(A, a, i, P)$    $C$ is unnested
30.   **if** $P''$ contains a transition rule $r$ from $A$ to $C$ **then**
31.     Let $ls_j$ be the location step in $p$ corresponding to $r$;
32.     **if** the axis of $ls_{j+1}$ is descendant-or-self **and** $ls_{j+2}$ is the first location step of a predicate **then**
33.       $p \leftarrow /ls_1/\cdots/ls_{j-1}$;
34.     **else**
35.       $axis \leftarrow$ the axis of $ls_{j+1}$;
36.       $l \leftarrow$ the node test of $ls_{j+2}$;
37.       $p \leftarrow /ls_1/\cdots/ls_j/axis :: l/ls_{j+3}/\cdots/ls_m$;
38.     **end**
39.   **end**
40. **end**
41. **return** $p$;

**Figure 7: Function Transform**

choose 4 expressions and add a predicate to each of 4 expressions (given later).

We transform the above 90 XPath expressions by our algorithm. For 85 expressions, the elements retrieved under $D_{222}$ coincides with the elements retrieved under $D_{211}$. For the rest 5 XPath expressions, our algorithm returns *nil* since the results of these expressions become empty under $D_{222}$. These 5 XPath expressions are shown in Table 2 (deleted elements are italicized). Since elements LIST, NOTE, FIGURE, and FORMULA (child elements of REMARK) and an element PRIVATE-CODES (child element of COMPANY-DOC-INFO) are deleted, the results of these XPath expressions become empty (note that if an element is deleted from a content model, then its descendants cannot be retrieved either).

The four XPath expressions with a predicate, denoted $p_1, p_2, p_3, p_4$, are transformed as follows (Table 3).

- Since element REMARK (child element of COMPANY-REVISION-INFO) is deleted, the predicate of $p_1$ is deleted.
- Since element REMARK is deleted, the last location step in the predicate of $p_2$ is deleted.
- Since element REMARK is deleted, the location step corresponding to an element REMARK in $p_3$ is deleted, therefore our algorithm return *nil*.
- Since element L-1 is inserted between P and FT and between P and STD, two location steps L-1 are inserted to $p_4$.

As shown above, we can say that every XPath expression is transformed appropriately by our algorithm, even if *del_state*'s are involved in the schema evolution. The total execution time of the algorithm for the 90 XPath expression and 191 update operations is 9.58 sec, thus it takes an average 0.111 sec per one XPath expression.

We also made a similar experimentation using the NLM Journal Publishing Tag Set Tag Library. Let $D_{23}$ be the version 2.3 The NLM Journal Publishing Tag Set Tag Library DTD and $D_{30}$ be the version 3.0 DTD. The number of elements of $D_{23}$ is 211 and that of $D_{30}$ is 233. Table 1 shows the number of update operations between $D_{23}$ and $D_{30}$.

We generate 97 XPath expressions under $D_{23}$ by XQgen. The average size of the XPath expressions is 6. Each XPath expression uses child axes and at most once descendant-or-self axes, where 95 XPath expressions include descendant-or-self axes. There are 10 XPath expressions whose result becomes empty under $D_{30}$. There is no XPath expression with a predicate, thus we choose 3 XPath expressions and added a predicate to each expression.

We transform the 97 XPath expressions by our algorithm. For 87 expressions, the elements retrieved under $D_{30}$ coincides with the elements retrieved under $D_{23}$. For the rest

10 XPath expressions, our algorithm returns *nil* since the results of these expressions become empty under $D_{30}$, which are listed in Table 4. Since elements citation, contract-num, contract-sponsor in element p are deleted, the results of these XPath expressions become empty. The three XPath expressions with a predicate, denoted $p_5, p_6, p_7$, are transformed as follows (Table 5).

- Since element chem-struct (child element of ack) is deleted, the predicate of $p_5$ is deleted.
- Since element chem-struct-wrapper is renamed to chem-struct-wrap, the predicate of $p_6$ is renamed accordingly.
- Since element custom-meta-wrap is renamed to custom-meta-group, the corresponding location step in $p_7$ is renamed to custom-meta-group.

Again our algorithm seems to work well despite of *del_state* operations. These results suggest that our algorithm can be applied to XPath expressions under real world schema evolutions. The total execution time of the algorithm for the 97 XPath expression and 733 update operations is 77.561 sec, thus it takes an average 0.800 sec per one XPath expression.

# 7. CONCLUSION

In this paper, we proposed an algorithm for transforming an XPath expression according to schema evolution allowing element deletions. However, this is just an ongoing work and we have a lot to do. First, we would like to extend our algorithm so that it can handle more general XPath expressions, e.g., supporting sibling and parent axes. Second, we use only two schema evolutions in our experimentation. Thus we would like to evaluate our algorithm under more real world schemas. Third, our algorithm supports neither attribute nor entity declaration in schemas. These should be incorporated into our algorithm.

# 8. ACKNOWLEDGEMENT

This work is partly supported by Grants-in-aid for Scientific Research (23500110).

# 9. REFERENCES

[1] *"Journal Publishing Tag Set" NLM Journal Archiving and Interchange Tag Suite.* http://dtd.nlm.nih.gov/publishing/.

[2] *"MSR Download" MSR Home.* http://www.msr-wg.de/medoc/downlo.html.

[3] M. Benedikt and J. Cheney. Destabilizers and independence of xml updates. *Proc. VLDB Endow.*, 3(1-2):906–917, 2010.

[4] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon, editors. *XML Path Language (XPath) 2.0 (Second Edition).* http://www.w3.org/TR/xpath20/.

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, editors. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* http://www.w3.org/TR/xml/.

[6] B. Choi. What are real DTDs like? In *Proc. WebDB*, pages 43–48, 2002.

[7] G. Guerrini, M. Mesiti, and D. Rossi. Impact of XML schema evolution on valid documents. In *Proc. WIDM*, pages 39–44, 2005.

[8] K. Hashimoto, Y. Ishihara, and T. Fujiwara. A proposal of update operations for schema evolution in XML databases and their properties on preservation of schema's expressive power. *IEICE Transactions on Information and Systems (Japanese Edition)*, J90-D(4):990–1004, 2007.

[9] K. Horie and N. Suzuki. Extracting differences between regular tree grammars. In *Proc. ACM SAC*, pages 859–864, 2013.

[10] Z. G. Ives, A. Y. Halevy, and D. S. Weld. An XML query engine for network-bound data. *The VLDB Journal*, 11(4):380–402, 2002.

[11] M. Junedi, P. Genevès, and N. Layaïda. Xml query-update independence analysis revisited. In *Proceedings of the 2012 ACM symposium on Document engineering*, DocEng '12, pages 95–98, 2012.

[12] E. Leonardi, T. T. Hoai, S. S. Bhowmick, and S. Madria. DTD-Diff: a change detection algorithm for DTDs. In *Proc. DASFAA*, pages 817–827, 2006.

[13] T. Morimoto, K. Hashimoto, Y. Ishinara, and T. Fujiwara. Translating XPath queries according to XML schema evolution based on the tree-embedding relation. In *IEICE Technical Report, vol.107, no.131, DE2007-40*, pages 109–114, 2007 (in Japanese).

[14] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5:660–704, 2005.

[15] A. Ohno, Y. Ishihara, and T. Fujiwara. Method of analyzing the correspondence between types defined by an XML schema and XPath subexpressions. In *IPSJ SIG-Report, 2009-MPS-75(20)*, pages 1–7, 2009 (in Japanese).

[16] R. Oliveira, P. Genevès, and N. Layaïda. Toward automated schema-directed code revision. In *Proceedings of the 2012 ACM symposium on Document engineering*, DocEng '12, pages 103–106, 2012.

[17] Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proc. ACM PODS*, pages 35–46, 2000.

[18] H. Su, D. Kramer, L. Chen, K. Claypool, and E. A. Rundensteiner. XEM: managing the evolution of XML documents. In *Proc. IEEE RIDE*, pages 103–110, 2001.

[19] N. Suzuki. An edit operation-based approach to the inclusion problem for DTDs. In *Proc. ACM SAC*, pages 482–488, 2007.

[20] Y. Wu, N. Lele, R. Aroskar, S. Chinnusamy, and S. Brenes. XQGen: an algebra-based xpath query generator for micro-benchmarking. In *Proc. CIKM*, CIKM '09, pages 2109–2110, 2009.

**Table 2: XPath expressions whose results become empty under $D_{222}$**

| XPath expression |
|---|
| /MSRREP/MATCHING-DCIS/MATCHING-DCI/REMARK/*LIST*/ITEM |
| /MSRREP/MATCHING-DCIS/MATCHING-DCI/REMARK/*NOTE* |
| /MSRREP/MATCHING-DCIS/MATCHING-DCI/REMARK/*FIGURE*/DESC |
| /MSRREP/MATCHING-DCIS/MATCHING-DCI/REMARK/*FORMULA*/FORMULA-CAPTION/LONG-NAME |
| //ADMIN-DATA/COMPANY-DOC-INFOS/COMPANY-DOC-INFO/*PRIVATE-CODES*/PRIVATE-CODE |

**Table 3: XPath expressions with predicate under $D_{211}$ and $D_{222}$**

| | |
|---|---|
| XPath expression $p_1$: | //DOC-REVISION/COMPANY-REVISION-INFOS/COMPANY-REVISION-INFO[*REMARK*]/COMPANY-REF |
| Transformed result: | //DOC-REVISION/COMPANY-REVISION-INFOS/COMPANY-REVISION-INFO/COMPANY-REF |
| XPath expression $p_2$: | //DOC-REVISION[COMPANY-REVISION-INFOS/COMPANY-REVISION-INFO/*REMARK*]/MODIFICATIONS/MODEFICATION |
| Transformed result: | //DOC-REVISION[COMPANY-REVISION-INFOS/COMPANY-REVISION-INFO]/MODIFICATIONS/MODEFICATION |
| XPath expression $p_3$: | //DOC-REVISION[COMPANY-REVISION-INFOS/COMPANY-DOC-INFO/PRIVATE-CODES/PRIVATE-CODE]/DOC-REVISIONS/DOC-REVISION/COMPANY-REVISION-INFOS/COMPANY-REVISION-INFO/*REMARK* |
| Transformed result: | *nil* |
| XPath expression $p_4$: | //P[FT]/STD |
| Transformed result: | //P[*L-1*/FT]/*L-1*/STD |

**Table 4: XPath expression whose results become empty under $D_{30}$**

| XPath expression |
|---|
| //table-wrap-group/table-wrap/speech/p/*citation*/conf-name |
| //table-wrap-group/table-wrap/speech/p/*citation*/conf-loc |
| //table-wrap-group/table-wrap/speech/p/*citation*/comment |
| //table-wrap-group/table-wrap/speech/p/*citation*/article-title |
| //table-wrap-group/table-wrap/speech/p/*citation*/annotation |
| //list-item/p/*contract-num*/named-content/ack/p |
| //p/*contract-num*/named-content/ack/boxed-text/sec |
| //p/*contract-sponsor*/named-content/verse-group/verse-group/verse-group |
| //statement/p/*contract-sponsor*/named-content/verse-group/verse-group |
| //statement/p/*contract-sponsor*/named-content/verse-group/verse-line |

**Table 5: XPath expressions with predicate under $D_{23}$ and $D_{30}$**

| | |
|---|---|
| XPath expression $p_5$: | //list-item/p/contract-num/named-content/ack[*chem-struct*]/p |
| Transformed result: | //list-item/p/contract-num/named-content/ack/p |
| XPath expression $p_6$: | //license/p/preformat/named-content/supplementary-material[*chem-struct-wrapper*]/disp-quote |
| Transformed result: | //license/p/preformat/named-content/supplementary-material[*chem-struct-wrap*]/disp-quote |
| XPath expression $p_7$: | /article/front/journal-meta/*custom-meta-wrap*[custom-meta/meta-name]/custom-meta/meta-value |
| Transformed result: | /article/front/journal-meta/*custom-meta-group*[custom-meta/meta-name]/custom-meta/meta-value |