Repairing Inconsistent XML Write-Access Control Policies

Loreto Bravo, James Cheney, and Irini Fundulaki

School of Informatics, University of Edinburgh, UK
{lbravo,jcheney,efountou}@inf.ed.ac.uk

Abstract. XML access control policies involving updates may contain security flaws, here called *inconsistencies*, in which a forbidden operation may be simulated by performing a sequence of allowed operations. This paper investigates the problem of deciding whether a policy is consistent, and if not, how its inconsistencies can be repaired. We consider policies expressed in terms of annotated DTDs defining which operations are allowed or denied for the XML trees that are instances of the DTD. We show that consistency is decidable in PTIME for such policies and that consistent partial policies can be extended to unique "least-privilege" consistent total policies. We also consider repair problems based on deleting privileges to restore consistency, show that finding minimal repairs is NP-complete, and give heuristics for finding repairs.

1 Introduction

Discretionary access control policies for database systems can be specified in a number of different ways, for example by storing access control lists as annotations on the data itself (as in most file systems), or using rules which can be applied to decide whether to grant access to protected resources. In relational databases, high-level policies that employ rules, roles, and other abstractions tend to be much easier to understand and maintain than access control list-based policies; also, they can be implemented efficiently using static techniques, and can be analyzed off-line for security vulnerabilities [7].

Rule-based, fine-grained access control techniques for XML data have been considered extensively for *read-only queries* [11,15,14,2,17,10]. However, the problem of controlling *write access* is relatively new and has not received much attention. Authors in [2,10,16] studied enforcement of write-access control policies following annotation-based approaches.

In this paper, we build upon the schema-based access control model introduced by Stoica and Farkas [19], refined by Fan, Chan, and Garofalakis [11], and extended to write-access control by Fundulaki and Maneth [13]. We investigate the problem of checking for, and repairing, a particular class of vulnerabilities in XML write-access control policies. An access control policy specifies which actions to allow a user to perform based on the syntax of the atomic update, not its actual behavior. Thus, it is possible that a single-step action which is explicitly forbidden by the policy can nevertheless be simulated by one or more allowed actions. This is what we mean by an *inconsistency*; a consistent policy is one in which such inconsistencies are not possible. We believe inconsistencies are an interesting class of policy-level security vulnerabilities since such policies allow users to circumvent the intended effect of the policy. The

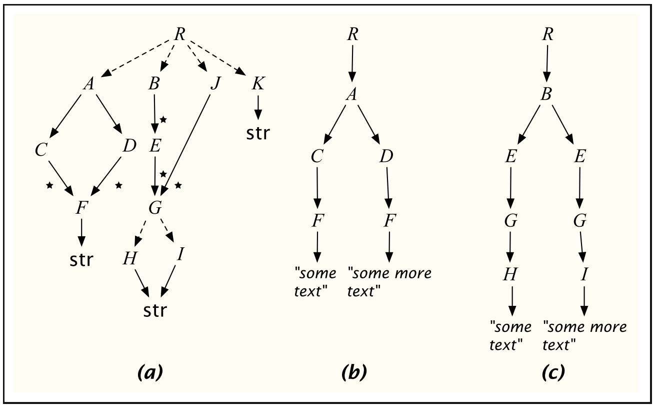

Fig. 1. DTD graph (a) and XML documents conforming to the DTD (b, c)

purpose of this paper is to define consistency, understand how to determine whether a policy is consistent, and show how to automatically identify possible repairs for inconsistent policies.

Motivating Example. We introduce here an example and refer to it throughout the paper. Consider the XML DTD represented as a graph in Fig. 1(a). A document conforming to this DTD has as root an R-element with a single child element that can either be an A, B, J or K-element (indicated with dashed edges); similarly for G. An A-element has one C and one D children elements. A B-element can have zero or more E children elements (indicated with $*$-labeled edges); similarly, E and J elements can have zero or more G children elements. Finally, F, H, I and K are text elements. Fig. 1(b) and (c) show two documents that conform to the DTD.

Suppose that a security policy *allows* one to *insert* and *delete* G elements and *forbids* one from replacing an H with an I element. It is straightforward to see that the forbidden operation can be simulated by first deleting the G element with an H child and then inserting a G element with an I child. There are different ways of fixing this inconsistency: either *(a)* to allow all operations below element G or *(b)* forbid one of the *insert* and *delete* operations at node G.

Now, suppose that the policy *allows* one to *replace* an A-element with a B-element and this with a J-element, but *forbids* the replacement of A with J elements. The latter operation can be easily simulated by performing a sequence of the allowed operations. As in the previous case, the repairs that one can propose are *(a)* to allow the forbidden replace operation or *(b)* forbid one of the allowed replace operations.

Our contributions. In this paper we consider policies that are defined in terms of *non-recursive structured* XML DTDs as introduced in [11] that capture without loss of generality more general non-recursive DTDs. We first consider *total* policies in which all allowed or forbidden privileges are explicitly specified. We define consistency for such policies and prove the correctness of a straightforward polynomial time algorithm for consistency checking. We also consider *partial* policies in which privileges may be omitted. Such a policy is consistent if it can be extended to a consistent total policy;

there may be many such extensions, but we identify a canonical *least-privilege* consistent extension, and show that this can be found in polynomial time (if it exists). Finally, given an inconsistent (partial or total) policy, we consider the problem of finding a "repair", or minimal changes to the policy which restore consistency. We consider repairs based on changing operations from allowed to forbidden, show that finding minimal repairs is NP-complete, and provide heuristic repair algorithms that run in polynomial time.

The rest of this paper is structured as follows: in Section 2 we provide the definitions for XML DTDs and trees. Section 3 discusses *i)* the atomic updates and *ii)* the access control policies that we are considering. Consistency is discussed in Section 4; Section 5 discusses algorithms for detecting and repairing inconsistent policies. We conclude in Section 6. Proofs of theorems and detailed algorithms can be found in the full version of the paper [4].

2 XML DTDs and Trees

We consider *structured* XML DTDs as discussed in [11]. Although not all DTDs are syntactically representable in this form, one can (as argued by [11]) represent more general DTDs by introducing new element types. The DTDs we consider here are 1-unambiguous as required by the XML standard [5].

Definition 1 (XML DTD). Let $\mathcal{L}$ be the infinite domain of labels. A DTD D is represented by (Ele, Rg, rt) where *i)* $Ele \subseteq \mathcal{L}$ is a finite set of *element types ii)* rt is a distinguished type in Ele called the *root type* and *iii)* Rg defines the element types: that is, for any $A \in Ele$, $Rg(A)$ is a regular expression of the form:

$$Rg(A) := \mathsf{str} \mid \epsilon \mid B_1, B_2, \ldots, B_n \mid B_1 + B_2 + \ldots + B_n \mid B_1*$$

where $B_i \in Ele$ are distinct, ",", "+" and "$*$" stand for *concatenation*, *disjunction* and *Kleene star* respectively, ϵ for the EMPTY element content and str for text values.

We will refer to $A \rightarrow Rg(A)$ as the *production rule* for A. An element type B_i that appears in the production rule of an element type A is called the *subelement* type of A. We write $A \leq_D B$ for the transitive, reflexive closure of the subelement relation.

A DTD can also be represented as a directed acyclic graph that we call *DTD graph*.

Definition 2 (DTD Graph). A DTD graph $G_D = (\mathcal{V}_D, \mathcal{E}_D, r_D)$ for a DTD $D = (Ele, Rg, rt)$ is a directed acyclic graph (DAG) where *i)* $\mathcal{V}_D$ is the set of nodes for the element types in $Ele \cup \{\mathsf{str}\}$, *ii)* $\mathcal{E}_D = \{(A, B) \mid A, B \in Ele$ and B is a subelement type of $A\}$ and *iii)* r_D is the distinguished node rt.

Example 1. The production rules for the DTD graph shown in Fig. 1 are:

$R \rightarrow A+B+J+K$	$D \rightarrow F*$	$G \rightarrow H + I$	$H \rightarrow \mathsf{str}$
$A \rightarrow C, D$	$B \rightarrow E*$	$J \rightarrow G*$	$I \rightarrow \mathsf{str}$
$C \rightarrow F*$	$E \rightarrow G*$	$F \rightarrow \mathsf{str}$	$K \rightarrow \mathsf{str}$ □

We model XML documents as *rooted unordered* trees with labels from $\mathcal{L} \cup \{\mathsf{str}\}$.

Definition 3 (XML Tree). An unordered XML tree t is an expression of the form $t = (N_t, E_t, \lambda_t, r_t, v_t)$ where *i)* N_t is the set of nodes *ii)* $E_t \subset N_t \times N_t$ is the set of edges, *iii)* $\lambda_t : N_t \rightarrow \mathcal{L} \cup \{\mathsf{str}\}$ is a labeling function over nodes *iv)* r_t is the root of t and is a distinguished node in N_t and *v)* v_t is a function that assigns a string value to nodes labeled with str.

We denote by $\mathsf{children}_t(n)$, $\mathsf{parent}_t(n)$ and $\mathsf{desc}_t(n)$, the children, parent and descendant nodes, respectively, of a node n in an XML tree t. The set $\mathsf{desc}^e_t(n)$ denotes the edges in E_t between descendant nodes of n. A node labeled with an element type A in DTD D is called an *instance* of A or an *A-element*.

We say that an XML tree $t = (N_t, E_t, \lambda_t, r_t, v_t)$ *conforms* to a DTD $D = (Ele, Rg, rt)$ at element type A if *i)* r_t is labeled with A (i.e., $\lambda_t(r_t) = A$) *ii)* each node in N_t is labeled with either an Ele element type B or with str, *iii)* each node in t labeled with an Ele element type B has a list of children nodes such that their labels are in the language defined by $Rg(B)$ and *iv)* each node in t labeled with str has a string value ($v_t(n)$ is defined) and is a leaf of the tree. An XML tree t is a valid instance of the DTD D if r_t is labeled with rt. We write $I_D(A)$ for the set of valid instances of D at element type A, and I_D for $I_D(rt)$.

Definition 4 (XML Tree Isomorphism). We say that an XML tree t_1 is isomorphic to an XML tree t_2, denoted $t_1 \equiv t_2$, iff there exists a bijection $h : N_{t_1} \rightarrow N_{t_2}$ where: *i)* $h(r_{t_1}) = r_{t_2}$ *ii)* if $(x, y) \in E_{t_1}$ then $(h(x), h(y)) \in E_{t_2}$, *iii)* $\lambda_{t_1}(x) = \lambda_{t_2}(h(x))$, and *iv)* $v_{t_1}(x) = v_{t_2}(h(x))$ for every x with $\lambda_{t_1}(x) = \mathsf{str} = \lambda_{t_2}(h(x))$.

3 XML Access Control Framework

3.1 Atomic Updates

Our updates are modeled on the XQuery Update Facility draft [8], which considers delete, replace and several insert update operations. A $\mathsf{delete}(n)$ operation will delete node n and all its descendants. A $\mathsf{replace}(n, t)$ operation will replace the subtree with root n by the tree t. A $\mathsf{replace}(n, s)$ operation will replace the text value of node n with string s. There are several types of insert operations, *e.g.,* $\mathsf{insert\ into}(n, t)$, $\mathsf{insert\ before}(n, t)$, $\mathsf{insert\ after}(n, t)$, $\mathsf{insert\ as\ first}(n, t)$, $\mathsf{insert\ as\ last}(n, t)$. Update $\mathsf{insert\ into}(n, t)$ inserts the root of t as a child of n whereas update $\mathsf{insert\ as\ first}(n, t)$ ($\mathsf{insert\ as\ last}(n, t)$) inserts the root of t as a first (resp. last) child of n. Update operations $\mathsf{insert\ before}(n, t)$ and $\mathsf{insert\ after}(n, t)$ insert the root node of t as a preceding and following sibling of n resp..

Since we only consider unordered XML trees, we deal only with the operation $\mathsf{insert\ into}(n, t)$ (for readability purposes, we are going to write $\mathsf{insert}(n, t)$). Thus, in what follows, we will restrict to four types of update operations: $\mathsf{delete}(n)$, $\mathsf{replace}(n, t)$, $\mathsf{replace}(n, s)$ and $\mathsf{insert}(n, t)$.

More formally, for a tree $t_1 = (N_{t_1}, E_{t_1}, \lambda_{t_1}, r_{t_1}, v_{t_1})$, a node n in t_1, a tree $t_2 = (N_{t_2}, E_{t_2}, \lambda_{t_2}, r_{t_2}, v_{t_2})$ and a string value s, the result of applying $\mathsf{insert}(n, t_2)$, $\mathsf{replace}(n, t_2)$, $\mathsf{delete}(n)$ and $\mathsf{replace}(n, s)$ to t_1, is a new tree $t = (N_t, E_t, \lambda_t, r_t, v_t)$ defined as shown in Table 1. We denote by $[\![op]\!](t)$ the result of applying update operation op on tree t.

Table 1. Semantics of update operations

	N_t	E_t	λ_t	r_t	v_t
$[\![\mathsf{insert}(n,t_2)]\!](t_1)$	$N_{t_1} \cup N_{t_2}$	$E_{t_1} \cup E_{t_2} \cup \{(n, r_{t_2})\}$	$\lambda_{t_1}(m), m \in N_{t_1}$ $\lambda_{t_2}(m), m \in N_{t_2}$	r_{t_1}	$v_{t_1}(m), m \in N_{t_1}$ $v_{t_2}(m), m \in N_{t_2}$
$[\![\mathsf{replace}(n,t_2)]\!](t_1)$	$N_{t_1} \cup N_{t_2}$ $\backslash \mathsf{desc}_{t_1}(n)$	$E_{t_1} \cup E_{t_2} \cup$ $\{(\mathsf{parent}_{t_1}(n), r_{t_2})\} \backslash$ $\mathsf{desc}^e_{t_1}(n)$	$\lambda_{t_1}(m)$, $m \in (N_{t_1} \setminus \{n\})$ $\lambda_{t_2}(m), m \in N_{t_2}$	r_{t_1}	$v_{t_1}(m)$, $m \in (N_{t_1} \backslash \{n\})$ $v_{t_2}(m), m \in N_{t_2}$
$[\![\mathsf{replace}(n,s)]\!](t_1)$	N_{t_1}	E_{t_1}	$\lambda_{t_1}(m), m \in N_{t_1}$	r_{t_1}	$v_{t_1}(m)$, $m \in (N_{t_1} \backslash \{n\})$ $v_{t_1}(n) = s$
$[\![\mathsf{delete}(n)]\!](t_1)$	$N_{t_1} \setminus \mathsf{desc}_{t_1}(n)$	$E_{t_1} \setminus \mathsf{desc}^e_{t_1}(n)$	$\lambda_{t_1}(m)$, $m \in (N_{t_1} \backslash \mathsf{desc}_{t_1}(n))$	r_{t_1}	$v_{t_1}(m)$, $m \in (N_{t_1} \backslash \mathsf{desc}_{t_1}(n))$

An update operation $\mathsf{insert}(n, t_2)$, $\mathsf{replace}(n, t_2)$, $\mathsf{replace}(n, s)$ or $\mathsf{delete}(n)$ is *valid* with respect to tree t_1 provided $n \in N_{t_1}$ and t_2, if present, does not overlap with t_1 (that is, $N_{t_1} \cap N_{t_2} = \emptyset$). We also consider *update sequences* $op_1; \ldots; op_n$ with the (standard) semantics $[\![op_1; \ldots; op_n]\!](t_1) = [\![op_n]\!]([\![op_{n-1}]\!](\cdots [\![op_1]\!](t_1)))$. A sequence of updates $op_1; \ldots; op_n$ is valid with respect to t_0 if for each $i \in \{1, \ldots, n\}$, op_{i+1} is valid with respect to t_i, where $t_1 = [\![op_1]\!](t_0)$, $t_2 = [\![op_2]\!](t_1)$, *etc*. The result of a valid update (or valid sequence of updates) exists and is unique up to tree isomorphism. We restrict attention to valid updates and sequences in the rest of the paper.

3.2 Access Control Framework

We use the notion of *update access type* to specify the access authorizations in our context. Our update access types are inspired from the XAcU^{annot} language discussed in [13]. Authors followed the idea of *security annotations* introduced in [11] to specify the access authorizations for XML documents in the presence of a DTD.

Definition 5 (Update Access Types). Given a DTD D, an *update access type* (*UAT*) defined over D is of the form $(A, \mathsf{insert}(B_1))$, $(A, \mathsf{replace}(B_1, B_2))$, $(A, \mathsf{replace}(\mathsf{str}, \mathsf{str}))$ or $(A, \mathsf{delete}(B_1))$, where A is an element type in D, B_1 and B_2 are subelement types of A and $B_1 \neq B_2$.

Intuitively, an *UAT* represents a set of *atomic update operations*. More specifically, for t an instance of DTD 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:

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

$$\frac{\lambda_t(n) = B, t' \in I_D(B'), \lambda_t(parent_t(n)) = A, B \neq B'}{\mathsf{replace}(n, t')\ \mathsf{matches}_t\ (A, \mathsf{replace}(B, B'))}$$

$$\frac{\lambda_t(n) = \mathsf{str}, \lambda_t(\mathsf{parent}_t(n)) = A}{\mathsf{replace}(n, s)\ \mathsf{matches}_t (A, \mathsf{replace}(\mathsf{str}, \mathsf{str}))}$$

It is trivial to translate our update access types to XAcU^{annot} security annotations. In this work we assume that the evaluation of an update operation on a tree that conforms to a DTD D results in a *tree that conforms to* D. It is clear then that each update access type only makes sense for specific element types. For our example DTD, the update access type $(A, \mathsf{delete}(C))$ is not meaningful because allowing the deletion of a C-element would result in an XML document that does not conform to the DTD, and therefore, the update will be rejected. Similar for $(R, \mathsf{delete}(A))$ or $(R, \mathsf{insert}(A))$. But, $(B, \mathsf{delete}(E))$ and $(B, \mathsf{insert}(E))$ are relevant for this specific DTD. The relation uat valid_in D, which indicates that an update access type uat is valid for the DTD D, is defined as follows:

$$\frac{Rg(A) := B_1^*}{(A, \mathsf{insert}(B_1))\ \mathsf{valid\_in}\ D} \qquad \frac{Rg(A) := B_1*}{(A, \mathsf{delete}(B_1))\ \mathsf{valid\_in}\ D}$$

$$\frac{Rg(A) := \mathsf{str}}{(A, \mathsf{replace}(\mathsf{str}, \mathsf{str}))\ \mathsf{valid\_in}\ D} \qquad \frac{Rg(A) := B_1 + \cdots + B_n, i, j \in [1, n] \quad i \neq j}{(A, \mathsf{replace}(B_i, B_j))\ \mathsf{valid\_in}\ D}$$

We define the set of valid *UATs* for a given DTD D as $\mathsf{valid}(D) = \{uat \mid uat\ \mathsf{valid\_in}\ D\}$. A *security policy* will be defined by a set of *allowed* and *forbidden* valid *UATs*.

Definition 6. A security policy P defined over a DTD D, is represented by $(\mathcal{A}, \mathcal{F})$ where $\mathcal{A}$ is the set of *allowed* and $\mathcal{F}$ the set of *forbidden* update access types defined over D such that $\mathcal{A} \subseteq \mathsf{valid}(D)$, $\mathcal{F} \subseteq \mathsf{valid}(D)$ and $\mathcal{A} \cap \mathcal{F} = \emptyset$. A security policy is *total* if $\mathcal{A} \cup \mathcal{F} = \mathsf{valid}(D)$, otherwise it is *partial*.

Example 2. Consider the DTD D in Fig. 1 and the total policy $P = (\mathcal{A}, \mathcal{F})$ where $\mathcal{A}$ is:

$(R, \mathsf{replace}(A, B))$	$(R, \mathsf{replace}(B, J))$	$(R, \mathsf{replace}(J, K))$	$(R, \mathsf{replace}(K, J))$
$(R, \mathsf{replace}(K, B))$	$(C, \mathsf{insert}(F))$	$(C, \mathsf{delete}(F))$	$(D, \mathsf{insert}(F))$
$(D, \mathsf{delete}(F))$	$(F, \mathsf{replace}(\mathsf{str}, \mathsf{str}))$	$(B, \mathsf{insert}(E))$	$(B, \mathsf{delete}(E))$
$(E, \mathsf{insert}(G))$	$(E, \mathsf{delete}(G))$	$(G, \mathsf{replace}(I, H))$	$(J, \mathsf{insert}(G))$
$(J, \mathsf{delete}(G))$	$(D, \mathsf{insert}(F))$	$(D, \mathsf{delete}(F))$	$(H, \mathsf{replace}(\mathsf{str}, \mathsf{str}))$
$(I, \mathsf{replace}(\mathsf{str}, \mathsf{str}))$	$(K, \mathsf{replace}(\mathsf{str}, \mathsf{str}))$		

and $\mathcal{F} = \mathsf{valid}(D) \setminus \mathcal{A}$. On the other hand, $P = (\mathcal{A}, \emptyset)$ is a partial policy. □

The operations that are allowed by a policy $P = (\mathcal{A}, \mathcal{F})$ on an XML tree t, denoted by $[\![\mathcal{A}]\!](t)$, are the union of the atomic update operations matching each *UAT* in $\mathcal{A}$. More formally, $[\![\mathcal{A}]\!](t) = \{op \mid op\ \mathsf{matches}_t\ uat, \text{and } uat \in \mathcal{A}\}$. We say that an update sequence $op_1; \ldots; op_n$ is allowed on t provided the sequence is valid on t and $op_1 \in [\![\mathcal{A}]\!](t)$, $op_2 \in [\![\mathcal{A}]\!]([\![op_1]\!](t))$, *etc.*[1] Analogously, the forbidden operations are $[\![\mathcal{F}]\!](t) = \{op \mid op\ \mathsf{matches}_t\ uat, \text{and } uat \in \mathcal{F}\}$. If a policy P is *total*, its semantics is given by its allowed updates, i.e. $[\![P]\!](t) = [\![\mathcal{A}]\!](t)$. The semantics of a partial policy is studied in detail in Section 4.1.

4 Consistent 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 7. *A policy $P = (\mathcal{A}, \mathcal{F})$ defined over a DTD D is consistent if for every XML tree t that conforms to D, there does not exist a valid sequence of updates $op_1; \ldots; op_n$ that is allowed on t and a valid update $op_0 \in [\![\mathcal{F}]\!](t)$ such that:*

$$[\![op_1; \ldots; op_n]\!](t) \equiv [\![op_0]\!](t).$$

[1] Note that this is *not* the same as $\{op_1, \ldots, op_n\} \subseteq [\![\mathcal{A}]\!](t)$.

In our framework inconsistencies can be classified as: insert/delete and replace.

Inconsistencies due to *insert/delete* operations arise when the policy *allows* one to insert *and* delete nodes of element type A whilst *forbidding* some operation in some descendant element type of A. In this case, the forbidden operation can be simulated by first deleting an A-element and then inserting a new A-element after having done the necessary modifications.

There are two kinds of inconsistencies created by *replace* operations on a production rule $A \rightarrow B_1 + \cdots + B_n$ of a DTD. First, if we are allowed to replace B_i by B_j and B_j by B_k but not B_i by B_k, then one can simulate the latter operation by a sequence of the first two. Second, consider that we are allowed to replace some element type B_i with an element type B_j and vice versa. If some operation in the subtree of *either* B_i or B_j is forbidden, then it is evident that one can simulate the forbidden operation by a sequence of allowed operations, leading to an inconsistency.

We say that *nothing is forbidden below an element type* A in a policy $P = (\mathcal{A}, \mathcal{F})$ defined over D if for every B_i s.t. $A \leq_D B_i$ and every $(B_i, x) \in \mathsf{valid}(D)$, $(B_i, x) \notin \mathcal{F}$. If $A \rightarrow B_1 + \ldots + B_n$, then we define the *replace graph* $\mathcal{G}_A = (\mathcal{V}_A, \mathcal{E}_A)$ for a policy $P = (\mathcal{A}, \mathcal{F})$, where *i)* $\mathcal{V}_A$ is the set of nodes for $B_1, \ldots, B_n$ and *ii)* $(B_i, B_j) \in \mathcal{E}_A$ if there exists $(A, \mathsf{replace}(B_i, B_j)) \in \mathcal{A}$. Also, the set of *forbidden edges* of A, is $\mathcal{E}_A^F = \{(B_i, B_j) \mid (A, \mathsf{replace}(B_i, B_j)) \in \mathcal{F}\}$. We say that a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is *transitive* if $(x, y), (y, z) \in \mathcal{E}$ then $(x, z) \in \mathcal{E}$. We write $\mathcal{G}_A^+$ for the transitive graph of $\mathcal{G}_A$. The following theorem characterizes policy consistency:

Theorem 1. *A policy $P = (\mathcal{A}, \mathcal{F})$ defined over DTD D is consistent if and only if for every production rule:*

1. *$A \rightarrow B*$ in D, if $(A, \mathsf{insert}(B)) \in \mathcal{A}$ and $(A, \mathsf{delete}(B)) \in \mathcal{A}$, then nothing is forbidden below B*
2. *$A \rightarrow B_1 + \cdots + B_n$ in D, if for every edge (B_i, B_j) in $\mathcal{G}_A^+$, $(B_i, B_j) \notin \mathcal{E}_A^F$, and*
3. *$A \rightarrow B_1 + \cdots + B_n$ in D, if for every $i \in [1, \ldots n]$, if B_i is contained in a cycle in $\mathcal{G}_A$ then nothing is forbidden below B_i.*

Proof (Sketch). The forward direction is straightforward, since if any of the rules are violated an inconsistency can be found, as sketched above. For the reverse direction, we first need to reduce allowed update sequences to certain (allowed) normal forms that are easier to analyze, then the reasoning proceeds by cases. A full proof is given in [4]. □

In the case of total policies, condition 2 in Theorem 1 amounts to requiring that the replace graph $\mathcal{G}_A$ is transitive (i.e., $\mathcal{G}_A = \mathcal{G}_A^+$).

Example 3. (example 2 continued) The total policy P is inconsistent because:

- $(E, \mathsf{insert}(G))$ and $(E, \mathsf{delete}(G))$ are in $\mathcal{A}$, but $(G, \mathsf{replace}(H, I)) \in \mathcal{F}$ (condition 1, Theorem 1),
- $(R, \mathsf{replace}(A, J))$, $(R, \mathsf{replace}(A, K))$ and $(R, \mathsf{replace}(B, K))$ are in $\mathcal{F}$ (condition 2, Theorem 1), and
- There are cycles in $\mathcal{G}_R$ involving both B and J, but below both of them there is a forbidden *UAT*, namely $(G, \mathsf{replace}(H, I))$ (condition 3, Theorem 1). □

It is easy to see that we can check whether properties 1, 2, and 3 hold for a policy using standard graph algorithms:

Proposition 1. *The problem of deciding policy consistency is in* PTIME.

We wish to emphasize that consistency is highly sensitive to the design of policies and update types. For example, we have consciously chosen to *omit* an update type $(A, \mathsf{replace}(B_i, B_i))$ for an element type A in the DTD whose production rule is either of the form $B*$ or $B_1 + \ldots + B_n$. Consider the case of a conference management system where a $paper$ element has a $decision$ and a $title$ subelement. Suppose that the policy allows the author of the paper to *replace* a $paper$ with another $paper$ element, but forbids to change the value of the $decision$ subelement. This policy is inconsistent since by replacing a $paper$ element by another with a different $decision$ subelement we are able to perform a forbidden update. In fact, the $\mathsf{replace}(paper, paper)$ can simulate any other update type applying below a $paper$ element. Thus, if the policy forbids replacement of $paper$ nodes, then it would be inconsistent to allow any other operation on $decision$ and $title$. Because of this problem, we argue that update type $(A, \mathsf{replace}(B_i, B_i))$ should not be used in policies. Instead, more specific privileges should be assigned individually, *e.g.,* by allowing replacement of the text values of $title$ or $decision$ element types.

4.1 Partial Policies

Partial policies may be smaller and easier to maintain than total policies, but are ambiguous because some permissions are left unspecified. An access control mechanism must either allow or deny a request. One solution to this problem (in accordance with the *principle of least privilege*) might be to deny access to the unspecified operations. However, there is no guarantee that the resulting total policy is *consistent*. Indeed, it is not obvious that a partial policy (even if consistent) has *any* consistent total extension. We will now show how to find consistent extensions, if they exist, and in particular how to find a "least-privilege" consistent extension; these turn out to be unique when they exist so they seem to be a natural choice for defining the meaning of a partial policy.

For convenience, we write $\mathcal{A}_P$ and $\mathcal{F}_P$ for the allowed and forbidden sets of a policy P; i.e., $P = (\mathcal{A}_P, \mathcal{F}_P)$. We introduce an *information ordering* $P \sqsubseteq Q$, defined as $\mathcal{A}_P \subseteq \mathcal{A}_Q$ and $\mathcal{F}_P \subseteq \mathcal{F}_Q$; that is, Q is "more defined" than P. In this case, we say that Q extends P. We say that a partial policy P is *quasiconsistent* if it has a consistent total extension. For example, a partial policy on the DTD of Figure 1 which allows $(B, \mathsf{insert}(E))$, $(B, \mathsf{delete}(E))$, and denies $(H, \mathsf{replace}(\mathsf{str}, \mathsf{str}))$ is not quasiconsistent, because any consistent extension of the policy has to allow $(H, \mathsf{replace}(\mathsf{str}, \mathsf{str}))$.

We also introduce a *privilege ordering* on total policies $P \leq Q$, defined as $\mathcal{A}_P \subseteq \mathcal{A}_Q$; that is, Q allows every operation that is allowed in P. This ordering has unique greatest lower bounds $P \wedge Q$ defined as $(\mathcal{A}_P \cap \mathcal{A}_Q, \mathcal{F}_P \cup \mathcal{F}_Q)$. We now show that every quasiconsistent policy has a *least-privilege* consistent extension $P^\dagger$; that is, $P^\dagger$ is consistent and $P^\dagger \leq Q$ whenever Q is a consistent extension of P.

Lemma 1. *If P_1, P_2 are consistent total extensions of P_0 then $P_1 \wedge P_2$ is also a consistent extension of P_0.*

Proof. It is easy to see that if P_1, P_2 extend P_0 then $P_1 \wedge P_2$ extends P_0. Suppose $P_1 \wedge P_2$ is inconsistent. Then there exists an XML tree t, an atomic operation $op_0 \in$

$[\![\mathcal{F}_{P_1 \wedge P_2}]\!](t)$, a sequence $\overline{op}$ allowed on t by $P_1 \wedge P_2$, such that $[\![op_0]\!](t) = [\![\overline{op}]\!](t)$. Now $\mathcal{A}_{P_1 \wedge P_2} = \mathcal{A}_{P_1} \cap \mathcal{A}_{P_2}$, so op_0 must be forbidden by either P_1 or P_2. On the other hand, $\overline{op}$ must be allowed by *both* P_1 and P_2, so $t, op_0, \overline{op}$ forms a counterexample to the consistency of P_1 (or symmetrically P_2). □

Proposition 2. *Each quasiconsistent policy P has a unique $\leq$-least consistent total extension $P^\dagger$.*

Proof. Since P is quasiconsistent, the set $S = \{Q \mid P \sqsubseteq Q, Q \text{ consistent}\}$ is finite, nonempty, and closed under $\wedge$, so has a $\leq$-least element $P^\dagger = \bigwedge S$. □

Finally, we show how to find the least-privilege consistent extension, or determine that none exists (and hence that the partial policy is not quasiconsistent). Define the operator $T : \mathcal{P}(\mathsf{valid}(D)) \rightarrow \mathcal{P}(\mathsf{valid}(D))$ as:

$$\begin{aligned} T(S) = S &\cup \{(C, x) \mid B \leq_D C, Rg(A) = B^*, \{(A, \mathsf{insert}(B)), (A, \mathsf{delete}(B))\} \subseteq S\} \\ &\cup \{(C, x) \mid B_i \leq_D C, Rg(A) = B_1 + \ldots + B_n, (B_i, B_i) \in \mathcal{G}_A^+(S)\} \\ &\cup \{(A, \mathsf{replace}(B_i, B_k)) \mid Rg(A) = B_1 + \ldots + B_n, (B_i, B_k) \in \mathcal{G}_A^+(S)\} \end{aligned}$$

where $\mathcal{G}_A^+(S)$ is the transitive graph of A for the partial policy S.

Lemma 2. *If $uat \in T(S)$ then for any valid operation op_0 matching uat on t there exists a valid sequence of operations $\overline{op}$ allowed on t by S such that $[\![op_0]\!](t) = [\![\overline{op}]\!](t)$.*

Theorem 2. *Let P be a partial policy. The following are equivalent: (1) P is quasiconsistent, (2) P is consistent (3) $T(\mathcal{A}_P) \cap \mathcal{F}_P = \emptyset$.*

Proof. To show (1) implies (2), if P' is a consistent extension of P, then any inconsistency in P would be an inconsistency in P', so P must be consistent. To show (2) implies (3), we prove the contrapositive. If $T(\mathcal{A}_P) \cap \mathcal{F}_P \neq \emptyset$ then choose $uat \in T(\mathcal{A}_P) \cap \mathcal{F}_P$. Choose an arbitrary tree t and atomic update op satisfying $op_0 \in [\![uat]\!](t)$. By Lemma 2, there exists a sequence $\overline{op}$ allowed by $\mathcal{A}_P$ on t with $[\![\overline{op}]\!](t) = [\![op_0]\!](t)$. Hence, policy P is inconsistent. Finally, to show that (3) implies (1), note that $(T(\mathcal{A}_P), \mathsf{valid}(D) \setminus T(\mathcal{A}_P))$ extends P and is consistent provided $T(\mathcal{A}_P) \cap \mathcal{F}_P = \emptyset$.

Indeed, for a (quasi-)consistent P, the least-privilege consistent extension of P is simply $P^\dagger = (T(\mathcal{A}_P), \mathsf{valid}(D) \setminus T(\mathcal{A}_P))$ (proof omitted). Hence, we can decide whether a partial policy is (quasi-)consistent and if so find $P^\dagger$ in PTIME.

5 Repairs

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.

There are several ways of defining these repairs. We might want to repair by changing the permissions of certain operations from allow to forbidden and vice versa; or we might give preference to some type of changes over others. Also, we can measure the minimality of the repairs as a minimal number of changes or a minimal set of changes under set inclusion.

Due to space restrictions, in this paper we will focus on finding repairs that transform *UATs* from *allowed* to *forbidden* and that minimize the number of changes. We believe that such repairs are a useful special case, since the repairs are guaranteed to be more restrictive than the original policy.

Definition 8. A policy $P' = (\mathcal{A}', \mathcal{F}')$ is a *repair* of a policy $P = (\mathcal{A}, \mathcal{F})$ defined over a DTD D iff: i) P' is a policy defined over D, ii) P' is consistent, and iii) $P' \leq P$.

A repair is *total* if $\mathcal{F}' = \mathsf{valid}(D) \setminus \mathcal{A}'$ and *partial* otherwise. Furthermore a repair $P' = (\mathcal{A}', \mathcal{F}')$ of $P(\mathcal{A}, \mathcal{F})$ is a *minimal-total-repair* if there is no total repair $P'' = (\mathcal{A}'', \mathcal{F}'')$ such that $|\mathcal{A}'| < |\mathcal{A}''|$ and a *minimal-partial-repair* if $\mathcal{F}' = \mathcal{F}$ and there is no partial repair $P'' = (\mathcal{A}'', \mathcal{F})$ such that $|\mathcal{A}'| < |\mathcal{A}''|$.

Given a policy $P = (\mathcal{A}, \mathcal{F})$ and an integer k, the total-repair (partial-repair) problem consists in determining if there exists a total-repair (partial-repair) $P' = (\mathcal{A}', \mathcal{F}')$ of policy P such that $|\mathcal{A} \setminus \mathcal{A}'| < k$. This problem can be shown to be NP-hard by reduction from the edge-deletion transitive-digraph problem [20].

Theorem 3. *The total-repair and partial-repair problem is* NP*-complete.*

If the DTD has no production rules of the type $A \rightarrow B_1 + \cdots + B_n$, then the total-repair problem is in PTIME.

5.1 Repair Algorithm

In this section we discuss a repair algorithm that finds a minimal repair of a total or partial policy. All the algorithms can be found in [4].

The algorithm to compute a minimal repair of a policy relies in the independence between inconsistencies *w.r.t.* insert/delete (Theorem 1, condition 1) and replace (Theorem 1, conditions 2 and 3) operations. In fact, a local repair of an inconsistency *w.r.t.* insert/delete operations will never solve nor create an inconsistency with respect to a replace operation and vice-versa. We will separately describe the algorithm for repairing the insert/delete inconsistencies and then the algorithm for the replace ones.

Both algorithms make use of the *marked DTD graph* $MG_D = (G_D, \mu, \chi)$ where μ is a function from nodes in $\mathcal{V}_D$ to $\{``+", ``-"\}$ and χ is a partial function from $\mathcal{V}_D$ to $\{\perp\}$. In a marked graph for a DTD D and a policy $P = (\mathcal{A}, \mathcal{F})$ *i)* each node in the graph is either marked with "+" (i.e., nothing is forbidden below the node) or with a "−" (i.e., there exists at least one update access type that is forbidden below the node). If, for nodes A and B in the DTD, *both* $(A, \mathsf{insert}(B))$ and $(A, \mathsf{delete}(B))$ are in $\mathcal{A}$ and $\mu(A) = ``-"$, then $\chi(A) = ``\perp"$. A marked graph is obtained from algorithm **markGraph** which takes as input a DTD graph and a policy P and traverses the DTD graph starting from the nodes with out-degree 0 and marks the nodes and edges as discussed above.

Example 4. Consider the graph for DTD D in Fig. 2(a) and policy $P = (\mathcal{A}, \mathcal{F})$, with $\mathcal{A}$ defined in Example 2. The result of applying **markGraph** to this DTD and policy is shown in Fig. 2(b). Notice that nodes B, E and J are marked with both a "−" and "$\perp$" since *i)* update access type $(G, \mathsf{replace}(H, I))$ is in $\mathcal{F}$ and *ii)* all insert and delete update access types for B, E and J are in $\mathcal{A}$. For readability purposes we do not show the multiplicities in the marked DTD graph. □

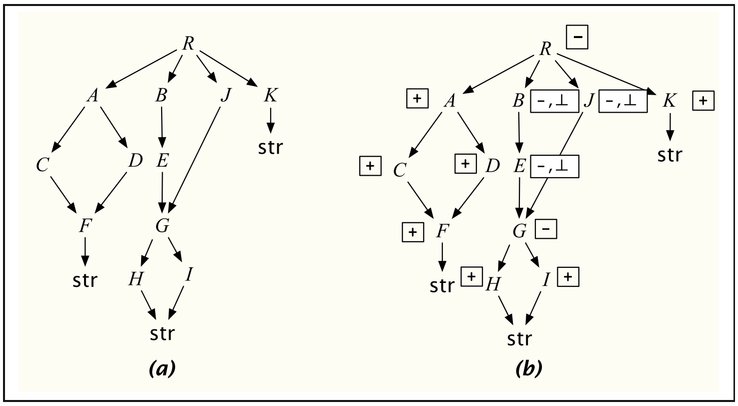

Fig. 2. DTD Graph (a) and Marked DTD Graph (b) for the DTD in Fig. 1

Repairing Inconsistencies for Insert and Delete Operations. Recall that if both the insert and delete operations are allowed at some element type and there is some operation below this type that is not allowed, then there is an inconsistency (see Theorem 1, condition 1). The marked DTD graph provides exactly this information: a node A is labeled with "$\perp$" if it is inconsistent w.r.t. *insert/delete* operations. For each such node and for the repair strategy that we have chosen, the inconsistency can be minimally repaired by removing either $(A, \mathsf{insert}(B))$ or $(A, \mathsf{delete}(B))$ from $\mathcal{A}$. Algorithm $\mathbf{InsDelRepair}$ in [4] takes as input a DTD graph G_D and a security policy $P = (\mathcal{A}, \mathcal{F})$ and returns a set of *UATs* to remove from $\mathcal{A}$ to restore consistency *w.r.t.* insert/delete-inconsistencies.

Example 5. Given the marked DTD graph in Fig. 2(b), it is easy to see that the *UATs* that must be repaired are associated with nodes B, J and E (all nodes are marked with "$\perp$"). The repairs that can be proposed to the user are to remove from $\mathcal{A}$ one *UAT* from each of the following sets: $\{(B, \mathsf{insert}(E)),\ (B, \mathsf{delete}(E))\}$, $\{(E, \mathsf{insert}(G)),\ (E, \mathsf{delete}(G))\}$ and $\{(J, \mathsf{insert}(G)),\ (J, \mathsf{delete}(G))\}$. □

Repairing Inconsistencies for Replace Operations. There are two types of inconsistencies related to replace operations (see Theorem 1, conditions 2–3): the first arises when some element type A is contained in some cycle and something is forbidden below it; the second arises when the replace graph $\mathcal{G}_A$ cannot be extended to a transitive graph without adding a forbidden edge in $\mathcal{E}_A^F$. In what follows we will refer to these type of inconsistencies as *negative-cycle* and *forbidden-transitivity*. By Theorem 3, the repair problem is NP-complete, and therefore, unless P = NP, there is no polynomial time algorithm to compute a minimal repair to the replace-inconsistencies. Our objective then, is to find an algorithm that runs in polynomial time and computes a repair that is not necessarily minimal.

Algorithm $\mathbf{ReplaceNaive}$ given in [4] traverses the marked graph MG_D and at each node, checks whether its production rule is of the form $A \to B_1 + \ldots + B_n$. If this is the case, it builds the replace graph for A, $\mathcal{G}_A$, and runs a modified version of the Floyd-Warshall algorithm [12]. The original Floyd-Warshall algorithm adds an edge (B, D) to the graph if there is a node C such that (B, C) and (C, D) are in the graph and (B, D) is not. Our modification consists on deleting either (B, C) or (C, D) if $(B, D) \in \mathcal{E}^F_A$, *i.e.,* if there is forbidden-transitivity. In this way, the final graph will satisfy condition 2 of Theorem 1. Also, if there are edges (B, C) and (C, B) and $\mu(C) =$ "$-$", *i.e.,* there is a negative-cycle, one of the two edges is deleted. Algorithm $\mathbf{ReplaceNaive}$ returns the set of edges to delete from each node to remove replace-inconsistencies.

Example 6. The replace graph $\mathcal{G}_G$ has no negative-cycles nor forbidden-transitivity, therefore it is not involved in any inconsistency. On the other hand, the replace graph $\mathcal{G}_R = (\mathcal{V}_R, \mathcal{E}_R)$, shown in Fig. 3(a) is the source of many inconsistencies. A possible execution of $\mathbf{ReplaceNaive}$ is: $(A, B), (B, J) \in \mathcal{E}_R$ but $(A, J) \in \mathcal{E}^F_R$, so (A, B) or (B, J) should be deleted, say (A, B). Now, (B, J), $(J, K) \in \mathcal{E}_R$ and $(B, K) \in \mathcal{E}^F_R$, therefore we delete either (B, J) or (J, K), say (B, J). Next, (K, J), $(J, K) \in \mathcal{E}_R$ and $\mu(J) =$ "$-$" in Fig. 2(b), therefore there is a negative-cycle and either (K, J) or (J, K) has to be deleted. If (K, J) is deleted, the resulting graph has no forbidden-transitivity nor negative-cycles. The policy obtained by removing $(R, \mathsf{replace}(A, B))$, $(R, \mathsf{replace}(B, J))$ and $(R, \mathsf{replace}(J, K))$ from $\mathcal{A}$ has no replace-inconsistencies. $\square$

The $\mathbf{ReplaceNaive}$ algorithm might remove more than the necessary edges to achieve consistency: in our example, if we had removed edge (B, J) at the first step, then we would have resolved the inconsistencies that involve edges (A, B), (B, J) and (J, K).

An alternative to algorithm $\mathbf{ReplaceNaive}$, that can find a solution closer to minimal repair, is algorithm $\mathbf{ReplaceSetCover}$ also given in [4]. This algorithm computes, using the Floyd-Warshall algorithm, the transitive closure of the replace graph $\mathcal{G}_A$ and labels each newly constructed edge e with a set of *justifications* $\mathcal{J}$. Each justification contains the sets of edges of $\mathcal{G}_A$ that were used to add e in $\mathcal{G}^+_A$. Also, if a node is found to be part of a negative-cycle, it is labeled with the justifications $\mathcal{J}$ of the edges in each cycle that contains the node. An edge or vertex might be justified by more than one set of edges. In fact, the number of justifications an edge or node might have is $O(2^{|\mathcal{E}_A|})$. To avoid the exponential number of justifications, $\mathbf{ReplaceSetCover}$ assigns at most $\mathfrak{J}$ justifications to each edge or node, where $\mathfrak{J}$ is a fixed number. This new labeled graph is then used to construct an instance of the minimum set cover problem (MSCP) [18]. The solution to the MSCP, can be used to determine the set of edges to remove from $\mathcal{G}_A$ so that none of the justifications that create inconsistencies are valid anymore. Because of the upper bound $\mathfrak{J}$ on the number of justifications, it might be the case that the graph still has forbidden-transitivity or negative-cycles. Thus, the justifications have to be computed once more and the set cover run again until there are no more replace inconsistencies.

Example 7. For $\mathfrak{J} = 1$, the first computation of justifications of $\mathbf{ReplaceSetCover}$ results in the graph in Fig. 3 (b) with the following justifications:

$$\mathcal{J}((A, J)) = \{\{(A, B), (B, J)\}\} \qquad \mathcal{J}((J, B)) = \{\{(J, K), (K, B)\}\}$$
$$\mathcal{J}((A, K)) = \{\{(A, B), (B, J), (J, K)\}\} \qquad \mathcal{J}(B) = \{\{(B, J), (J, K), (K, B)\}\}$$
$$\mathcal{J}((B, K)) = \{\{(B, J), (J, K)\}\} \qquad \mathcal{J}(J) = \{\{(J, K), (K, J)\}\}$$

Justifications for edges represent violations of transitivity. Justification for nodes represent negative-cycles. If we want to remove the inconsistencies, it is enough to delete one edge from each set in $\mathcal{J}$. $\square$

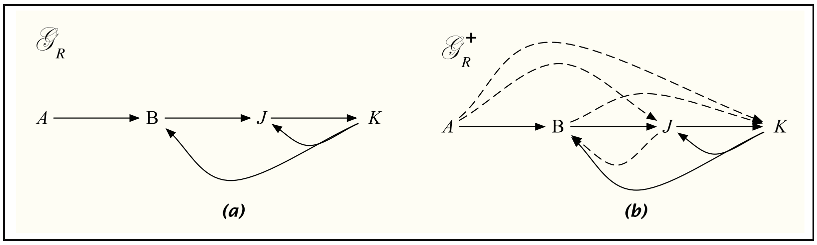

Fig. 3. Replace $\mathcal{G}_R$ (a) and Transitive Replace Graph $\mathcal{G}_R^+$(b)

The previous example shows that, for each element type A, replace-inconsistencies can be repaired by removing at least one edge from each of the justifications of edges and vertices in $\mathcal{G}_A^+$. It is easy to see that this problem can be reduced to the MSCP. An instance of the MSCP consists of a universe $\mathcal{U}$ and a set $\mathcal{S}$ of subsets of $\mathcal{U}$. A subset $\mathcal{C}$ of $\mathcal{S}$ is a set cover if the union of the elements in it is $\mathcal{U}$. A solution of the MSCP is a set cover with the minimum number of elements.

The set cover instance associated to $\mathcal{G}_A^+ = (\mathcal{V}_A, \mathcal{E}_A)$ and the set of forbidden edges $\mathcal{E}_A^F$, is $MSCP(\mathcal{G}_A^+, \mathcal{E}_A^F) = (\mathcal{U}, \mathcal{S})$ for i) $\mathcal{U} = \{s \mid s \in \mathcal{J}(e), e \in \mathcal{E}_A^F\} \cup \{ s \mid s \in \mathcal{J}(V), V \in \mathcal{V}_A\}$, and ii) $\mathcal{S} = \bigcup_{e \in \mathcal{E}} \mathcal{I}(e)$ where $\mathcal{I}(e) = \{s \mid s \in \mathcal{U}, e \in s\}$. Intuitively, $\mathcal{U}$ contains all the inconsistencies, and the set $\mathcal{I}(e)$ the replace-inconsistencies in which an edge e is involved. Notice that in this instance of the MSCP, $\mathcal{U}$ is a set of justifications, therefore, $\mathcal{S}$ is a set of sets of justifications.

Example 8. The minimum set cover instance, $MSCP(\mathcal{G}_R^+, \mathcal{E}_R^F) = (\mathcal{U}, \mathcal{S})$, is such that $\mathcal{U} = \{\{(A,B),(B,J),(J,K)\}, \{(A,B),(B,J)\}, \{(B,J),(J,K)\}, \{(J,K),(K,B)\}, \{(J,K),(K,J)\}, \{(K,J),(J,K)\}, \{(B,J),(J,K),(K,B)\}\}$ and $\mathcal{S} = \{\mathcal{I}((A,B)), \mathcal{I}((B,J)), \mathcal{I}((J,K)), \mathcal{I}((K,J)), \mathcal{I}((K,B))\}$. The extensions of $\mathcal{I}$ are given in Table 2, where each column corresponds to a set $\mathcal{I}$ and each row to an element in $\mathcal{U}$. Values 1 and 0 in the table represent membership and non-membership respectively. A minimum set cover of $MSCP(\mathcal{G}_R^+, \mathcal{E}_R^F)$ is $\mathcal{C} = \{\mathcal{I}((B,J)), \mathcal{I}((J,K))\}$, since $\mathcal{I}((B,J))$ covers all the elements of $\mathcal{U}$ except for the element $\{(A,B), (B,J)\}$, which is covered by $\mathcal{I}((J,K))$. Now, using the solution from the set cover, we remove edges (B,J) and (J,K) from $\mathcal{G}_R$. If we try to compute the justifications once again, it turns out that there are no more negative-cycles and that the graph is transitive. Therefore, by removing $(R, \mathsf{replace}(B,J))$ and $(R, \mathsf{replace}(J,K))$ from $\mathcal{A}$, there are no replace-inconsistencies in node R. $\square$

The set cover problem is MAXSNP-hard [18], but its solution can be approximated in polynomial time using a greedy-algorithm that can achieve an approximation factor of $\log(n)$ where n is the size of $\mathcal{U}$ [9]. In our case, n is $O(\mathfrak{J} \times |Ele|)$. In the ongoing example, the approximation algorithm of the set cover will return a cover of size 2. This is better than what was obtained by the **ReplaceNaive** algorithm. In order to decide which one is better, we need to run experiments to investigate the trade off between efficiency and the size of the repaired policy.

Algorithm **ReplaceRepair** will compute the set of *UATs* to remove from $\mathcal{A}$, by using either **ReplaceNaive** or **ReplaceSetCover** .

Table 2. Set cover problem

	$\mathcal{S}$				
$\mathcal{U}$	$\mathcal{I}((A,B))$	$\mathcal{I}((B,J))$	$\mathcal{I}((J,K))$	$\mathcal{I}((K,J))$	$\mathcal{I}((K,B))$
$\{(A,B),(B,J),(J,K)\}$	1	1	1	0	0
$\{(A,B),(B,J)\}$	1	1	0	0	0
$\{(B,J),(J,K)\}$	0	1	1	0	0
$\{(J,K),(K,B)\}$	0	0	1	0	1
$\{(J,K),(K,J)\}$	0	0	1	1	0
$\{(K,J),(J,K)\}$	0	0	1	1	0
$\{(B,J),(J,K),(K,B)\}$	0	1	1	0	1

Computation of a Repair. Algorithm **Repair** computes a new consistent policy $P' = (\mathcal{A}', \mathcal{F}')$ from $P = (\mathcal{A}, \mathcal{F})$ by removing from $\mathcal{A}$ the union of the *UATs* returned by algorithms **InsDelRepair** and **ReplaceRepair**. The algorithm is capable of computing total and partial repairs.

Theorem 4. *Given a total (partial) policy P, algorithm* **Repair** *returns a total (partial) repair of P.*

6 Conclusion

Access control policies attempt to constrain the actual operations users can perform, but are usually enforced in terms of syntactic representations of the operations. Thus, policies controlling update access to XML data may forbid certain operations but permit other operations that have the same effect. In this paper we have studied such *inconsistency* vulnerabilities and shown how to check consistency and repair inconsistent policies. This is, to our knowledge, the first investigation of consistency and repairs for XML write-access control policies. We also considered consistency and repair problems for partial policies which may be more convenient to write since many privileges may be left unspecified.

Cautis, Abiteboul and Milo in [6] discuss XML update constraints to restrict insert and delete updates, and propose to detect updates that violate these constraints by measuring the size of the modification of the database. This approach differs from our security framework for two reasons: a) we consider in addition to insert/delete also *replace* operations and b) we require that each operation in the sequence of updates does not violate the security constraints, whereas in their case, they require that only the input and output database satisfies them.

Minimal repairs are used in the problem of returning consistent answers from inconsistent databases [1]. There, a consistent answer is defined in terms of all the minimal repairs of a database. In [3] the set cover problem was used to find repairs of databases *w.r.t.* denial constraints.

There are a number of possible directions for future work, including running experiments for the proposed algorithms, studying consistency for more general security policies specified using XPath expressions or constraints, investigating the complexity of and algorithms for other classes of repairs, and considering more general DTDs.

Acknowledgments. We would like to thank Sebastian Maneth and Floris Geerts for insightful discussions and comments.

References

1. Arenas, M., Bertossi, L., Chomicki, J.: Consistent Query Answers in Inconsistent Databases. In: PODS, pp. 68–79. ACM Press, New York (1999)
2. Bertino, E., Ferrari, E.: Secure and Selective Dissemination of XML Documents. ACM TISSEC 5(3), 290–331 (2002)
3. Bertossi, L., Bravo, L., Franconi, E., Lopatenko, A.: Complexity and Approximation of Fixing Numerical Attributes in Databases Under Integrity Constraints. In: Bierman, G., Koch, C. (eds.) DBPL 2005. LNCS, vol. 3774, pp. 262–278. Springer, Heidelberg (2005)
4. Bravo, L., Cheney, J., Fundulaki, I.: Repairing Inconsistent XML Write-Access Control Policies (August 2007), `http://arxiv.org/abs/0708.2076`
5. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible Markup Language (XML) 1.0 (Fourth Edition) (September 2006), `http://www.w3.org/TR/REC-xml/`
6. Cautis, B., Abiteboul, S., Milo, T.: Reasoning about XML Update Constraints. In: PODS, pp. 195–204 (2007)
7. Centonze, P., Naumovich, G., Fink, S.J., Pistoia, M.: Role-Based Access Control Consistency Validation. In: ISSTA, pp. 121–132. ACM Press, New York (2006)
8. Chamberlin, D., Florescu, D., Robie, J.: XQuery Update Facility. W3C Working Draft (July 2006), `http://www.w3.org/TR/xqupdate/`
9. Chvatal, V.: A Greedy Heuristic for the Set Covering Problem. Mathematics of Operations Research 4, 233–235 (1979)
10. Damiani, E., De Capitani di, S., Paraboschi, S., Samarati, P.: A Fine-grained Access Control System for XML Documents. ACM TISSEC 5(2), 169–202 (2002)
11. Fan, W., Chan, C.-Y., Garofalakis, M.: Secure XML Querying with Security Views. In: ACM SIGMOD, pp. 587–598. ACM Press, New York (2004)
12. Floyd, R.: Algorithm 97: Shortest path. Communications of the ACM 5(6), 345 (1962)
13. Fundulaki, I., Maneth, S.: Formalizing XML Access Control for Update Operations. In: SACMAT, pp. 169–174 (2007)
14. Fundulaki, I., Marx, M.: Specifying Access Control Policies for XML Documents with XPath. In: SACMAT, pp. 61–69 (2004)
15. Kuper, G., Massacci, F., Rassadko, N.: Generalized XML Security Views. In: SACMAT, pp. 77–84 (2005)
16. Lim, C.-H., Park, S., Son, S.H.: Access control of XML documents considering update operations. In: ACM Workshop on XML Security, pp. 49–59. ACM Press, New York (2003)
17. Murata, M., Tozawa, A., Kudo, M., Hada, S.: XML Access Control Using Static Analysis. ACM TISSEC 9(3), 290–331 (2006)
18. Papadimitriou, C.: Computational Complexity. Addison-Wesley, Reading (1994)
19. Stoica, A., Farkas, C.: Secure XML Views. In: IFIP WG 11.3, vol. 256, pp. 133–146. Kluwer, Dordrecht (2002)
20. Yannakakis, M.: Edge-Deletion Problems. SIAM Journal on Computing 10(2), 297–309 (1981)