ACCOn: Checking Consistency of XML Write-Access Control Policies

Loreto Bravo[1] James Cheney[1] Irini Fundulaki[1,2]
[1]University of Edinburgh, UK [2]ICS-FORTH, Greece

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. ACCOn implements *i) consistency checking* algorithms that examine whether a write-access control policy defined over a DTD is inconsistent and *ii) repair algorithms* that propose repairs to an inconsistent policy to obtain a consistent one.

1. MOTIVATION

XML access control has received much attention over the last years as the amount of sensitive XML data exchanged between applications is increasing. Access control techniques for XML data have been considered extensively for *read-only queries* [4, 8, 9, 11, 12, 13]. However, the problem of controlling *write access* is relatively new and has not received much attention. An important problem in this context is the presence of a certain type of vulnerabilities, here called *inconsistencies*, that allow *one explicitly forbidden update operation to be simulated by a sequence of allowed ones*.

In general, an XML write-access control policy specifies the update actions a user can perform based on the *syntax* of the update and *not* its actual *behavior*. Thus, it is possible that a single update request which is explicitly forbidden by the policy can nevertheless be simulated by a sequence of more than one allowed update requests.

Consider for example the XML DTD in Fig. 1 that describes patient data. A *patient* has a *name* and is associated with zero or more treatments. A *treatment* consists of a *drug* that was prescribed to the patient and that can be one of *placebo*, *presDrug* (prescription) and *OTC* (off-the-counter) drug, a *diagnosis* and the *date* of a patient's visit. The XML document shown in Fig. 2(a) is an instance of the hospital DTD shown in Fig. 1. The document can be updated and queried by different users, e.g., doctors, nurses, administrators. A user is allowed to perform certain updates or access only part of the data. For example, a nurse is allowed to

EDBT'08, March 25–30, 2008, Nantes, France.

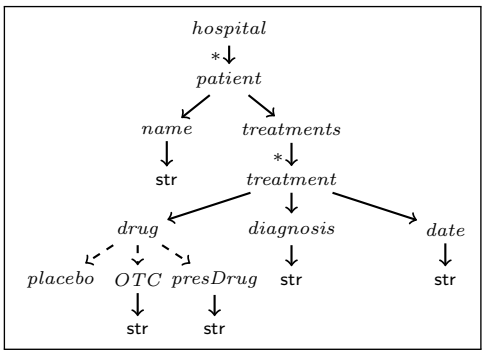


Figure 1: Hospital DTD Graph

insert and *delete* patients, but she *cannot modify* a patient's diagnosis or change a prescription drug to an off-the-counter drug. It is easy to see that the diagnosis of a patient can be changed by deleting a patient record and then inserting it back again with a modified value of diagnosis. Thus, a forbidden update request can be achieved by a sequence of allowed ones. We call an access control policy with this characteristic *inconsistent*.

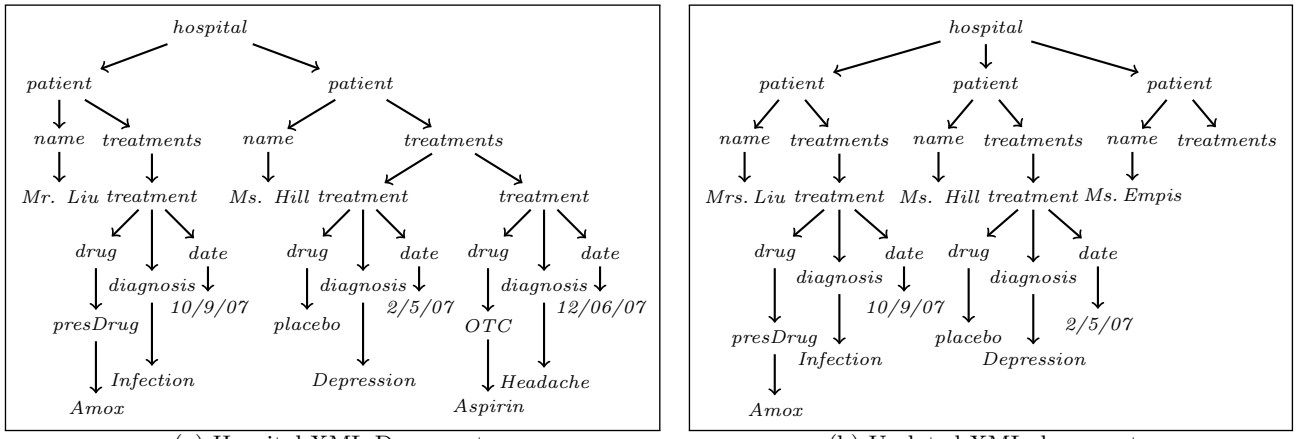
It is important to be able to detect inconsistencies and suggest possible ways of repairing policies in order to ensure their consistency. We addressed this problem in [5] for a particular class of inconsistencies in XML write-access control policies, where we: *i)* provided a formal definition of consistency *ii)* showed that checking consistency for the studied type of access policies is in PTIME *iii)* developed a polynomial time algorithm for checking consistency and *iv)* suggested repair algorithms for fixing the detected inconsistencies.

In this demo, we will present ACCOn, a system that implements the consistency checking and repair algorithms proposed in [5] and [6].

The outline of this demonstration proposal is the following: in Section 2 we introduce some basic concepts about XML write-access control. In Section 3 we explain when an access control policy is inconsistent and using the ongoing example we present alternative algorithms to find possible repairs. Finally, in Section 4 we describe the demonstration's objectives.

2. PRELIMINARIES

DTDs and XML Documents. We consider *structured* XML DTDs as discussed in [9]. Although not all DTDs are syntactically representable in this form, one can (as argued



(a) Hospital XML Document (b) Updated XML document

Figure 2: XML documents

by [9]) represent more general DTDs by introducing new element types.

A DTD D consists of a finite set of *element types* Ele, a distinguished type rt in Ele called the *root type* and a function Rg that maps a given element type A to a regular expression of the form $Rg(A) := \mathsf{str} \mid \epsilon \mid B_1, \ldots, B_n \mid B_1 + \ldots + B_n \mid B_1*$. The $B_i \in Ele$ are distinct, ",", "+" and "$*$" stand for *concatenation*, *disjunction* and *Kleene star* respectively, ϵ for the EMPTY element content and str for text values. A DTD can be represented as a directed acyclic graph that we call *DTD graph*. In it, conjunctions and disjunctions are represented by solid and dashed lines respectively.

We call $A \rightarrow Rg(A)$ 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. The production rules for the DTD graph shown in Fig. 1 are:

$hospital \rightarrow patient*$	$placebo \rightarrow \epsilon$
$patient \rightarrow name, treatments$	$presDrug \rightarrow \mathsf{str}$
$treatments \rightarrow treatment*$	$OTC \rightarrow \mathsf{str}$
$treatment \rightarrow drug, diagnosis, date$	$diagnosis \rightarrow \mathsf{str}$
$drug \rightarrow placebo \mid presDrug \mid OTC$	$date \rightarrow \mathsf{str}$
$name \rightarrow \mathsf{str}$	

We model an XML document t as a rooted tree. An XML document is said to be *valid* w.r.t an XML DTD D, if it conforms to the constraints (i.e., production rules) defined by D. It is easy to see that the XML document shown in Fig. 2(a) is valid w.r.t the DTD of Fig. 1.

XML Updates. There have been several XML update language proposals [3, 7, 15, 16, 17]. In ACCOn, we consider the *delete*, *replace* and *insert* update operations proposed in the XQuery Update Facility document [7]. In our context, an update operation is applied to a set of nodes specified by an XPath *target* expression. In a *delete* operation, *target* specifies the XML nodes to be deleted together with their descendants. The *insert* and *replace* operations have an additional component called *source* that is an XML tree or a text value. For the *insert* operation, *target* determines the node to which *source* will be inserted as a child node. Finally in the case of *replace* operations *target* specifies the node whose subtree will be replaced by *source*.

Examples of the update operations on the XML tree shown in Fig. 2(a) are shown below. The resulting XML document is shown in Fig. 2(b).

1. *delete* $//treatment[date =$ "12/06/07"]
2. *insert*
<patient>
<name>Ms. Empis</name>
<treatments/>
</patient> *into* /hospital
3. *replace* $//name[. =$ "Mr.Liu"] *with*
<name>Mrs. Liu</name>

For the purpose of our work, we abstract from the syntax of the XQuery operations and we consider *atomic update operations*. These operate on the XML nodes obtained by evaluating the *target* expression of an update operation on the document in question. A $\mathsf{delete}(n)$ operation deletes 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, but since we consider unordered XML trees, we deal only with $\mathsf{insert\ into}(n, t)$ (for readability purposes, we are going to write $\mathsf{insert}(n, t)$) which inserts the root of t as child of n. We also consider *update sequences* $op_1; \ldots; op_n$ with the standard semantics: $[\![op_1; \ldots; op_n]\!](t_1) = [\![op_n]\!]([\![op_{n-1}]\!](\cdots [\![op_1]\!](t_1)))$ where $[\![op_i]\!](t)$ is the document obtained by evaluating op_i on tree t.

XML Write-Access Policies An access control policy consists of a set of *rules* or *authorizations* that determine whether a user is allowed to perform some action on the data. These rules can be either XPath-based or annotation-based. In this work we use *update access types* that are similar to the latter. We have based our access types on the XAcU^{annot} language discussed in [10]. The language follows the idea of *security annotations* introduced in [9] to specify the access authorizations for XML documents in the presence of a DTD.

Given a DTD D, an *update access type* (*UAT*) defined over D is of the form *i)* $(A, \mathsf{insert}(B_1))$, *ii)* $(A, \mathsf{replace}(B_1, B_2))$, *iii)* $(A, \mathsf{replace}(\mathsf{str}, \mathsf{str}))$ or *iv)* $(A, \mathsf{delete}(B_1))$ where A is an element type in D, and B_1 and B_2 are distinct subelement types of A.

For our XML DTD shown in Fig. 1 and for the update privileges of nurses that were discussed in the motivating example, we can write the following update access types:
(*hospital*, insert (*patient*)) (*drug*,replace (*presDrug*,*OTC*))
(*hospital*, delete (*patient*)) (*diagnosis*, replace (str,str))
Intuitively, an *UAT* represents a set of *atomic update operations*. For example, the UAT $(hospital, \mathsf{insert}(patient))$

Set of Allowed UATs A

$\mathcal{A}_1$:($treatments$, insert($treatment$)) $\mathcal{A}_5$:(OTC, replace(str, str))
$\mathcal{A}_2$:($drug$, replace(OTC, $presDrug$)) $\mathcal{A}_6$:($date$, replace(str, str))
$\mathcal{A}_3$:($drug$, replace($presDrug$, OTC)) $\mathcal{A}_7$:($hospital$, insert($patient$))
$\mathcal{A}_4$:($drug$, replace($placebo$, OTC)) $\mathcal{A}_8$:($hospital$, delete($patient$))

Set of Forbidden UATs F

$\mathcal{F}_1$:($treatments$, delete($treatment$)) $\mathcal{F}_5$:($name$, replace(str, str))
$\mathcal{F}_2$:($drug$, replace($presDrug$, $placebo$)) $\mathcal{F}_6$:($diagnosis$, replace(str, str))
$\mathcal{F}_3$:($drug$, replace($placebo$, $presDrug$)) $\mathcal{F}_7$:($presDrug$, replace(str, str))
$\mathcal{F}_4$:($drug$, replace(OTC, $placebo$))

Table 1: Total policy $P_1 = (\mathcal{A}, \mathcal{F})$

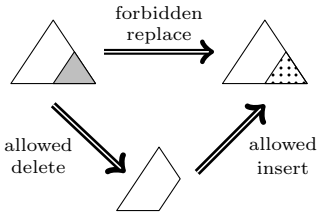


Figure 3: Insert/delete inconsistency

consists of the set of updates that insert a tree with root *patient* as a child of node *hospital*. On the other hand, the update access type ($drug$, replace($presDrug$, OTC)) represents the set of updates that replace a child of *drug* of type *presDrug* by a new node of type OTC.

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. Therefore, each update access type only makes sense for specific element types. For our example DTD, any valid XML document would contain a *placebo*, a *presDrug* or an OTC as a child node of a *drug* node, thus, an update that inserts an OTC node as a child of a *drug* node would result in a document that is not valid w.r.t the DTD. Hence, the UATs ($drug$, insert (OTC)), ($drug$,delete (OTC)) ($patient$,replace ($name$,$treatments$)) are not relevant for the hospital DTD. On the other hand, UATs ($treatments$, delete ($treatment$)) and ($name$, replace(str, str)) are relevant. We say that a UAT is *valid* w.r.t a DTD D if one of the following holds:

- $(A, \mathsf{insert}(B))$ and $A \rightarrow B^*$
- $(A, \mathsf{delete}(B))$ and $A \rightarrow B^*$
- $(A, \mathsf{replace}(B_i, B_j))$, $A \rightarrow B_1 + \cdots + B_n$, $i, j \in [1, n]$ and $i \neq j$
- $(A, \mathsf{replace}(\mathsf{str}, \mathsf{str}))$ and $A \rightarrow \mathsf{str}$

The set of valid *UATs* for a DTD D is denoted by $\mathsf{valid}(D)$.

A *security policy* P is defined by a set of *allowed* $\mathcal{A}$ and forbidden $\mathcal{F}$ valid update access types, where $\mathcal{A} \cap \mathcal{F} = \emptyset$ (no UAT can be allowed and forbidden at the same time) and $\mathcal{A} \cup \mathcal{F} = \mathsf{valid}(D)$. Intuitively, an update is allowed by a policy $P = (\mathcal{A}, \mathcal{F})$ if the update access type *uat* that represents the update is allowed in the policy, this is, if $uat \in \mathcal{A}$.

Table 1 shows the set of allowed and forbidden UATs for policy $P_1 = (\mathcal{A}, \mathcal{F})$ for the hospital DTD. The update *replace* $//drug[OTC =$ "Aspirin"] *with* <drug><placebo/></drug>)) is not allowed by P_1 since the update is represented by $\mathcal{F}_4$.

3. CONSISTENCY AND REPAIRS

3.1 Consistency

A policy defined over a DTD D is said to be *consistent* if for every XML document that conforms to D it is impossible to simulate a forbidden update through a sequence of allowed updates.

Policy $P_1 = (\mathcal{A}, \mathcal{F})$ shown in Table 1 is inconsistent for several reasons:

$\mathcal{I}_1$: Even though it is forbidden to change the value of a *name* node, child of a *patient* node ($\mathcal{F}_5$), one can delete the latter and then insert a *patient* node (allowed by $\mathcal{A}_8$ and $\mathcal{A}_7$ resp.) with a changed value for the *name* node.

$\mathcal{I}_2$: It is possible to change the value of *presDrug* node (forbidden by $\mathcal{F}_7$), by replacing the *presDrug* node with an OTC node, and then this with a *presDrug* node with a changed value (allowed by $\mathcal{A}_3$ and $\mathcal{A}_2$ resp.).

$\mathcal{I}_3$: Finally, one can replace a *placebo* with a *presDrug* node (forbidden by $\mathcal{F}_3$) by replacing a *placebo* with an OTC node (allowed by $\mathcal{A}_4$) and then this with a *presDrug* node (allowed by $\mathcal{A}_2$).

Inconsistency $\mathcal{I}_1$ is an *insert/delete inconsistency*. $\mathcal{I}_2$ and $\mathcal{I}_3$ are called *replace inconsistencies*. The first kind of inconsistencies arise when the policy *allows* one to insert **and** delete nodes of some element type A whilst *forbidding* some operation in a descendant element type of A. It is evident that in this case, the forbidden operation can be simulated by *first* deleting a node, instance of element type A and then inserting a new node, instance of A after having done the necessary modifications (see Fig. 3).

The *replace* inconsistencies arise in presence of replace operations for an element type A whose production rule is of the form $A \rightarrow B_1 + \cdots + B_n$. There are two kinds of inconsistencies: the *forbidden-transitivity* and *negative-cycle*. The former happens in the case in which 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 (inconsistency $\mathcal{I}_3$). The latter occurs when 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 (inconsistency $\mathcal{I}_2$).

To check whether a policy contains *insert/delete* inconsistencies we build the *marked* graph of the XML DTD. In this graph a node A is marked either with "+" if no operation is forbidden for any descendant element type below A or with "−" if that is not the case. Also, if for nodes A and B in the DTD, *both* $(A, \mathsf{insert}(B))$ and $(A, \mathsf{delete}(B))$ are in $\mathcal{A}$, and node A is marked with "−", then we also mark it with "⊥". This marked graph can be obtained by traversing the DTD graph starting from the nodes with out-degree 0.

Fig. 4 shows the marked DTD for policy P_1 in Table 1. The element type *hospital* is marked with both "−" and "⊥" since *(a)* one can delete and insert patients ($\mathcal{A}_8$ and $\mathcal{A}_7$) but *(b)* one cannot delete a treatment of a patient ($\mathcal{F}_1$).

To find *replace inconsistencies* we build the *replace graph* for each element type with a production rule $A \rightarrow B_1 + \ldots + B_n$. The graph contains an edge between B_i and B_j if $(A, \mathsf{replace}(B_i, B_j)) \in \mathcal{A}$. It is easy to see that node A has no replace inconsistencies if *i)* the replace graph is transitive and *ii)* none of the elements involved in cycles in the replace graph is marked with "−". The replace graph for *drug* is shown in Fig. 5 and confirms that P_1 is inconsistent since the graph is not transitive and *presDrug* is in a cycle but the update access type ($presDrug$, replace(str, str)) is forbidden ($\mathcal{F}_7$).

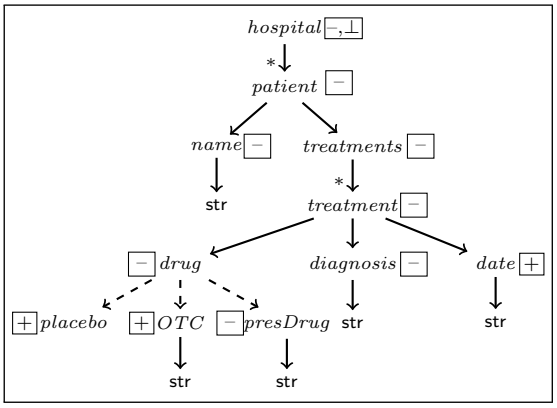


Figure 4: Marked Hospital DTD Graph

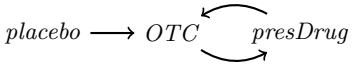


Figure 5: Replace graph $\mathcal{G}_{drug}$

It is easy to see that we can check whether a policy has no insert/delete and replace inconsistencies in PTIME [5] using standard graph algorithms.

3.2 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 a *repair* that is as close as possible to the inconsistent policy. More specifically, we want to obtain repairs by changing a minimal number of UATs from allowed to forbidden. We believe such repairs are a useful special case, since the repairs are guaranteed to be more restrictive than the original policy.

A possible repair for the policy P_1 shown in Table 1 consists in forbidding the UATs (*hospital*, delete (*patient*)) and (*drug*, replace (*OTC*, *presDrug*)).

Repair Algorithms: The problem of deciding whether it exists a repair that removes less than k UATs from the allowed operations is NP-complete. Thus, in order to find repairs, we will need to use approximation algorithms that will return a repair which is not necessarily minimal.

In this demonstration we will present the repair algorithms proposed in [5] and [6].

The algorithm to compute a repair of a policy relies on the independence between insert/delete and replace inconsistencies. 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.

In the case of *insert/delete* inconsistencies the algorithm for finding the repairs is straightforward: we iterate over all nodes in the marked DTD, and if a node A ($A \rightarrow B^*$) is marked with both "−" and "⊥" then we delete one of the (A,insert (B)) and (A, delete (B)) from the set of allowed UATs.

Finding the minimal repairs for replace inconsistencies is an NP-complete problem, and unless P = NP, there is no polynomial time algorithm to compute a minimal repair to the replace-inconsistencies. Therefore our algorithms run in polynomial time but compute a repair that is not necessarily minimal. We have developed two alternative algorithms to solve the replace inconsistencies.

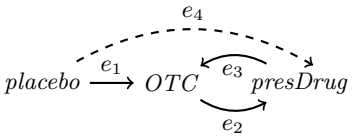


Figure 6: Transitive Replace graph $\mathcal{G}_{drug}$

The first is the *naive algorithm* that does not find a minimal repair. It takes as input the replace graph for a node A and runs a modified version of the Floyd-Warshall algorithm.

A possible execution of the naive algorithm over the policy P_1 of the hospital DTD, constructs the replace graph for element type *drug* (see Fig. 5). Then, since there are edges from *placebo* to *OTC* and from *OTC* to *presDrug* but there is no edge from *placebo* to *presDrug*, the algorithm randomly chooses one of the edges to be deleted. Say, for example, that it deletes the edge (*placebo*,*OTC*). The algorithm continues and detects that there is a cycle between *OTC* and *presDrug*, and that *presDrug* is labeled with "−" in the marked DTD graph (see Fig. 4). Thus, one of the edges is randomly chosen to be deleted, say edge (*OTC*, *presDrug*). This implies that by removing (*drug*, replace(*placebo*, *OTC*)) and (*drug*, replace(*OTC*, *presDrug*)) the policy has no replace inconsistencies. Notice that this is not a minimal repair, since deleting (*drug*, replace(*OTC*, *presDrug*)) in the first step would have solved all replace inconsistencies.

Set Cover algorithm An alternative to the naive algorithm is an algorithm based on set cover. 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 the transitive graph. 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. To avoid the potentially exponential number of justifications, the algorithm 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) [14]. The solution to this MSCP can be used to determine the set of edges to remove from $\mathcal{G}_A$ to invalidate all of the justifications of inconsistencies. 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 recomputed and the set cover run again until there are no more replace inconsistencies.

The first computation of justifications for $\mathfrak{J} = 1$ over policy P_1 and element type *drug* results in the graph in Fig. 6 where the dashed edges are the ones needed for transitivity. The justifications for edges and nodes are: $\mathcal{J}(e_4) = \{\{e_1, e_2\}\}$ and $\mathcal{J}(presDrug) = \{\{e_2, e_3\}\}$. Each justification represents violations of transitivity or negative-cycles. If we want to remove the inconsistencies, it is enough to delete one edge from each set in $\mathcal{J}$. The problem of removing one edge per justification in such a way that the total number of edges removed is minimal, 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 used to repair the policy is obtained

from the justifications. Intuitively, each element in $\mathcal{U}$ is a justification, and each set in $\mathcal{S}$ contains the justifications solved by removing a specific edge from the replace graph. The MSCP associated to the justifications for element type *drug* and policy P_1 is given in Table 2, where each column corresponds to a set in $\mathcal{S}$ and each row to an element in $\mathcal{U}$. Values 1 and 0 in the table represent membership and non-membership respectively. The table shows, for example, that if edge e_2 is deleted, then all the justifications are solved. In fact, the minimal set cover contains only the set associated to e_2, namely $\{1, 1\}$. Thus, the solution obtained from the set cover algorithm shows that the replace inconsistencies can be solved by removing (*drug*, replace(OTC, *presDrug*)) from the allowed operations of P_1.

	$\mathcal{S}$		
$\mathcal{U}$	e_1	e_2	e_3
$\{e_1, e_2\}$	1	1	0
$\{e_2, e_3\}$	0	1	1

Table 2: Set cover instance

By putting together the changes suggested to solve insert/ delete and replace inconsistencies, a possible repair is obtained by changing from allowed to forbidden the following UATs: (*hospital*, delete (*patient*)) and (*drug*, replace(OTC, *presDrug*)) from allowed to forbidden.

The set cover problem is MAXSNP-hard [14], 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 $\mathcal{U}$. In our ongoing example, the greedy approximation algorithm would return the minimal solution.

4. DEMONSTRATION

Through the demo we are going to present the problem of checking the consistency of XML write-access control policies and show how one can repair such policies to obtain consistent ones. We believe that the development and demonstration of ACCOn is a promising step towards the study of the above problems that are still in their infancy.

In ACCOn the user will be able to choose from a set of possible XML DTDs and their associated XML write-access control policies. Amongst the DTDs that we will use are DBPL, Sigmod Record Data modified to fit our framework and other DTDs for different applications. During the demonstration *i)* we will show the DTD graph that marks possible inconsistencies and *ii)* propose changes to the policy. The user will then be able to select amongst the suggested changes, and ACCOn will apply the changes to the policy and check whether consistency is achieved. In the case in which the policy is not consistent, the process will be repeated until no more inconsistencies are found.

5. RELATED WORK

Many different consistency problems have been studied in the context of relational databases and XML. Minimal repairs are also used in the problem of returning consistent answers from inconsistent databases [1]. Also, consistency of XML Schemas, i.e. the existence of an XML document that conforms to a DTD and satisfies a set of constraints has been studied in [2]. However, we are aware of no previous work on consistency and repair for XML security policies. Consistency is not an issue in security views [9] because security views consider only queries, not updates.

Our previous work [5] and [6] was the first to define and study semantic consistency properties of XML update access control policies. Since consistency checking and repair algorithms run at the schema level, scalability and heterogeneity at data level are not an immediate concern for our approach. ACCOn implements the algorithms presented in that work and allows users detect and repairing inconsistent security policies, thereby eliminating some security vulnerabilities that could lead to loss or damage to critical data.

6. REFERENCES

[1] M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *PODS*, pages 68–79, 1999.

[2] M. Arenas, W. Fan, and L. Libkin. On Verifying Consistency of XML Specifications. In *PODS*, pages 587–598, 2002.

[3] M. Benedikt, A. Bonifati, S. Flesca, and A. Vyas. Adding updates to XQuery: Semantics, optimization, and static analysis. In *XIME-P*, 2005.

[4] E. Bertino and E. Ferrari. Secure and Selective Dissemination of XML Documents. *ACM TISSEC*, 5(3):290–331, 2002.

[5] L. Bravo, J. Cheney, and I. Fundulaki. Repairing Inconsistent XML Write-Access Control Policies. In *DBPL*, pages 97–111, 2007.

[6] L. Bravo, J. Cheney, and I. Fundulaki. Repairing Inconsistent XML Write-Access Control Policies. http://arxiv.org/abs/0708.2076, August 2007.

[7] D. Chamberlin, D. Florescu, and J. Robie. XQuery Update Facility. http://www.w3.org/TR/xqupdate/, July 2006. W3C Working Draft.

[8] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A Fine-grained Access Control System for XML Documents. *ACM TISSEC*, 5(2):169–202, 2002.

[9] W. Fan, C-Y. Chan, and M. Garofalakis. Secure XML Querying with Security Views. In *ACM SIGMOD*, pages 587–598, 2004.

[10] I. Fundulaki and S. Maneth. Formalizing XML Access Control for Update Operations. In *SACMAT*, pages 169–174, 2007.

[11] I. Fundulaki and M. Marx. Specifying Access Control Policies for XML Documents with XPath. In *SACMAT*, pages 61–69, 2004.

[12] G. Kuper, F. Massacci, and N. Rassadko. Generalized XML Security Views. In *SACMAT*, pages 77–84, 2005.

[13] M. Murata, A. Tozawa, M. Kudo, and S. Hada. XML Access Control Using Static Analysis. *ACM TISSEC*, 9(3):290–331, 2006.

[14] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[15] G. Sur, J. Hammer, and J. Siméon. UpdateX - an XQuery-based language for processing updates in XML. In *PLAN-X*, 2004.

[16] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD*, pages 413–424, 2001.

[17] XUpdate - XML Update Language. *http://xmldb-org.sourceforge.net/xupdate/*.