Simplified Access Control Policies for XML Databases

Loreto Bravo and Ricardo Segovia
Universidad de Concepcin, Chile
{lbravo,risegovia}@udec.cl

Abstract. When defining Access Control Policies for XML Databases administrators need to make sure that they are not inconsistent, this is, that it is not possible to perform a forbidden operation through a sequence of allowed operations. This problem has been studied before for policies defined using authorizations based in insert, delete, replace and replaceVal types to control updates in documents that conform to structured DTDs and chain DTDs. For those policies, consistency can be checked in polynomial time, but the problem of minimally restoring consistency is NP-hard. In this article we show how the administration of authorization can be simplified by considering only insert and delete permissions, while still being able to control access of replace updates, in such a way that they can be checked for consistency and repaired if they are not in polynomial time. Also, this simplified policies allow to control a more general class of updates than the ones previously studied.

1 Introduction

XML documents are increasingly being used not only to share data but also to store it in that format, i.e. XML Databases. For this reason, it has become important to provide ways to control the way in which different users read and modify this data. There's been work on access control techniques for both read [1, 5–8, 10] and write [2–4] queries over XML data.

For write queries, a special type of vulnerability has been studied: the possibility that a policy specification enables a user to simulate a forbidden update through a sequence of allowed updates in the same policy. These "inconsistent" policies open up security problems and as such, need to be properly detected and repaired before being enforced.

Example 1. The US National Library of Medicine developed *Journal Publishing Tags* to describe the content and metadata of journal articles. A fragment of its schema is shown in Fig. 1[1] and a sample document in Fig. 2. □

[1] Each element is defined using regular expressions. We assume that all the elements that are not defined correspond to text(#PCDATA).

article → (front + front-stub), body?, (sub-article + response)*
front → journal-meta, article-meta, notes?
front-stub → journal-meta?, article-meta?, notes?
body → (address + fig + p + list)*, sec*
sub-article → (front + front-stub), body?
response → (front + front-stub), body?
journal-meta → journal-id+, issn+, isbn*, publisher?, notes?
article-meta → article-id*,author+,pub-date+, volume?, issue?
sec → title?, (fig + p + list + tex-math)*, sub-sec*
fig → object-id*, label?, caption?, graphic, (attrib + permissions)*
list → label?, title?, list-item+
sub-sec → title?, (address + fig + p + list + tex-math)*

Fig. 1. Journal Publishing Schema

```
<article>
 <front>
  <journal-meta>
   <journal-id>BMJ</journal-id>
   <issn>0959-8138</issn>
   <publisher>BMJ</publisher>
  </journal-meta>
  <article-meta>
   <article-id>jBMJ</article-id>
   <article-id>1508</article-id>
   <author>Freeman,George</author>
   <pub-date>13-4-2002</pub-date>
   <volume>324</volume>
   <issue>7342</issue>
  </article-meta>
 </front>
 <sub-article>
  <front-stub>
   <journal-meta>
    <journal-id>SMN</journal-id>
    <issn>7225-4139</issn>
    <publisher>SMN</publisher>
   </journal-meta>
  </front-stub>
 </sub-article>
</article>
```

Fig. 2. Journal Publishing XML Document D_0

Consider, for example, a policy that allows a user to insert and remove elements of type sub-article but does not allow him/her to insert new journal-id elements. This policy has an inconsistency, since it is possible to add a journal-id by first removing the element sub-article, and then inserting a new one that includes the

extra journal-id. For example, it shouldn't be possible to obtain document D_2 (in Figure 5) from document D_0 (in Figure 2) since inserting journal-id elements is forbidden. However, this can be achieved by first removing from D_0 the sub-article element, which results in document D_1 in Figure 3, and then inserting the new sub-article element shown in Figure 4 below the front element of document D_1. Thus, through two allowed updates such as deleting an inserting a sub-article element, it is possible obtain a document D_2 that is also the result of applying a forbidden update.

```
<article>
 <front>
  <journal-meta>
   <journal-id>BMJ</journal-id>
   <issn>0959-8138</issn>
   <publisher>BMJ</publisher>
  </journal-meta>
  <article-meta>
   <article-id>jBMJ</article-id>
   <article-id>1508</article-id>
   <author>Freeman,George</author>
   <pub-date>13-4-2002</pub-date>
   <volume>324</volume>
   <issue>7342</issue>
  </article-meta>
 </front>
```

Fig. 3. Document D_1 obtained from D_0 by deleting the sub-article element

```
 <sub-article>
   <front-stub>
    <journal-meta>
     <journal-id>SMN</journal-id>
     <journal-id>DKE</journal-id>
     <issn>7225-4139</issn>
     <publisher>SMN</publisher>
    </journal-meta>
   </front-stub>
  </sub-article>
```

Fig. 4. Tree with root sub-article to insert to document D_1

```
<article>
 <front>
  <journal-meta>
   <journal-id>BMJ</journal-id>
   <issn>0959-8138</issn>
   <publisher>BMJ</publisher>
  </journal-meta>
  <article-meta>
   <article-id>jBMJ</article-id>
   <article-id>1508</article-id>
   <author>Freeman,George</author>
   <pub-date>13-4-2002</pub-date>
   <volume>324</volume>
   <issue>7342</issue>
  </article-meta>
 </front>
 <sub-article>
  <front-stub>
   <journal-meta>
    <journal-id>SMN</journal-id>
    <journal-id>DKE</journal-id>
    <issn>7225-4139</issn>
    <publisher>SMN</publisher>
   </journal-meta>
  </front-stub>
 </sub-article>
```

Fig. 5. Document D_2 obtained from D_0 after inserting a new journal-ID element.

In [2] the authors defined inconsistencies in XML access control policies for write operations, studied the problem of identifying such policies for structured DTDs, showed that finding repairs to those policies is NP-complete and provided approximation algorithms to compute them. Later, in [4], authors extended those results to deal with a general kind of DTD called Chain Extended DTDs (CEDTDs for

short), like the one in Fig. 1, imposing some restriction over the type of updates that were considered. For instance, it only considered updates that replaced element types belonging to a disjunction without $+$, $*$ or ?. Thus, in the Journal Publishing example, the only valid replace updates would be between element types front and front-stub. In this setting, the problem of repairing inconsistent policies was shown to be NP-complete.

In this article, we show how an access control policy can be defined for XML databases that conform to CEDTDs in such a way that: (i) it deals with insert, delete and replace updates between any two distinct element types, (ii) policy consistency can be checked in polynomial time, (iii) repairs to inconsistent policies can be found in polynomial time. The cost to achieve this is that it cannot express some policies that can be defined in [4]. However, the type of policies that cannot be expressed are not common in practice.

This document is structured as follows. The definitions of XML documents, schemas and updates are provided in Section 2. Section 3 defines access control policies for CEDTDs. Next, in Section 4 the consistency of those policies in analyzed and for the cases in which they are inconsistent, the repair problem is studied in Section 5. Finally, the conclusions are provided in Section 6.

2 Preliminaries

An *extended DTD (EDTD)* is an abstraction of the two main schema definition languages for XML [9], namely DTDs and XSDs, and represented by a quintuple (Ele, $Types$, Rg, rt, μ) where *i)* Ele is a finite set of *element names ii)* $Types$ is a finite set of *element types iii)* rt is a distinguished element name in Ele and in $Types$ called the *root iv)* μ is a mapping from $Types$ to Ele such that $\mu(rt) = rt$, and *v)* Rg defines the element types: that is, $Rg : Types \rightarrow Reg_{Types}$, where Reg_{Types} is a set of regular expressions r over $Types$, considering concatenation, disjunction, element repetition and optionality, empty and string elements. Given a type A we will refer to $Rg(A)$ as its *production rule*. An element type B_i that appears in the production rule of an element type A is called a *subelement* type of A. Given an EDTD D, we write $A \leq_D B$ for the transitive, reflexive closure of the subelement relation.

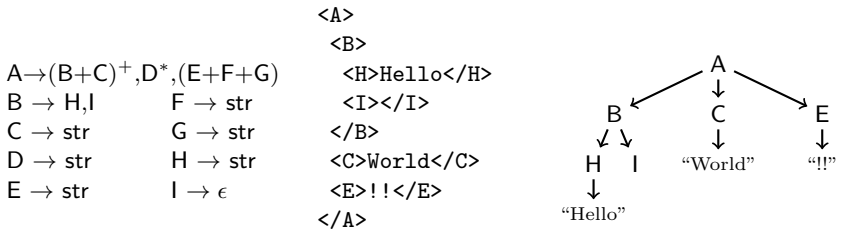
A *structured EDTD* is an EDTD such that its regular expressions are defined using the grammar: $\mathsf{str} \mid \epsilon \mid \mathsf{B}_1, \ldots, \mathsf{B}_n \mid \mathsf{B}_1 + \cdots + \mathsf{B}_n \mid \mathsf{B}^*$, where ",", "+" and "$*$" stand for *concatenation*, *disjunction* and *Kleene star* respectively, ϵ for the EMPTY element content, str for text values and $B \in Types$.

A *Chain EDTDs* consists of an EDTD whose regular expressions are: (i) a sequence of terms $(\mathsf{A}_1 + \cdots + \mathsf{A}_n)^q$, where q is an optional qualifier of the form $+, *$, or ?; (ii) a string str; or (iii) empty ϵ [11]. Every structured EDTD is a Chain EDTD.

Example 2. The Journal Publishing Schema in Fig. 1 is a chain EDTD but it is not a structured EDTD. For example, element article contains either a front or front-stub element, optionally a body element and zero or more repetitions of sub-articles or response elements. □

We will model an XML document as *rooted unordered* trees which conforms to a schema if it validates against its regular expressions. The XML document in Fig. 1 conforms to the chain EDTD in Fig. 2.

Example 3. Let t_0 be the XML document shown in Figure 6(b), also represented by the tree in Figure 6(c). Let D_0 be the chain EDTD rooted in A shown in Figure 6(a). Document t_0 conforms to schema D_0. □



(a) CEDTD D_0 (b) Document t_0 (c) Tree representation of t_0

Fig. 6. CEDTD D_0 with a document t_0 that conforms to it

We consider the following updates: $op ::= \mathsf{delete}(n) \mid \mathsf{insert}(n,t) \mid \mathsf{replace}(n,t) \mid \mathsf{replaceVal}(n,s)$. A $\mathsf{delete}(n)$ will remove node n and all its descendants. An $\mathsf{insert}(n,t)$ operation will add a tree t as a child node below n. A $\mathsf{replace}(n,t)$ operation will replace the subtree with root n by the tree t. A $\mathsf{replaceVal}(n,s)$ operation will replace the text value of node n with string s. An update operation is said to be *valid* for t and D if the result of applying it over t results in a document that conforms to D. For example, the update $\mathsf{delete}(E)$ would not be valid over $t0$, because the resulting document would not conform to D. In what follows we consider only valid updates. We denote by $[\![op]\!](t, D)$ the resulting XML document obtained by applying update operation op on tree t that conforms to schema D.

3 Access Control Policies

We use the notion of *update access type* to specify the access authorizations.

Definition 1. Given a CEDTD D, a *base update access type* (base UAT) is an expression of the form (A, ut) where $ut ::= \mathsf{insert}(\mathsf{B}) \mid \mathsf{delete}(\mathsf{B}) \mid \mathsf{replaceVal}$ and A and B are types from D. □

Intuitively, each of these *UATs* controls the operations that insert or delete an element of type B or operations that replace a string. The relation *uat* valid_in D, which indicates that a base update access type *uat* is valid for the EDTD D, is constrained by the following inference rules:

$$\frac{}{\mathsf{replaceVal\ valid\_in\ str}}$$

$$\frac{U\ \mathsf{valid\_in}\ t}{U\ \mathsf{valid\_in}\ t^*} \qquad \frac{U\ \mathsf{valid\_in}\ t}{U\ \mathsf{valid\_in}\ t^+} \qquad \frac{U\ \mathsf{valid\_in}\ t}{U\ \mathsf{valid\_in}\ t?}$$

$$\frac{}{\mathsf{insert}(\mathsf{B}_i)\ \mathsf{valid\_in}\ (\mathsf{B}_1 + \cdots + \mathsf{B}_n)} \qquad \frac{}{\mathsf{delete}(\mathsf{B}_i)\ \mathsf{valid\_in}\ (\mathsf{B}_1 + \cdots + \mathsf{B}_n)}$$

$$\frac{U\ \mathsf{valid\_in}\ f_i}{U\ \mathsf{valid\_in}\ f_1, \ldots, f_n} \qquad \frac{U\ \mathsf{valid\_in}\ Rg(\mathsf{A})}{(\mathsf{A}, U)\ \mathsf{valid\_in}\ D}$$

The set of valid base *UATs* for a given EDTD D is denoted by $\mathsf{valid}(D) = \{uat \mid uat\ \mathsf{valid\_in}\ D\}$.

Example 4. (example 3 cont.) Given schema D_0, $\mathsf{valid}(D_0) = \{(\mathsf{A}, \mathsf{insert}(\mathsf{B})), (\mathsf{A}, \mathsf{delete}(\mathsf{B})), (\mathsf{A}, \mathsf{insert}(\mathsf{C})), (\mathsf{A}, \mathsf{delete}(\mathsf{C})), (\mathsf{A}, \mathsf{insert}(\mathsf{D})), (\mathsf{A}, \mathsf{delete}(\mathsf{D})), (\mathsf{A}, \mathsf{insert}(\mathsf{E})), (\mathsf{A}, \mathsf{delete}(\mathsf{E})), (\mathsf{A}, \mathsf{insert}(\mathsf{F})), (\mathsf{A}, \mathsf{delete}(\mathsf{F})), (\mathsf{A}, \mathsf{insert}(\mathsf{G})), (\mathsf{A}, \mathsf{delete}(\mathsf{G})), (\mathsf{C}, \mathsf{replaceVal}), (\mathsf{D}, \mathsf{replaceVal}), (\mathsf{E}, \mathsf{replaceVal}), (\mathsf{F}, \mathsf{replaceVal}), (\mathsf{G}, \mathsf{replaceVal}), (\mathsf{H}, \mathsf{replaceVal})\}$ □

Definition 2. A policy P defined over an EDTD D, is represented by the set of allowed update access types defined over D such that $P \subseteq \mathsf{valid}(D)$. The set of forbidden *UAT* of a policy P defined over D are $\mathcal{F}(P, D) = \mathsf{valid}(D) \setminus P$. □

Example 5. (example 4 cont.) A policy for D_0 is P_{D_0}=$\{(\mathsf{A},\mathsf{insert}(\mathsf{B})),(\mathsf{A},\mathsf{delete}(\mathsf{B})), (\mathsf{A}, \mathsf{insert}(\mathsf{C})), (\mathsf{A}, \mathsf{delete}(\mathsf{C})), (\mathsf{A}, \mathsf{insert}(\mathsf{E})), (\mathsf{A}, \mathsf{delete}(\mathsf{E})),(\mathsf{A}, \mathsf{insert}(\mathsf{F})),(\mathsf{A}, \mathsf{delete}(\mathsf{F})), (\mathsf{A}, \mathsf{insert}(\mathsf{G})), (\mathsf{A}, \mathsf{delete}(\mathsf{G})), (\mathsf{C}, \mathsf{replaceVal}), (\mathsf{E}, \mathsf{replaceVal}), (\mathsf{G}, \mathsf{replaceVal})\}$. The set of UATs that are forbidden by P_{D_0} are $\mathcal{F}(P_{D_0}, D_0) = (\mathsf{A}, \mathsf{delete}(\mathsf{D})), (\mathsf{A}, \mathsf{insert}(\mathsf{D})), (\mathsf{D}, \mathsf{replaceVal}), (\mathsf{F}, \mathsf{replaceVal}), (\mathsf{H}, \mathsf{replaceVal})\}$. □

We want to enforce restriction over the applications of insert, delete and replace update operations, but the policies, as we have defined so far, allow or forbid only insertions and deletions. Thus, we need to deduce from our policies update access types to control replacements.

Definition 3. Given a CEDTD D, an *update access type* (*UAT*) defined over D is a base UAT or an expression of the form $(\mathsf{A}, \mathsf{replace}(\mathsf{B}, \mathsf{C}))$ and A, B, C are types from D. □

In order to enforce a policy P on updates, an expanded version $P^{\uparrow}$ is inferred from P, adding certain replace operations that are allowed. For instance, $\mathsf{replace}(\mathsf{B}_i, \mathsf{B}_j)$ is considered an *inferred update access type*, for which the permissions are obtained from the allowed and forbidden base *UATs* such as insert, delete and replaceVal.

Since we can apply over a document only updates that result in a new document that conforms to the DTD, we need to define replace UATs only in the cases where the replace could take place.

Example 6. For the production rule $\mathsf{A} \rightarrow (\mathsf{B}+\mathsf{C})^+,\mathsf{D}^*,(\mathsf{E}+\mathsf{F}+\mathsf{G})$ of the ongoing example, elements E, F and G can only be replaced between them, since a replacement of an element E by an B, for example, would result in a document that does not conform to the DTD. On the other hand, elements B, C and D can only be replaced between them. □

As the previous example shows, factor of the form $(\mathsf{B}_1 + \cdots + \mathsf{B}_n)$ impose restrictions over the type of elements that can be replaced.

Definition 4. An *XOR factor in A* is a factor of the form $(\mathsf{B}_1 + \cdots + \mathsf{B}_n)$ in the production rule of A. Let $XOR(\mathsf{A}) = \{\{\mathsf{B}_1, \cdots, \mathsf{B}_n\} | (\mathsf{B}_1 + \cdots + \mathsf{B}_n)$ is an XOR factor in $\mathsf{A}\}$. Types B_i and B_j are *alternates in* A if both B_i and B_j belong to the same XOR factor in A. A types B_i is *independent in* A if it does not belong to any XOR factor in A. □

In the policies defined in [4] updates that replaced an independent element by another were not considered.

Example 7. The production rule $\mathsf{A} \rightarrow (\mathsf{B}+\mathsf{C})^+,\mathsf{D}^*,(\mathsf{E}+\mathsf{F}+\mathsf{G})$ has one XOR factor, namely (E+F+G). The alternates in A are {E,F}, {E, G} and {F, G}. The independent types in A are B, C and D. □

Definition 5. Given a set of *UATs* U, the *expanded version* $U^\uparrow$ is obtained from U using the following inference rules:

$$\frac{(\mathsf{A}, \mathsf{delete}(\mathsf{B}_i)) \in \mathcal{U}, (\mathsf{A}, \mathsf{insert}(\mathsf{B}_j)) \in \mathcal{U}, i \neq j, \text{ and } \mathsf{B}_i, \mathsf{B}_j \text{ are independent in } \mathsf{A}}{(\mathsf{A}, \mathsf{replace}(\mathsf{B}_i, \mathsf{B}_j)) \in \mathcal{U}^\uparrow}$$

$$\frac{(\mathsf{A}, \mathsf{delete}(\mathsf{B}_i)) \in \mathcal{U}, (\mathsf{A}, \mathsf{insert}(\mathsf{B}_j)) \in \mathcal{U}, i \neq j, \text{ and } \mathsf{B}_i, \mathsf{B}_j \text{ are alternates in } \mathsf{A}}{(\mathsf{A}, \mathsf{replace}(\mathsf{B}_i, \mathsf{B}_j)) \in \mathcal{U}^\uparrow}$$

$$\frac{(\mathsf{A}, \mathsf{insert}(\mathsf{B})) \in \mathcal{U}, \text{ and } \mathsf{B} \text{ is independent in } \mathsf{A}}{(\mathsf{A}, \mathsf{insert}(\mathsf{B})) \in \mathcal{U}^\uparrow}$$

$$\frac{(\mathsf{A}, \mathsf{delete}(\mathsf{B})) \in \mathcal{U}, \text{ and } \mathsf{B} \text{ is independent in } \mathsf{A}}{(\mathsf{A}, \mathsf{delete}(\mathsf{B})) \in \mathcal{U}^\uparrow} \qquad \frac{(\mathsf{A}, \mathsf{replaceVal}) \in \mathcal{U}}{(\mathsf{A}, \mathsf{replaceVal}) \in \mathcal{U}^\uparrow}$$

Given a schema D and a policy P, the *set of extended valid* UATs corresponds to $\mathsf{valid}(D)^\uparrow$ and the *expanded policy* to $P^\uparrow$. □

This expanded policy can be used to check if an update is allowed or forbidden by a policy.

Example 8. (example 4 cont.) The set of extended valid *UATs* $\mathsf{valid}(\mathcal{D})^{\uparrow}$ = {(A,insert(B)), (A,insert(C)), (A,insert(D)), (A,delete(B)), (A,delete(C)), (A,delete(D)), (A, replace(E, F)),(A,replace(E, G)), (A, replace(F, E)), (A,replace(F, G)),(A,replace(G, E)), (A,replace(G, F)),(A,replace(B,C)),(A,replace(C,B)),(A,replace(C,D)),(A,replace(D,C)), (A, replace(B,D)), (A, replace(D, B)), (C, replaceVal), (D, replaceVal), (E, replaceVal), (F, replaceVal), (G, replaceVal), (H, replaceVal)}. It differs from $\mathsf{valid}(\mathcal{D})$ since it adds all the replace *UATs* and removes some insert and delete. The set of extended valid *UATs* represent the types of updates that could result in a document that conforms to schema D_0. The extended policy $P^{\uparrow}_{D_0}$ = {(A, insert(B)), (A, insert(C)), (A, delete(C)), (A, replace(C, B)), (A, delete(B)), (A, replace(B, C)), (A,replace(E,F)), (A,replace(F,E)),(A,replace(G,E)), (A,replace(E,G)), (A,replace(F,G)), (A,replace(G,F)),(C, replaceVal),(E,replaceVal),(G,replaceVal)}. The *UATs* that are forbidden by the policy correspond to $\mathsf{valid}(\mathcal{D})^{\uparrow} \setminus P^{\uparrow}_{\mathcal{D}}$. □

Intuitively, an *UAT* in $P^{\uparrow}$ represents a set of atomic update operations. More specifically, for t an instance of CEDTD D, *op* an atomic update and *uat* an update access type we say that *op matches uat* on t (*op* $\mathsf{matches}_t$ *uat*) if one of the following inference rules applies:

$$\frac{\eta(n) = \mathsf{A} \quad t' \in I_D(\mathsf{B})}{\mathsf{insert}(n, t')\ \mathsf{matches}_t\ (\mathsf{A}, \mathsf{insert}(\mathsf{B}))} \qquad \frac{\eta(n) = \mathsf{B} \quad \eta(\mathsf{parent}_t(n)) = \mathsf{A}}{\mathsf{delete}(n)\ \mathsf{matches}_t\ (\mathsf{A}, \mathsf{delete}(\mathsf{B}))}$$

$$\frac{\eta(n) = \mathsf{B} \quad t' \in I_D(\mathsf{B}') \quad \eta(\mathsf{parent}_t(n)) = \mathsf{A}}{\mathsf{replace}(n, t')\ \mathsf{matches}_t\ (\mathsf{A}, \mathsf{replace}(\mathsf{B}, \mathsf{B}'))} \qquad \frac{\eta(n) = \mathsf{str} \quad \eta(\mathsf{parent}_t(n)) = \mathsf{A}}{\mathsf{replaceVal}(n, s)\ \mathsf{matches}_t(\mathsf{A}, \mathsf{replaceVal})}$$

Function η corresponds to the unique mapping from each node in t to the element type associated to it. This uniqueness follows from the assumption that D is unambiguous.

As we said before, we will restrict only to valid update operation which given a document t and a CEDTD D the result of applying it over t results in a document that conforms with D. Since the problem of validity and authorization are orthogonal, we assume that all the operations to checked by the policy are valid.

The set of operations that are allowed by a policy P on an XML tree t in an CEDTD D, denoted by $[\![P]\!](t, D)$, is the union of the atomic valid update operations matching each *UAT* in $P^{\uparrow}$. More formally, $[\![P]\!](t, D) = \{op \mid op\ \mathsf{matches}_t\ uat$ and $uat \in P^{\uparrow}\}$. Analogously, the forbidden operations are $[\![\mathcal{F}]\!](t, D) = \{op \mid op\ \mathsf{matches}_t\ uat$, and $uat \in (\mathsf{valid}(D) \setminus P^{\uparrow})\}$. A safe run $t_0 \xrightarrow{op_1} t_1 \ldots \xrightarrow{op_n} t_n$ is *allowed w.r.t.* P and D if for every i we have $op_i \in [\![P]\!](t_{i-1}, D)$.

4 Consistency of Policies

A policy is said to be consistent if it is not possible to simulate a forbidden update through a sequence of allowed updates. More formally:

Definition 6. [2] Consider a policy P defined over schema D. An **inconsistency** in P for D consists of an allowed run $t_0 \xrightarrow{op_1} t_1 \ldots \xrightarrow{op_n} t_n$ and an update $op_0 \in [\![\mathcal{F}]\!](t_0, D)$ such that $t_n = [\![op_0]\!](t_0, D)$. A policy P is consistent for D if there are no inconsistencies. □

We say that *nothing is forbidden below an element type* A in a policy P defined over D if for every B_i such that $\mathsf{A} \leq_D \mathsf{B}_i$ and every $(\mathsf{B}_i, x) \in \mathsf{valid}(D)$, $(\mathsf{B}_i, x) \in P$. If A has any replace operations in $\mathsf{valid}(D)$, then we define the *replace graph* $\mathcal{G}_A^P = (\mathcal{V}_A, \mathcal{E}_A)$ where *i)* $\mathcal{V}_A$ is the set of subelements of A and *ii)* $(\mathsf{B}_i, \mathsf{B}_j) \in \mathcal{E}_A$ if there exists $(A, \mathsf{replace}(\mathsf{B}_i, \mathsf{B}_j)) \in P^{\uparrow}$.

Theorem 1. A policy P defined over a CEDTD D is consistent if and only if for every production rule:

1. If $(\mathsf{A},\mathsf{insert}(\mathsf{B})) \in P^{\uparrow}$ and $(\mathsf{A},\mathsf{delete}(\mathsf{B})) \in P^{\uparrow}$, then nothing is forbidden below B
2. If for every $i \in [1, \ldots n]$, if B_i is contained in a cycle in $\mathcal{G}_A^P$ then nothing is forbidden below B_i.

Proof Sketch: Policy P and schema D can be transformed into a new policy P_{struct} defined over a new structured schema D_{struct} in such a way that P is consistent over D if and only if P_{struct} is consistent over D_{struct}. In [2] the authors identify three cases in which a policy defined over a structured schema is consistent: insert/delete, negative-cycle and forbidden-transitivity. Because of the characteristics of P_{struct} only the first two can occur. By reversing the transformations, we get the conditions for CEDTDs that correspond to insert/delete and negative cycle inconsistencies. □

The conditions in Theorem 1 are defined over the expanded policy $P^{\uparrow}$. In order to simplify the consistency checking problem, it is better to see how these conditions can be checked in P instead of $P^{\uparrow}$.

Proposition 1. A policy P defined over a CEDTD D is consistent if and only if for every element types A, B and C in D:

1. If B is independent in A and $\{(\mathsf{A}, \mathsf{insert}(\mathsf{B})), (\mathsf{A}, \mathsf{delete}(\mathsf{B}))\} \subseteq P$ then nothing is forbidden below B.
2. If B and C are alternates in A and $\{(\mathsf{A}, \mathsf{insert}(\mathsf{B}))$, $(\mathsf{A}, \mathsf{delete}(\mathsf{B}))$, $(\mathsf{A}, \mathsf{insert}(\mathsf{C}))$, $(\mathsf{A}, \mathsf{delete}(\mathsf{C}))\} \subseteq P$ then nothing is forbidden below B and C. □

It is possible to check if conditions in Proposition 1 hold for a policy using standard polynomial-time graph algorithms over the DTD. Thus, the problem of deciding policy consistency with respect to CEDTDs is in PTIME.

Example 9. (example 8 cont) Policy P_{D_0} has several inconsistencies. First, by having both $(\mathsf{A}, \mathsf{insert}(\mathsf{B}))$ and $(\mathsf{A}, \mathsf{delete}(\mathsf{B}))$ allowed for an independent element

type B under A, but forbidding (H, replaceVal). Another one is given by having (A, insert(E)), (A, delete(E)), (A, insert(F)) and (A, delete(F)) as allowed operations under alternate types E and F, but forbidding (F, replaceVal). Finally, given by having (A, insert(F)), (A, delete(F)), (A, insert(G)) and (A, delete(G)) as allowed operations under alternate types F and G, but forbidding (F, replaceVal).

5 Repairing Inconsistent Policies

If a policy is inconsistent, we would like to suggest possible minimal ways of modifying it in order to restore consistency. In other words, we would like to find *repairs* that are as close as possible to the inconsistent policy. We would like this repairs to be more restrictive than the original, this is, a repair should provide less access to the user.

Definition 7. [2] A policy P' is a *repair* of a policy P defined over a DTD D if and only if: i) P' is defined over D, ii) P' is consistent, and iii) $P' \subseteq P$. Furthermore a repair P' of P is a *minimum-repair* if there is no repair P'' such that $|P'| < |P''|$.[2] □

In other words a repair is a new policy over the same schema, which is consistent and contains a subset of the allowed *UAT* in P. For this repair to be minimal, we require that it contains the maximum number of allowed *UATs* or, equivalently, that a minimal number of *UATs* has been removed from P.

The problem of computing a minimum-repair can be solved in PTIME. In what follows we describe an algorithm to do so. By Proposition 1 we have that we need to solve two types of inconsistencies:

Type 1: if B is independent in A, $\{(A, insert(B)), (A, delete(B))\} \subseteq P$ and something is forbidden below B.

Type 2: if B and C are alternates in A, $\{$(A,insert(B)),(A,delete(B)),(A,insert(C)), (A, delete(C))$\} \subseteq P$ and something is forbidden below B or below C.

Because of minimality of repairs, repairs of inconsistencies of Type 1 can be solved independently by removing either (A, insert(B)) or (A, delete(B)). This is not the case for inconsistencies of Type 2, since there might be more than one such inconsistency involving shared types.

Example 10. (example 9 continued) There is an inconsistency of Type 2 between E and F since (F, replaceVal) $\notin P$. To solve this inconsistency, we need to forbid either (A, insert(E)),(A, delete(E)),(A, insert(F)) or (A, delete(F)). However, choosing to forbid (A, insert(E)) or (A, delete(E)) will not result in a minimal repair since element F is involved in another inconsistency of Type 2 with G since (F, replaceVal) $\notin P$. Thus, when there are several inconsistencies of Type 2 it

[2] $|P'|$ denotes the cardinality of set P'

Algorithm 1: PolicyRepair

Input : CEDTD $D = (Ele, Types, Rg, rt, \mu)$, Policy P
Output : set *toRemove* of *UATs* to remove to restore consistency of P

```
1  (T1, T2) ← DetectInconsistencies(D, P)
2  toRemove ← ∅
3  for A ∈ Types do
4      for B ∈ T1(A) do
5          U ← either (A, insert(B)) or (A, delete(B)) (choose randomly)
6          toRemove ← toRemove ∪ {U}
7      for S ∈ T2(A) do
8          if ∇ ∈ S then
9              S ← S ∖ {∇}
10         else
11             Let E be a randomly chosen element in S
12             S ← S ∖ {E}
13         for C ∈ S do
14             U ← either (A, insert(B)) or (A, delete(B)) (choose randomly)
15             toRemove ← toRemove ∪ {U}
16             S ← S ∖ {C}
17 return toRemove
```

is necessary to choose carefully to ensure that a minimal repair is found. In this case, either $(\mathsf{A}, \mathsf{insert}(\mathsf{F}))$ or $(\mathsf{A}, \mathsf{delete}(\mathsf{F}))$ need to be forbidden. The policy also has a inconsistency of Type 1 that can be solved by forbidding either $(\mathsf{A}, \mathsf{insert}(\mathsf{B}))$ or $(\mathsf{A}, \mathsf{delete}(\mathsf{B}))$ from our policy. Thus, a minimum-repair for P is $P' = P \setminus \{(\mathsf{A}, \mathsf{insert}(\mathsf{B})), (\mathsf{A}, \mathsf{delete}(\mathsf{F}))\}$. □

Algorithm **PolicyRepair**, takes a policy P defined over a CEDTD D and returns a set of *UATs* to remove from P to restore consistency. The algorithm first calls Algorithm **DetectInconsistencies** that computes with the recursive Algorithm **NodeInconsistency** the inconsistencies of Type 1 and Type 2 and stores them in two functions T_1 and T_2 using depth-first search over schema D. Function $T_1 : Types \rightarrow 2^{Types}$, returns for every type A the set of *Types* B such that $\{(\mathsf{A}, \mathsf{insert}(\mathsf{B})), (\mathsf{A}, \mathsf{delete}(\mathsf{B}))\} \subseteq P$ and something is forbidden below B. These inconsistencies are solved by either deleting $(\mathsf{A}, \mathsf{insert}(\mathsf{B}))$ or $(\mathsf{A}, \mathsf{delete}(\mathsf{B}))$ (lines 4-6 of Algorithm **PolicyRepair**). Function $T_2 : Types \rightarrow 2^{2^{Types}}$, returns for every type A, a set of subsets of *Types* in such a way that if $S \in T_2(\mathsf{A})$, then every $\mathsf{B} \in S$ belongs to the same XOR factor below A and $\{(\mathsf{A}, \mathsf{insert}(\mathsf{B})), (\mathsf{A}, \mathsf{delete}(\mathsf{B}))\} \subseteq P$ and something is forbidden below B. Also, if $\nabla \in S$ it means there is a type C in the same XOR factor such that $\{(\mathsf{A}, \mathsf{insert}(\mathsf{C})), (\mathsf{A}, \mathsf{delete}(\mathsf{C}))\} \subseteq P$ but everything is allowed below C. It is easy to check that in order to solve the inconsistencies of Type 2, for every $S \in T_2(\mathsf{A})$ we need to delete either $(\mathsf{A}, \mathsf{insert}(\mathsf{B}))$ or $(\mathsf{A}, \mathsf{delete}(\mathsf{B}))$

Algorithm 2: DetectInconsistencies

Input : CEDTD $D = (Ele, Types, Rg, rt, \mu)$, Policy P
Output : A tuple (T_1, T_2) where T_1 and T_2 are functions that given a type they return the inconsistencies of Type 1 and Type 2 below it

```
1 for A ∈ Types do
2     visited(A) ← −1
3     if there exists (A, op) ∈ F(P, D) then
4         ν(A) = 0
5     else
6         ν(A) ← −1
7     T1(A) ← ∅
8     T2(A) ← ∅
9 NodeInconsistency(D, rt, visited, ν, T1, T2)
10 return (T1, T2)
```

for every $\mathsf{B} \in S$ except for ∇ if it belongs to F or any type if it does not (see lines 7-15 of Algorithm **PolicyRepair**).

Algorithm 3: NodeInconsistency

Input : CEDTD D, Node A, functions:*visited*, ν, T_1 and T_2
Result: $T_1(\mathsf{A})$ and $T_2(\mathsf{A})$ contain inconsistencies of Type 1 and Type 2 below A

```
1  if visited(A) < 0 then
2      for B independent in A do
3          nodeInconsistency(MG_D, visited, B)
4          if ν(B) = 0 then
5              ν(A) ← 0
6              if (A, insert(B)) ∈ P and (A, delete(B)) ∈ P then
7                  T_1(A) = T_1(A) ∪ {B}
8      for F ∈ XOR(A) do
9          S ← ∅
10         for B ∈ F do
11             nodeInconsistency(MG_D, visited, B)
12             if ν(B) = 0 then
13                 ν(A) ← 0
14             if (A, insert(B)) ∈ P and (A, delete(B)) ∈ P then
15                 if ν(B) = 0 then
16                     S = S ∪ {B}
17                 else
18                     S = S ∪ {∇}
19         if |S| > 1 then
20             T_2(A) ← T_2(A) ∪ {S}
21     if ν(A) = −1 then
22         ν(A) ← 1
23     visited(A) ← 1
```

Example 11. (example 5 continued) Algorithm **DetectInconsistencies** returns T_1 and T_2 such that:

$$T_1(\mathsf{E}) = \begin{cases} \{\mathsf{B}\} \text{ for } \mathsf{E}{=}\mathsf{A} \\ \emptyset \text{ for } \mathsf{E}{\neq}\mathsf{A} \end{cases} \qquad T_2(\mathsf{E}) = \begin{cases} \{\{\mathsf{F}, \nabla\}\} \text{ for } \mathsf{E}{=}\mathsf{A} \\ \emptyset \text{ for } \mathsf{E}{\neq}\mathsf{A} \end{cases}$$

To solve the only inconsistency of Type 1, it chooses from either $(\mathsf{A}, \mathsf{insert}(\mathsf{B}))$ or $(\mathsf{A}, \mathsf{delete}(\mathsf{B}))$ and adds it to the *toRemove* set. Next, for sets in $T_2(\mathsf{A})$, it loads them into a variable S and checks if $\nabla \in S$. If such element is in the set, it means that there are cycles inside the XOR factor involving elements that do not have something forbidden below them. In this case, the algorithm erases ∇ from S. Then the algorithm proposes to add either $(\mathsf{A}, \mathsf{insert}(\mathsf{F}))$ or $(\mathsf{A}, \mathsf{delete}(\mathsf{F}))$ to the *toRemove* set. So, by choosing one *UAT* from each combination shown, and changing them from allowed to forbidden, consistency is restored. □

Even though finding a single minimum-repair can be done in polynomial time, there might an exponential number of them.

Proposition 2. Given a policy P over a CEDTD D, a minimum-repair of it can be computed with Algorithm 1 in $O(n+m)$ where n is the number of types in D and m is the number edges in the tree that represents D. All the repairs can be computed in $O(n \times 2^n)$. □

In most of the cases m is $O(n)$ and thus a minimum repair can be found in linear time on the number of types in the DTD. In the worst case m is $O(n^2)$. Even thought computing all the repairs requires exponential time, it is also possible to show in linear time all the inconsistencies to the designer of the policy so that he can choose how to solve each of them.

6 Conclusions

We have provided an access control policy definition language based in basic update access types, that simplifies the administration of authorization, since it only requires that the policy manager defines whether the user should be able to insert or delete elements of the DTD, and automatically deduct replace authorizations from such permissions. For these type of policies, consistency checking and repair computation problems can be solved in polynomial time. We also provided efficient algorithms that, given an inconsistent policy, computes a consistent one that is as close as possible to the inconsistent one in terms of forbidden *UATs*.

These policies are an improvement over the ones defined in [4], since the repair computation problem for those is NP-hard. This is achieved by instead of defining policies in terms of insert, delete and replace *UATs*, we here define policies using only insert and delete *UATs* and deducing from them the replace permissions. In terms of expressive power they are not comparable since we can define policies that they cannot and vice-versa. For example, consider a DTD with the production rule $\mathsf{A} \rightarrow \mathsf{B}^*, \mathsf{C}^*$ and B and C containing text. A policy $P = \{\mathsf{insert}(C), \mathsf{delete}(B)\}$ with $P^{\uparrow} = \{\mathsf{insert}(C), \mathsf{delete}(B), \mathsf{replace}(B, C)\}$ cannot be expressed in policies defined in [4] since they do not consider replace between elements that do not belong to an XOR factor. On the other hand, we cannot write a policy P for which $P^{\uparrow} = \{(A, \mathsf{replace}(B, C)), (B, \mathsf{replaceVal}), (C, \mathsf{replaceVal})\}$ even though $P^{\uparrow}$ is a consistent policy in [4]. We believe that the gain in expressive power outweighs the loss since being able to provide permissions for replacements outside of XOR factors is a useful feature and the policies that cannot be expressed do not seem useful in practice. Thus, the policies here introduced can be used as a simple, but powerful, way to provide consistent access control to XML databases.

Acknowledgements: This work is funded by FONDECYT grant 11080260 and CONICYT grant PSD-57. We would like to thank Irini Fundulaki and James Cheney for useful discussions.

References

1. Bertino, E., Ferrari, E.: Secure and Selective Dissemination of XML Documents. ACM TISSEC 5(3), 290–331 (2002)
2. Bravo, L., Cheney, J., Fundulaki, I.: Repairing Inconsistent XML Write-Access Control Policies. In: DBPL. pp. 97–111 (2007)
3. Bravo, L., Cheney, J., Fundulaki, I.: ACCOn: Checking Consistency of XML Write-Access Control Policies. In: EDBT'08 (Demo) (2008)
4. Bravo, L., Cheney, J., Fundulaki, I., Segovia, R.: Consistency and Repair for XML Write-Access Control Policies, VLDB Journal. Accepted (2012)
5. Damiani, E., di Vimercati, S.D.C., Paraboschi, S., Samarati, P.: A Fine-grained Access Control System for XML Documents. ACM TISSEC 5(2), 169–202 (2002)
6. Fan, W., Chan, C.Y., Garofalakis, M.: Secure XML Querying with Security Views. In: ACM SIGMOD. pp. 587–598 (2004)
7. Fundulaki, I., Marx, M.: Specifying Access Control Policies for XML Documents with XPath. In: SACMAT. pp. 61–69 (2004)
8. Kuper, G., Massacci, F., Rassadko, N.: Generalized XML Security Views. In: SACMAT. pp. 77–84 (2005)
9. Martens, W., Neven, F., Schwentick, T., Bex, G.J.: Expressiveness and Complexity of XML Schema. ACM Trans. Database Syst. 31(3), 770–813 (2006)
10. Murata, M., Tozawa, A., Kudo, M., Hada, S.: XML Access Control Using Static Analysis. ACM TISSEC 9(3), 290–331 (2006)
11. Van den Bussche, J., Neven, F., Bex, G.J.: DTDs versus XML Schema: A Practical Study. In: WebDB'04. Paris, France (2004)