View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Edinburgh Research Explorer

THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

Consistency and repair for XML write-access control policies

Citation for published version:
Bravo, L, Cheney, J, Fundulaki, I & Segovia, R 2012, 'Consistency and repair for XML write-access control policies' VLDB Journal, vol. 21, no. 6, pp. 843-867. DOI: 10.1007/s00778-012-0273-y

Digital Object Identifier (DOI):
10.1007/s00778-012-0273-y

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
VLDB Journal

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 05. Apr. 2019

Noname manuscript No.
(will be inserted by the editor)

Consistency and Repair for XML Write-Access Control Policies

Loreto Bravo · James Cheney · Irini Fundulaki · Ricardo Segovia

Received: date / Accepted: date

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 article investigates the problem of deciding whether a policy is consistent, and if not, how its inconsistencies can be repaired. We consider total and partial policies expressed in terms of annotated schemas defining which operations are allowed or denied for the XML trees that are instances of the schema. 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. Finally, we experimentally evaluate these algorithms in comparison with an exact approach based on answer-set programming.

1 Introduction

Valuable data such as scientific databases or electronic healthcare records are often represented using XML. Such data may include a mixture of information that should be publicly-accessible and other information that should be kept confidential. XML database security is becoming an important problem because of the increasing use of XML databases to store and manage such data. There are over 50 XQuery implementations, including a number of commercial native XML database products as well as extensions of relational databases with XML or XQuery features. However, security, and specifically fine-grained access control, have not yet been adopted widely in these products. Most of them, in contrast, support a coarse-grained model wherein users are granted or denied access to a whole document (XML tree).

L. Bravo
Universidad de Concepción, Chile
E-mail: lbravo@udec.cl

J. Cheney
University of Disburse

I. Fundulaki
ICS-FORTH, Greece

R. Segovia
Universidad de Concepción, Chile

This coarse-grained model has the same drawbacks as similar models for relational databases:

– Security policies may be implemented by the middleware or client, by code spread across many modules. This makes it difficult to understand, validate, or change the security policy, which in turn opens the door to bugs or vulnerabilities.
– Individual database users may have unnecessary capabilities to perform actions. Thus, they can accidentally or maliciously damage the integrity of the data. If any user's account is subverted, an attacker immediately gains a great deal of power over the system.

Fine-grained access control policies that are enforced by the database itself mitigate these problems, by providing a high-level, declarative description of the policy that makes it possible to assign users only necessary capabilities. Thus, a number of approaches to specifying and enforcing fine-grained access control policies for XML data (or for XML views of databases) have been developed. The main techniques explored to date include *security views*, which hide confidential data [28, 11, 18]; *accessibility maps* that annotate the XML data indicating which parts may be read by different users [31, 16]; or *static analysis* techniques that attempt to statically determine whether a requested part of the database may be read [23].

These techniques mainly address *confidentiality* — the problem of keeping sensitive information secret. This is only one part of security. Another important aspect of security is *integrity* — the problem of ensuring that sensitive data cannot be changed or corrupted accidentally or by attackers. Both confidentiality and integrity can be addressed by access control policies that check read and write operations. For XML databases, most work so far only addresses read-access policies, but there is a growing literature on XML write-access control techniques that constrain updates [20, 13, 17, 22, 15].

In this article, we study an important issue for write-access control policies for XML data: the problem of *consistency*. In general, an XML write-access control policy consists of a set of *rules* that specify 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 that is explicitly forbidden by the policy can nevertheless be

simulated by a sequence of allowed update requests. We call such write-access control policies *inconsistent*.

For highly-expressive classes of policies whose rules can be defined by arbitrary XPath expressions, consistency becomes undecidable [13], indeed, in this case it is closely related to *reachability* problems that are also undecidable, as shown by Moore [22] and (for reachability modulo a schema) by Jacquemard and Rusinowitch [15]. Such policies and rules can describe when an XML document (i.e., a tree) corresponds to a Turing machine state, and can constrain updates so that only valid transitions can be performed. If policies can also contain negative rules asserting that the machine cannot make a transition from a start state to an accept state, then a policy is consistent if and only if the corresponding Turing machine cannot halt.

To avoid this undecidability, we must consider weaker classes of policies or smaller classes of updates. In this article, we develop an approach in which policies and allowable updates are both constrained by a schema. That is, there must be a DTD-like schema constraining all XML documents under access control; the policies are defined in terms of this schema (rather than as arbitrary XPath expressions), and only update operations that are guaranteed to preserve the schema are permitted. Although these restrictions may appear draconian, in this article we provide examples which show that some common schemas and policies can fit this model. Moreover, there is a substantial payoff in doing so, in terms of complexity: consistency checking becomes a syntactic, PTIME check of the policy, rather than undecidable.

The first contribution of the article is a *nontrivial and useful class of security policies for which consistency checking is in PTIME.* To our knowledge, this is the first example of such a class for XML write-access control policies. In this context, we study both *total* and *partial policies*. For the former, each possible privilege is explicitly allowed or denied, whereas for the latter, some privileges can be inferred from others. Partial policies are a less verbose way express access control information, but this conciseness only introduces a polynomial-time increase in the complexity of consistency checking.

Simply reporting that a policy is inconsistent may not be helpful to users, especially if policies can grow large so that the ramifications of small changes are hard to predict. Thus, a natural question is how to suggest *repairs*, or updates to policies that restore consistency. There are a number of interesting possible choices for classes of repairs; we focus on what we call *least-privilege* repairs that can only remove privileges, in line with the *principle of least privilege* [26]. That is, repairs may remove privileges that lead to inconsistencies, but cannot introduce new privileges.

Thus, the second main contribution of the article is a study of the repair problem. We show that computing minimal repairs is in general NP-complete. Moreover, we develop and experimentally evaluate practical repair algorithms, including a naive greedy algorithm and a more sophisticated algorithm based on encoding repair problems as instances of the Minimum Set Cover Problem (MSCP) [9]. Our experiments show that both naive and MSCP-based repair algorithms find small repairs in practice, and do so considerably faster compared to a generic, exact approach based on the DLV answer-set programming system [19]. Moreover, as policy and schema size grow, our algorithms remain fast enough for semi-interactive use: for example, our approach runs in seconds to repair a policy over a schema with 500 elements, whereas DLV does not finish within a minute for the same problem.

+(*hospital*, insert (*patient*))	+(*hospital*, delete (*patient*))
-(*treatments*, insert (*treatment*))	-(*treatments*, delete (*treatment*))
+(*name*,replaceVal)	-(*drug*,replace (*,*))
-(*OTC*,replaceVal)	-(*presDrug*,replaceVal)
-(*diagnosis*,replaceVal)	-(*date*,replaceVal)

Table 1 Policy example for the hospital XML DTD

1.1 Example

To give a flavor of the approach to policies, consistency and repairs taken in the rest of the article, we consider a small example. Consider a typical hospital information system where patient data are stored in a database and accessed by client-side or middleware code that authenticates users and then mediates queries and updates to the data on their behalf. In the rest of the example, we write "the user updates the data" as shorthand for "the client or middleware updates the data on behalf of an authenticated user." The raw data could be stored either in a traditional RDBMS or native XML database, but the system provides an XML view according to a schema and clients can update the data through this view. Moreover, different users and user roles may have different XML security views associated with them [11], ensuring that users can only see data that they need for their work. However, we may want to grant only read-access to some of the visible data.

The XML DTD in Fig. 1 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* (over-the-counter) drug, a *diagnosis* and the *date* of a patient's visit. The XML document shown in Fig. 2 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. The policy shown in Table 1 shows a policy describing the actions that nurses are allowed (+) and not allowed (-) to perform.

We can use a variant of the policy language XAcU [13] to provide fine-grained write-access control over the hospital

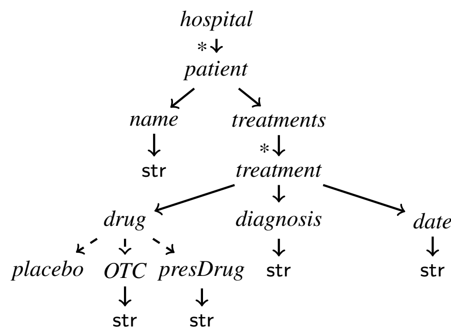

Fig. 1 Hospital DTD Graph

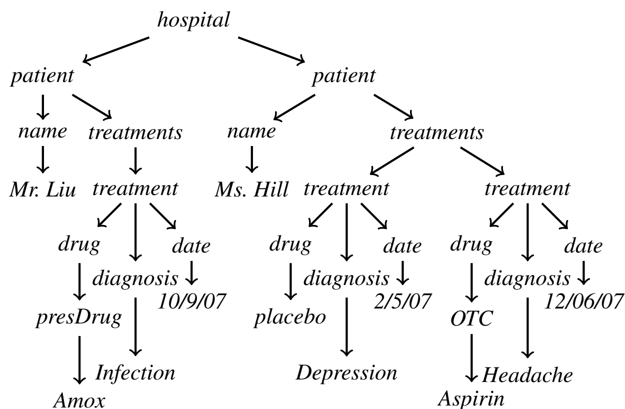

Fig. 2 Hospital XML document

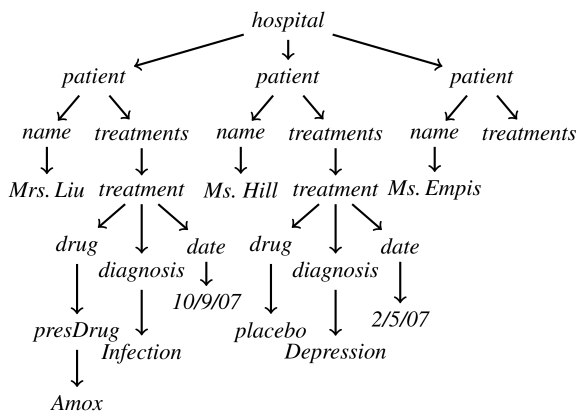

Fig. 3 Updated XML document

data, enforced by the database, to complement any security checks performed by the client or middleware. In this naive policy we allow a nurse to perform *insert*, *delete* and *replace* update actions. More specifically, a nurse is allowed to *insert* and *delete* patients (rules +(*hospital*, insert (*patient*)) and +(*hospital*, delete (*patient*))) but she cannot *modify* a patient's diagnosis or change a prescription drug to an off-the-counter drug (rule -(*drug*,replace (*,*))). Note that here (*drug*,replace (*,*)) is used as a shorthand for all possible replacements of one drug with another. However, it is easy to see that the diagnosis of a patient can still 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*. Given an inconsistent policy, it is a natural question to ask how the policy can be repaired to restore consistency. Obviously, any inconsistent policy can be repaired by removing all of its positive or negative rules (or simply all of its rules). However, we expect security administrators may prefer to select among smaller repairs, and select among them based on domain knowledge. For example, the above policy may be repaired by taking away either rule +(*hospital*, insert (*patient*) or +(*hospital*, delete (*patient*)). It seems sensible to allow nurses to insert patient records and change some information such as names, while forbidding nurses to delete patient records. Instead, a record deletion should be done by an administrator, and a change to the treatment information should be done by a doctor.

In addition to *consistency*, an important property that should be considered when defining a policy language is *succinctness*. We identify here two forms of succinctness: *syntactic* and *semantic*. In the first case, we can define *macros* that can be expanded (e.g., this is the case for rule -(*drug*, replace (*,*))) to all matching privileges. This kind of syntactic shorthand can be simulated by performing a sequence of actions justified by other privileges. For the second case, we believe it is useful to allow policies to be *partial*, that is, to omit privileges that are implied by others. For example, in the original policy above, it is redundant to state that nurses can replace patient names, since they can already just delete the patient record and replace it with a new record with an updated name. In a partial policy, we may omit this rule.

1.2 Outline

As described above, this article makes the following contributions:

1. We define consistency and identify a useful class of security policies for which consistency checking is in PTIME; and we extend these results to partial policies, from which total policies can be recovered in PTIME.
2. We study the problem of repairing consistent policies by removing privileges and show that it is NP-complete; and we give exact and approximation algorithms for finding repairs.
3. We present an implemented tool for policy consistency checking and repair called ACCon, and we experimentally evaluate the consistency and repair algorithms.

Preliminary versions of this work have appeared previously in a conference paper [5] and a demonstration of ACCon [6]. This article incorporates revised and extended material from

those papers. Specifically, compared to [5], this article includes full proofs and extends the schema language from the relatively restricted case of structured DTDs [11] to a larger class of XML schemas. We consider CEDTDs, or EDTDs based on Chain Regular Expressions [3]. While CEDTDs are not as expressive as full XML Schemas, many schemas encountered in practice only use chain regular expressions. Compared to [6], we present a more mature tool and a systematic experimental evaluation, including comparison with an encoding of the repair problem using answer-set programming in DLV.

The structure of the rest of the article is as follows. In Section 2 we review background concerning XML trees and DTDs. Section 3 we introduce our version of the XML Access Control for Updates (XAcU) policy formalism [13]. In Section 4 we define consistency and give polynomial time algorithms for checking consistency of total and partial policies. In Section 5 we define the policy repair problem, prove its NP-completeness and give algorithms for repairing policies. In Section 6 we present an experimental evaluation based on the ACCon implementation. Section 7 discusses additional related work and Section 8 concludes.

Note to practitioners: From the point of view of implementation, the key sections of this article are Section 3 (defining security policies), Sections 4.1 and 4.3 giving definitions and algorithms for consistency. Section 4.2 gives details of the proof of the main theoretical result, and can be skipped by readers only interested in implementation. The repair algorithms in Section 5 and their experimental evaluation in Section 6 can also be appreciated without reading the accompanying proofs.

2 Background

2.1 Trees

We model XML documents as *rooted unordered* trees with labels from $\mathcal{L} \cup \{\mathsf{str}\}$, where L is the infinite domain of labels.

Definition 1 (XML Tree) An unordered XML tree t is a structure of the form $t = (N_t, E_t, \lambda_t, r_t, v_t)$ where:

1. (N_t, E_t) is a tree in the usual sense with root r_t: N_t is the set of nodes, $E_t \subset N_t \times N_t$ is the set of edges, and there is exactly one path from r_t to each node of N_t;
2. $\lambda_t : N_t \to \mathcal{L} \cup \{\mathsf{str}\}$ is a labeling function over nodes, such that $\lambda(n) = \mathsf{str}$ implies n is a leaf; and
3. v_t is a function that assigns string values to leaves 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.

Trees have the standard notion of isomorphism; two isomorphic trees have the same structure, even if they differ in the choice of vertex names used.

Definition 2 (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 $h : N_{t_1} \to N_{t_2}$ where:

1. $h(r_{t_1}) = r_{t_2}$,
2. if $(x, y) \in E_{t_1}$ then $(h(x), h(y)) \in E_{t_2}$,
3. $\lambda_{t_1}(x) = \lambda_{t_2}(h(x))$, and
4. $v_{t_1}(x) = v_{t_2}(h(x))$ for every x with $\lambda_{t_1}(x) = \mathsf{str} = \lambda_{t_2}(h(x))$.

2.2 Atomic Updates

Our updates are modeled according to the XQuery Update Facility 1.0 Recommendation [25]. The syntax of atomic updates is as follows:

$$op ::= \mathsf{delete}(n) \mid \mathsf{insert}(n,t) \mid \mathsf{replace}(n,t) \mid \mathsf{replaceVal}(n,s)$$

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 $\mathsf{replaceVal}(n,s)$ operation will replace the text value of node n with string s. An $\mathsf{insert}(n,t)$ operation will insert a tree t as a child node below n. In the standard, there are several additional types of insert operations, allowing to insert nodes before, after, or into the content of other nodes, but since we consider unordered XML trees these distinctions are unnecessary.

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 defined formally as shown in Table 2. We denote by $[\![op]\!](t)$ the result of applying update operation op on tree t.

Consider for instance the atomic operations $\mathsf{insert}(n_1, t_1)$ and $\mathsf{delete}(n_2)$ where n_1 is the *hospital* node and n_2 is the last, in document order, *treatment* node of the XML document shown in Figure 2. The inserted subtree t_1 and the deleted subtree t_2 are shown in Figure 4. The result of applying these atomic operations on the XML document of Figure 2, is the updated XML document shown in Figure 3.

2.3 Schemas

There are two main schema languages for XML: Document Type Definition (DTD) and XML schema (XSD). Since we want to deal with both types of schemas, we need a common abstraction for both. In [21], it is shown that *Extended DTDs (EDTDs)* can be used as such an abstraction. For the

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

Table 2 Semantics of update operations

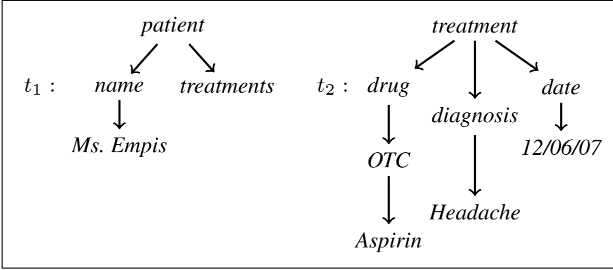

Fig. 4 XML trees t_1 and t_2

purposes of this article, we use the generic term *schema* to refer to an arbitrary Extended DTD. In this article, we consider nonrecursive schemas only.

Definition 3 (EDTD) An *extended DTD (EDTD) D* is represented by $(Ele, Types, Rg, rt, \mu)$ where

1. Ele is a finite set of *element names*,
2. $Types$ is a finite set of *element types*,
3. rt is a distinguished element name in Ele and in $Types$ called the *root*,
4. μ is a mapping from $Types$ to Ele such that $\mu(rt) = rt$, and
5. Rg defines the element types: that is, $Rg : Types \rightarrow Reg_{Types}$, where Reg_{Types} is the set of regular expressions r over $Types$, defined using the grammar:

$$r ::= \mathsf{str} \mid \epsilon \mid B \mid r_1, \ldots, r_n \mid r_1 + \ldots + r_n \mid r^*$$

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

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 node labeled with an element name A in an EDTD D is called an *instance* of A. An XML tree $t = (N_t, E_t, \lambda_t, r_t, v_t)$ *conforms* to a EDTD $D = (Ele, Types, Rg, rt, \mu)$ at element type A if there is a function $\eta : N_t \rightarrow Types \cup \{\mathsf{str}\}$ such that:

1. r_t is labeled with the name associated to type A (i.e., $\eta(r_t) = A$)
2. if $\eta(n) \in Types$ then the label associated with $\eta(n)$ is the label of n ($\mu(\eta(n)) = \lambda_t(n)$) and the labels of the children of n match the content regular expression associated with type $\eta(n)$ ($\eta[\mathsf{children}_t(n)] \in L(Rg(\eta(n)))$).
3. if $\eta(n) = \mathsf{str}$ then $\lambda_t(n) = \mathsf{str}$ and $v_t(n)$ is defined and $\mathsf{children}_t(n)$ is empty.

An XML tree t is a valid instance of the EDTD D if it conforms to D at element type rt. We write $I_D(A)$ for the set of valid instances of D at element type A, and I_D for $I_D(rt)$.

Note that even though we are not including regular expressions of the form $r^?$ and r^+ which are common in DTDs, we can express them using $(r+\epsilon)$ and (r, r^*) respectively. In XSDs, it is possible to express cardinality restrictions using minOccurs and maxOccurs which are not expressible using EDTD. However, for the purposes of this article, we treat elements with cardinality restrictions the same as Kleene star; the more specific information about cardinality is not relevant to our access control policies. Anonymous types in XSDs can be easily handled by EDTD by adding a type with a fresh name.

The distinction in EDTDs between element names and types is needed only to represent XSDs. In the case of DTDs, there is only one type associated for each name, i.e. $Ele = Types$ and μ is the identity. Thus, a DTD can be expressed simply by (Ele, Rg, rt).

In this article, we will often restrict attention to EDTDs that are unambiguous in the sense that for every tree $t \in I_D$ there is exactly one mapping η showing how t validates in D. This property is important because we will use type names in security policies, and ambiguity complicates the meaning of such policies significantly. Both ordinary DTDs and XML Schemas are constrained to be unambiguous, and [21] investigate several other classes of EDTDs that are unambiguous, including single-type and 1-pass preorder typing. They argue that these restrictions are more liberal than the ones employed in the DTD or XML Schema standards.

In any case, we simply assume unambiguous EDTDs, leaving the question of how this requirement is enforced as a separate issue that is already largely addressed by prior work.

2.3.1 Structured Regular Expressions and SEDTDs

In [11] and a number of other papers, the authors study a special type of DTDs that they call *structured* DTDs. Although not all DTDs are syntactically representable in this form, one can (as argued in [11]) represent general DTDs by introducing new element types.

Here, we would like to do the same for EDTD. This simplification will allow us to analyze the problems of access control in a simpler setting, and then generalize.

Definition 4 (SEDTD) Let $\mathcal{L}$ be the infinite domain of labels. A *structured EDTD (SEDTD) D* is an EDTD such that $Rg : \mathit{Types} \rightarrow \mathit{SReg}_{\mathit{Types}}$, where $\mathit{SReg}_{\mathit{Types}}$ is the set of simple regular expressions r over Types, defined using the grammar:

$$r ::= \mathsf{str} \mid \epsilon \mid B_1, \ldots, B_n \mid B_1 + \cdots + B_n \mid B_1*$$

where we further require that in a sequential composition $B_1, \ldots, B_n$ or sum $B_1 + \cdots + B_n$, the type names B_i are distinct (that is, $B_i = B_j$ implies $i = j$.)

The relationship between the element types of a SEDTD can be represented using a directed graph that we call a *SEDTD graph.*

Definition 5 (SEDTD Graph) A SEDTD graph for a SEDTD $D = (\mathit{Ele}, \mathit{Types}, Rg, rt, \mu)$ is a directed graph $G_D = (\mathcal{V}_D, \mathcal{E}_D, r_D, Op_D)$ where

1. $\mathcal{V}_D$ is the set of nodes for the element types in $\mathit{Types} \cup \{\mathsf{str}\}$,
2. $\mathcal{E}_D = \{(A, B) \mid A, B \in \mathit{Types}$ and B is a subelement type of $A\}$
3. r_D is the distinguished node rt
4. For each A, $Op(A)$ is the regular expression operation associated with A (that is, either $\mathsf{str}, \epsilon, +, *$, or sequential composition ','.

We depict SEDTD graphs using the convention that dotted lines indicate a content type of the form $B_1 + \cdots + B_n$, an asterisk on an edge indicates a content type of the form B^*, and no decoration indicates a content model of the form $B_1, \ldots, B_n$. Also, it should be clear that we can convert a graph back to the original SEDTD, although we do not make use of this connection explicitly, other than to help visualize DTD graphs and policies.

Example 1 The EDTD graph presented in Fig. 5 corresponds to the SEDTD $(\mathit{Ele}, \mathit{Types}, Rg, rt, \mu)$, where $\mathit{Ele} = \mathit{Types} = \{A, B, \ldots, K\}$, $rt = R$ and μ is the identity function for all element types except $\mu(G) = L$ and $\mu(F) = L$. The production rules are:

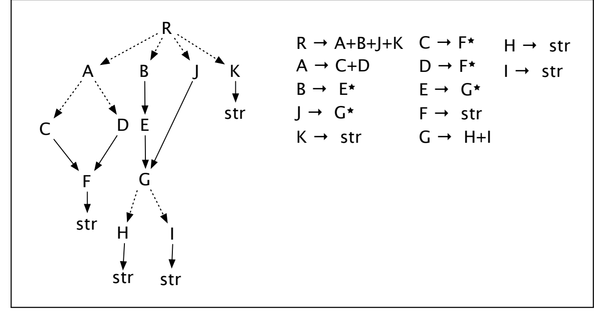

Fig. 5 Example DTD

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

2.3.2 Chain Regular Expressions and CEDTDs

Besides structured schemas, we consider schemas whose regular expressions belong to a restricted class called *chain regular expressions* (or CHAREs). These were introduced in work on *inferring* DTDs or schemas from example data, where it was argued that over 90% of regular expressions encountered in practical schemas are chain regular expressions [3].

Essentially, a chain RE is a sequence of factors, each of which is of the form $(A_1 + \cdots + A_n)$ or $(A_1 + \cdots + A_n)^q$, where q is one of ?, +, or $*$. Recall that we consider only the Kleene star operation as primitive; thus, for unordered data, chain REs can be further simplified since $(A_1 + \cdots + A_n)^* \equiv A_1^*, \ldots, A_n^*$.

Definition 6 An unordered chain regular expression (CHARE) is an expression of the form r where:

$$f ::= (A_1 + \cdots + A_n) \mid A^*$$
$$r ::= f_1, \ldots, f_n \mid \mathsf{str}$$

That is, a CHARE is either an atomic type str or a sequence of sum factors $(A_1 + \cdots + A_n)$, or starred single elements. Moreover, we require that types be single-occurrence: no type name is reused. Note that a sum factor may have just one alternative; thus, $A, (B + C), D^*, E, F^*, (G + H)$ is a CHARE according to our definition. Note also that repeated element names within a sum term are redundant. Other regular expression operators such as $A^?$ (optional A) and A^+ (sequence of one or more As) could also be included. For the purposes of access control they can be handled the same as Kleene star A^*, by allowing only insertion and deletion operations; the additional cardinality constraint on the number of As is irrelevant to access control.

We restrict attention to regular expressions that use each type name at most once. We say that a *chain EDTD*, or CEDTD is an EDTD in which every regular expression used

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="customer">
  <xs:complexType>
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="caCustomer" type="caCustomer"/>
    <xs:element name="usCustomer" type="usCustomer"/>
   </xs:choice>
  </xs:complexType>
  </xs:element>
  <xs:complexType name="caCustomer">
   <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="address" type="caAddress"/>
   </xs:sequence>
  </xs:complexType>
  <xs:complexType name="usCustomer">
   <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="address" type="usAddress"/>
   </xs:sequence>
 </xs:complexType>
 <xs:complexType name="caAddress">
  <xs:sequence>
   <xs:element name="street" type="xs:string"/>
   <xs:element name="city" type="xs:string"/>
   <xs:element name="province" type="xs:string"/>
   <xs:element name="postalCode" type="xs:string"/>
   <xs:element name="country" type="xs:string"
            fixed="Canada"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="usAddress">
  <xs:sequence>
   <xs:element name="street" type="xs:string"/>
   <xs:element name="city" type="xs:string"/>
   <xs:element name="state" type="xs:string"/>
   <xs:element name="zip" type="xs:decimal"/>
   <xs:element name="country" type="xs:string"
            fixed="US"/>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

Fig. 6 Canadian and US Customer XSD

in a rule is a CHARE. Every SEDTD is a CEDTD, so in the sequel results proved for CEDTDs will automatically apply to SEDTDs as well.

Example 2 (Relating XSDs and CEDTDs) In this article, we use SEDTDs and CEDTDs, which include features not present in DTDs; on the other hand, XML Schemas include many features that are not supported by CEDTDs. Nevertheless, we can easily we can easily check whether a general XML Schema defines a CEDTD, and many schemas do fit this restriction. For example, the XSD in Figure 6 can be represented by the CEDTD $D = (Ele, \mathit{Types}, Rg, rt, \mu)$ with

$$\begin{aligned}
Ele &= \{\mathit{customer}, \mathit{caCustomer}, \mathit{usCustomer}, \mathit{name}, \mathit{address}\}\\
\mathit{Types} &= \{\mathit{customer}, \mathit{caCustomer}, \mathit{usCustomer}, \mathit{name},\\
&\quad \mathit{caAddress}, \mathit{usAddress}\}\\
rt &= \mathit{customer}\\
\mu(\mathit{caAddress}) &= \mathit{address}\\
\mu(\mathit{usAddress}) &= \mathit{address}
\end{aligned}$$

and Rg is such that:

$$\begin{array}{ll}
\mathit{customer} \to (\mathit{caCustomer}{+}\mathit{usCustomer})^* & \mathit{street} \to \mathsf{str}\\
\mathit{caCustomer} \to (\mathit{name},\mathit{caAddress}) & \mathit{state} \to \mathsf{str}\\
\mathit{usCustomer} \to (\mathit{name},\mathit{usAddress}) & \mathit{province} \to \mathsf{str}\\
\mathit{caAddress} \to (\mathit{street},\mathit{city},\mathit{province},\mathit{postalCode}) & \mathit{zip} \to \mathsf{str}\\
\mathit{usAddress} \to (\mathit{street},\mathit{city},\mathit{state},\mathit{zip}) & \mathit{postalCode} \to \mathsf{str}\\
\mathit{name} \to \mathsf{str} &
\end{array}$$

Note that elements of type *caAddress* and *usAddress* are associated to the element name *address*. ■

3 XML Access Control Framework

3.1 Policies

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

Definition 7 (Update Access Types) Given a SEDTD D, an *update type* (*UT*) is an expression of the form:

$$ut ::= \mathsf{insert}(B) \mid \mathsf{delete}(B) \mid \mathsf{replace}(A, B) \mid \mathsf{replaceVal}$$

where A, B are types from D. An *update access type* (*UAT*) defined over D is an expression of the form (A, ut) where A is an element type in D and ut is an update type.

Intuitively, an *UAT* represents a set of atomic update operations. More specifically, given an instance t of SEDTD D, an atomic update *op*, and an update access type uat we say that *op matches* uat on t (*op* $\mathsf{matches}_t$ uat) if:

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

$$\frac{\eta(n) = B \quad \eta(\mathsf{parent}_t(n)) = A}{\mathsf{delete}(n) \;\; \mathsf{matches}_t \;\; (A, \mathsf{delete}(B))}$$

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

$$\frac{\eta(n) = \mathsf{str} \quad \eta(\mathsf{parent}_t(n)) = A}{\mathsf{replaceVal}(n, s) \;\; \mathsf{matches}_t \;\; (A, \mathsf{replaceVal})}$$

Here, η is the unique mapping from each node in t to the element type associated to it (uniqueness follows from the assumption that D is unambiguous).

It is trivial to translate our update access types to XAcU^{annot} security annotations. In this work we require that the evaluation of an update operation on a tree that conforms to a schema 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 the example DTD in Figure 5, 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. Similarly, $(R, \mathsf{delete}(A))$ or $(R, \mathsf{insert}(A))$ are invalid since there has to be exactly one node (labeled A, B, J or K) under R. On the other hand, $(B, \mathsf{delete}(E))$ and $(B, \mathsf{insert}(E))$ are appropriate for this specific DTD. Finally, we exclude *reflexive replacement* update access types of the form $(A, \mathsf{replace}(B_i, B_i))$ from consideration. These would be schema-preserving when the content of A matches either B_i^* or $B_1 + \cdots + B_n$, but they do not actually provide any expressive power that cannot be simulated by other UATs. Moreover, as discussed further in Section 4, reflexive replacement UATs are undesirable from the point of view of consistency analysis.

The relation uat valid_in D, which indicates that an update access type uat is valid for the DTD D, is defined as follows:

$$\frac{}{\mathsf{insert}(B) \ \mathsf{valid_in}\ B^*} \qquad \frac{}{\mathsf{delete}(B)\ \mathsf{valid_in}\ B^*}$$

$$\frac{}{\mathsf{replaceVal\ valid_in\ str}}$$

$$\frac{i, j \in [1, n] \quad i \neq j}{\mathsf{replace}(B_i, B_j)\ \mathsf{valid_in}\ B_1 + \cdots + B_n}$$

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

We define the set of valid *UATs* for a given CEDTD D as $\mathsf{valid}(D) = \{uat \mid uat\ \mathsf{valid_in}\ D\}$. We define a *valid run* as a sequence $t_0 \xrightarrow{op_1} \cdots \xrightarrow{op_n} t_n$ such that:

1. each t_i is valid with respect to D, and
2. for each $i \in \{1, \ldots, n\}$, we have $t_i = [\![op]\!](t_{i-1})$.

A *security policy* will be defined by a set of *allowed* and *forbidden* valid *UATs*.

Definition 8 A security policy P defined over a CEDTD 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 3 Consider the EDTD D in Fig. 5 and the policy $P = (\mathcal{A}, \mathcal{F})$ where $\mathcal{A}$ is:

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

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

The set of operations that are allowed by a policy $P = (\mathcal{A}, \mathcal{F})$ on an XML tree t, denoted by $[\![\mathcal{A}]\!](t, \eta)$, is 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 a valid run $t_0 \xrightarrow{op_1} t_1 \ldots \xrightarrow{op_n} t_n$ is *allowed* if for every $i \in \{1, \ldots, n\}$, we have $op_i \in [\![\mathcal{A}]\!](t_{i-1})$. 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 meaning of partial policies is studied in detail in Section 4.3.

Note that we do not allow replacements or deletes that target the root of the tree. Indeed, replacing or deleting the root of an XML tree does not make sense because the resulting (empty) tree will not match the DTD. Replacements and deletes targeting the root are examples of updates that are not covered by any UAT; thus, they cannot be allowed or forbidden by a policy. Instead, if we want a user to be able to do anything to the tree we can allow this using a policy $(\mathsf{valid}(D), \emptyset)$ that allows all of the valid UATs. (It is easy to see that this suffices to allow any tree to be updated to any other tree.)

4 Consistency

4.1 Total 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 9 Consider a policy $P = (\mathcal{A}, \mathcal{F})$ defined over schema D. An *inconsistency* in P consists of an allowed run $t_0 \xrightarrow{op_1} t_1 \ldots \xrightarrow{op_n} t_n$, and an update $op_0 \in [\![\mathcal{F}]\!](t_0)$ such that $t_n = [\![op_0]\!](t_0)$. Conversely, P is *consistent* if there are no inconsistencies.

Consistency is highly sensitive to the design of policies and update types. For example, we have consciously chosen to *omit* reflexive replace update types of the form $(A, \mathsf{replace}(B_i, B_i))$ for an element type in the schema whose production rule is either of the form $B*$ or $B_1 + \ldots + B_n$. To see why, consider 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 UAT 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 types $\mathsf{replace}(B_i, B_i)$ 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*.

Inconsistencies can be classified into two forms: insert/delete and replace:

– Inconsistencies due to insert and 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 the node. 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 $A \to B_1 + \cdots + B_n$ of an unambiguous schema. 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 composing the first two. Second, suppose 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.

In addition, it is worth pointing out that insert/delete and replace inconsistencies are *independent* in a certain sense: that is, the presence or absence of an insert/delete inconsistency does not affect that of a replace inconsistency. This is due partly to the limits we place on the schema language, since each node in a valid document can only be updated by insert/delete operations or by replace operations, never both. This independence is important in the development of repair algorithms in the next section, since it means that we can repair the insert/delete inconsistencies and replace inconsistencies separately.

We say that *nothing is forbidden below* A in a policy $P = (\mathcal{A}, \mathcal{F})$ defined over D if for every B_i s.t. $A \leq_D B_i$, we have $(B_i, op) \notin \mathcal{F}$ for every $(B_i, op) \in \mathsf{valid}(D)$. If $A \to B_1 + \ldots + B_n$, then we define the *replace graph* $\mathcal{G}_A = (\mathcal{V}_A, \mathcal{E}_A)$ where:

1. $\mathcal{V}_A$ is the set of nodes for $B_1, B_2, \ldots B_n$, and
2. $(B_i, B_j) \in \mathcal{V}_A$ if there exists $(A, \mathsf{replace}(B_i, B_j)) \in \mathcal{A}$.

Also, the set of *forbidden edges* of A is $\mathcal{E}_A = \{(B_i, B_j) \mid (A, \mathsf{replace}(B_i, B_j)) \in \mathcal{F}\}$. 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 closure of $\mathcal{G}_A$.

Definition 10 We say that a policy $P = (\mathcal{A}, \mathcal{F})$ defined over CEDTD D is *syntactically consistent* if and only if for every type A in D:

1. If $B^* \in Rg_D(A)$ and $(A, \mathsf{insert}(B)), (A, \mathsf{delete}(B)) \in \mathcal{A}$, then nothing is forbidden below B, and
2. If $(B_1 + \cdots + B_n) \in Rg_D(A)$, then:
   (a) for every $i \neq j \in [1..n]$, if $(B_i, B_j) \in \mathcal{G}_A^+$, then $(B_i, B_j) \notin \mathcal{F}_A$, and
   (b) 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.

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

Our main result is that syntactic consistency is the same as consistency:

Theorem 1 *A policy $P = (\mathcal{A}, \mathcal{F})$ defined over CEDTD D is consistent if and only if it is syntactically consistent.*

Proof The reverse direction is proved in detail in Section 4.2. For the forward direction, we prove the contrapositive. Suppose P is not syntactically consistent. If property (1) is violated, then there is obviously an insert/delete inconsistency since we can simulate a forbidden UAT by deleting and inserting. If property (2a) is violated then we can simulate a forbidden replace by performing a sequence of allowed replacements. Finally, if property (2b) is violated then we can simulate a forbidden UAT by performing a chain of replacements (similar to the case for property (1)). In each case, P is inconsistent. □

Example 4 (example 3 continued) The total policy P is inconsistent. Some of the inconsistencies that can be found are:

– $(E, \mathsf{insert}(G))$ and $(E, \mathsf{delete}(G))$ are in $\mathcal{A}$, but there is a forbidden operation below G since $(G, \mathsf{replace}(H, I)) \in \mathcal{F}$ (by condition 1, Theorem 1),
– $(R, \mathsf{replace}(A, J)), (R, \mathsf{replace}(A, K))$ are in $\mathcal{A}$, but there is a forbidden replace $(R, \mathsf{replace}(B, K))$ in $\mathcal{F}$ (condition 2, Theorem 1), and
– There are cycles in $\mathcal{G}_R$ involving both B and J, but below each 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 a policy is syntactically consistent using standard polynomial-time graph algorithms to check properties (1), (2a) and (2b).

Proposition 1 *The problem of deciding policy consistency with respect to CEDTDs is in* PTIME.

Proof By Theorem 1, there are two cases in which a policy can be inconsistent. The first case can be checked by traversing the graph following a topological sorting of the DTD graph. This can be done in time polynomial in the number of edges and vertices of the DTD graph.

The second case consists of checking if the graphs $\mathcal{G}_A$ are acyclic and transitive. Checking these two conditions for each element A can be done in polynomial time. □

4.2 Proof of Theorem 1

In this section we present a detailed proof of Theorem 1. We first show the main result for SEDTDs, and then show that the characterization can be lifted to CEDTDs. The proof for SEDTDs requires considering many combinations of cases. The main difficulty is in proving that syntactic consistency implies consistency, that is, there is no way to simulate a single forbidden operation via a sequence of allowed operations. The obvious approach by induction on the length of the allowed sequence does not work because subsequences of the allowed sequence do not necessarily continue to simulate the denied operation.

The solution is to establish the existence of an appropriate *normal form* for update sequences, such that (roughly speaking):

1. The normal form of an arbitrary update sequence $\overline{op}$ applied to input t is

   $\mathsf{delete}(n_1); \cdots ; \mathsf{delete}(n_i);$
   $\overline{r};$
   $\mathsf{insert}(l_1, t_1), \ldots, \mathsf{insert}(l_j, t_j)$

   consisting of a sequence of deletes, then replacements, then inserts
2. The replacements $\overline{r}$ can be partitioned into chained subsequences $\overline{r_1}, \ldots, \overline{r_j}$ of the form

   $\overline{r_i} = \mathsf{replace}(m_i, u_1^i); \mathsf{replace}(r_{u_1^i}, u_2^i); \cdots .$
3. Each n_i, m_j, l_k is in t.
4. No deleted or replaced node (n_i or m_j) is an ancestor of another of the modified nodes (n_i, m_j, l_k)
5. Allowed update sequences have allowed normal forms.

Pictorially, a normalized update sequence can be visualized as a tree with some of its nodes annotated with insertion operations $\mathsf{insert}(u)$, deletions delete, and replacement sequences $\mathsf{replace}(u_1, \ldots, u_n)$, such that no annotation occurs below a node with a delete or replace annotation.

Normalized update sequences are much easier to analyze than arbitrary allowed sequences in the proof of the reverse direction of Theorem 1.

We introduce some additional helpful notation: write

$$target(\mathsf{delete}(n)) = n$$
$$target(\mathsf{insert}(n, u)) = n$$
$$target(\mathsf{replace}(n, u)) = n$$

for the target node of an operation; write $\leq_t$ for the ancestor-descendant ordering on t (that is, E^*); write $\perp_t$ for the relation $\{(n, m) \in N_t \times N_t \mid n \not\leq_t m \text{ and } m \not\leq_t n\}$ (that is, $n \perp_t m$ means n and m are $\leq_t$-incomparable). We write $\overline{op} \equiv \overline{op'}$ to indicate that the (partial) functions $[\![\overline{op}]\!](-)$ and $[\![\overline{op'}]\!](-)$ are equal; that is, for any tree t, op is valid on t if and only if op' is valid on t, and if both are valid, then $[\![\overline{op}]\!](t) = [\![\overline{op'}]\!](t)$.

Lemma 1 *The laws in Figures 7, 8, and 9 are valid for rewriting update sequences relative to a given input tree t. Moreover, if one of the rules is applied to an update sequence that is allowed by a policy P, then the resulting update sequence is still allowed.*

Proof Proof is by case analysis. For many cases, to ensure that the rewrites preserve validity and allowedness we need the fact that updates only affect the types of nodes under the target node. This is easy to show for deterministic CEDTDs.

Note that most of the identities only rearrange existing allowed updates and do not introduce any new update operations that we need to check against the policy. In a few cases, we need to do some work to check that the rewritten sequence is still allowed. In some cases, the rewrite rules cover cases that cannot happen in allowed sequences: for example, $\mathsf{replace}(n, u); \mathsf{delete}(m)$ to $\mathsf{delete}(n)$ with $m = r_u$ cannot happen in an allowed sequence. □

Proposition 2 *Let P be a security policy and $\overline{op}$ an allowed update sequence mapping t to t'. Then there is an equivalent allowed update sequence $\overline{op'}$ that is in normal form.*

Proof We can use the identities to normalize an update sequence as follows. First, move occurrences of inserts to the end of the sequence. Next, move deletes to the beginning of the sequence. Finally, we use the remaining rules to eliminate dependencies among deletes, replacements and inserts, and to build chains of replacements. The resulting sequence is in normal form.

We say that two trees *agree above n* if the trees are equal after deleting the subtree rooted at n from each. Note that for all of the operations we consider, if op has target node n and op is valid on t then t agrees with $[\![op]\!](t)$ above n.

Lemma 2 *Consider a policy P with respect to a SEDTD D. If t and t' conforming to D are equal above n, and P allows a sequence $\overline{op}$ that maps t to t', then there is an equivalent, normalized, allowed sequence $\overline{op'}$ that only affects nodes at, above, or below n.*

Proof We show that for each node m unrelated to n, updates applying directly to m can be eliminated. If a deletion applies to m, then there must be an insertion replacing the deleted subtree exactly, and these are the only updates affecting m. Thus, it is safe to remove this useless deletion-insertion pair. If a replacement applies to m, then there must be subsequent replacements that restore the subtree at m. This sequence of replacements can be eliminated. No other possibilities are consistent with t and t' being equal above n. Thus, by considering each node m in the tree that is unrelated to n, and removing the updates having an effect on m, we can obtain an equivalent update sequence $\overline{op'}$ having only updates whose target node is related to n. This update

$$\mathsf{insert}(n,u);\mathsf{insert}(m,v) \equiv \begin{cases} \mathsf{insert}(n, [\![\mathsf{insert}(m,v)]\!](u)) & \text{if } m \in N_u \\ \mathsf{insert}(m,v);\mathsf{insert}(n,u) & \text{if } m \notin N_u \end{cases}$$

$$\mathsf{insert}(n,u);\mathsf{replace}(m,v) \equiv \begin{cases} \mathsf{replace}(m,v) & \text{if } m \leq_t n \\ \mathsf{replace}(m,v);\mathsf{insert}(n,u) & \text{if } m \in N_t, m \not\leq_t n \\ \mathsf{insert}(n,v) & \text{if } m = r_u \\ \mathsf{insert}(n, [\![\mathsf{replace}(m,v)]\!](u)) & \text{if } m \in N_u - \{r_u\} \end{cases}$$

$$\mathsf{insert}(n,u);\mathsf{delete}(m) \equiv \begin{cases} \mathsf{delete}(m) & \text{if } m \leq_t n \\ \mathsf{delete}(m);\mathsf{insert}(n,u) & \text{if } m \in N_t, m \not\leq_t n \\ \epsilon & \text{if } m = r_u \\ \mathsf{insert}(n, [\![\mathsf{delete}(m)]\!](u)) & \text{if } m \in N_u - \{r_u\} \end{cases}$$

Fig. 7 Moving inserts forward

$$\mathsf{replace}(n,u);\mathsf{delete}(m) \equiv \begin{cases} \mathsf{delete}(m) & \text{if } m \leq_t n \\ \mathsf{delete}(m);\mathsf{replace}(n,u) & \text{if } m \in N_t, m \not\leq_t n \\ \mathsf{delete}(n) & \text{if } m = r_u \\ \mathsf{replace}(n, [\![\mathsf{delete}(m)]\!](u)) & \text{if } m \in N_u - \{r_u\} \end{cases}$$

$$\mathsf{delete}(n);\mathsf{delete}(m) \equiv \begin{cases} \mathsf{delete}(m) & \text{if } m \leq_t n \\ \mathsf{delete}(m);\mathsf{delete}(n) & \text{if } m \not\leq_t n \end{cases}$$

Fig. 8 Moving deletes backward

$$\mathsf{replace}(n,u);\mathsf{replace}(m,v) \equiv \begin{cases} \mathsf{replace}(m,v) & \text{if } m \leq_t n \\ \mathsf{replace}(m,v);\mathsf{replace}(n,u) & \text{if } m \in N_t, m \not\leq_t n \\ \mathsf{replace}(n, [\![\mathsf{replace}(m,v)]\!](u)) & \text{if } m \in N_u - \{r_u\} \end{cases}$$

Fig. 9 Chaining and commuting replacements

sequence is still allowed since we have only removed allowed operations (and since all of the operations we have removed are independent of the remaining ones), and is still in normal form. □

If t, t' agree above n, and $\overline{op}$ is an allowed sequence, then we define the n-related normal form of $\overline{op}$ to be an equivalent allowed, normalized sequence of operations affecting the tree above or below n, which must exist by the above lemma. We can now show the result for SEDTDs:

Lemma 3 *For SEDTD-based policies, syntactic consistency implies consistency.*

Proof We prove the contrapositive. Suppose P is inconsistent, and let t be a tree, $\overline{op}$ a sequence allowed on t, and op' denied on t by P, such that $[\![\overline{op}]\!](t) = [\![op']\!](t)$. We consider the four cases for op':

- $op' = \mathsf{insert}(n,t)$. Consider the normal form of the $\overline{op}$ restricted to the updates related to n. Clearly $\overline{op}$ cannot consist only of updates at or below n since an insertion at n cannot be simulated by a deletion at n or by any operations that only apply below n. Also, no replacement can happen at n. If there is a deletion above n, there must also be an insertion above n that restores the extra deleted nodes and also has the effect of $\mathsf{insert}(n,t)$. Hence there is a violation of rule 1. Otherwise, if there is a replacement above node n, then there must be one or more replacements restoring the rest of the tree to its previous form and inserting t, violating rule 2b (since the chain of replacements must be allowed by a cycle in some graph $\mathcal{G}_A$)
- $op' = \mathsf{delete}(n,t), \mathsf{replace}(n,s)$. Similar to case for insert, since again these operations cannot be simulated solely by operations at or below n.
- $op' = \mathsf{replace}(n,t')$. There are two possibilities. If the n-related normal form of $\overline{op}$ consists only of replacements at n, then the policy must violate rule 2. Otherwise, an argument similar to that in the above cases can be used to show that P must violate rule 1 or 3. □

4.2.1 Lifting to CEDTDs

We now complete the proof of Theorem 1 by showing that there is a translation from CEDTDs to SEDTDs that associates each CEDTD policy with a unique SEDTD policy.

To translate CEDTDs to SEDTDs, let D be a CEDTD. We define SEDTD D° as follows. If $A \to \mathsf{str} \in D$ then we add $A \to \mathsf{str}$ to D°. If $A \to f_i, \ldots, f_n$ is a rule in D, where each f is a factor of the form $\mathcal{B}_1 + \cdots + B_n$ or B^*, then we translate to $n+1$ rules

$$A \to A_1, \ldots, A'_n \qquad A'_1 \to f_1 \qquad \cdots \qquad A'_n \to f_n$$

The set of type names of D° is that of D extended with each of the A'_i, and each new type name is associated with a fresh node label. This translation can be lifted to a translation on trees, by inserting intermediate nodes with appropriate labels. If t is a tree over CEDTD D, then we write t° for the tree over D° obtained by inserting appropriate intermediate nodes (we assume some deterministic naming scheme for the new nodes). This is analogous to the translations used in other work on SDTDs [11].

More importantly, this translation also can be lifted to handle atomic updates, UATs and policies. For an atomic update, we translate as follows:

$$\begin{aligned}
\mathsf{insert}(n,t)^\circ &= \mathsf{insert}(m_i,t^\circ)\\
\mathsf{delete}(n)^\circ &= \mathsf{delete}(n)\\
\mathsf{replace}(n,t)^\circ &= \mathsf{replace}(n,t^\circ)\\
\mathsf{replaceVal}(n,s)^\circ &= \mathsf{replaceVal}(n,s)
\end{aligned}$$

where in the case for insert, we assume $\eta(n) = A$ and $A \to f_1,\dots,f_n$ and $f_i = B^*$ and $t \in L(B)$, and node m_i is the child of n in t° associated with A_i. For the other updates, the translation is type-insensitive.

Similarly, UATs can be translated as follows:

$$\begin{aligned}
(A,\mathsf{insert}(B))^\circ &= (A_i,\mathsf{insert}(B))\\
(A,\mathsf{delete}(B))^\circ &= (A_i,\mathsf{delete}(B))\\
(A,\mathsf{replace}(B_i,B_j))^\circ &= (A_i,\mathsf{replace}(B_j,B_k))\\
(A,\mathsf{replaceVal})^\circ &= (A,\mathsf{replaceVal})
\end{aligned}$$

where, in the case for insert and delete, we assume $A \to f_1,\dots,f_n$ and $f_i = B^*$, and in the case for replace we assume that $A \to f_1,\dots,f_n$ and $f_i = (B_1 + \dots + B_m)$, with B_j, B_k among $B_1,\dots,B_m$. If S is a set of UATs, we write S° for the set of translations and we write P° for $(\mathcal{A}^\circ, \mathcal{F}^\circ)$.

The key property of the translation is that it preserves consistency. To show this, we need a number of (routine) properties:

Lemma 4 *Fix a CEDTD D and policy P.*

1. *If $T \in L(D)$ then $T^\circ \in L(D^\circ)$.*
2. *If $UAT \in \mathsf{valid}(D)$ then $UAT^\circ \in \mathsf{valid}(D^\circ)$.*
3. *If op $\mathsf{matches}_t^\eta$ UAT then op° $\mathsf{matches}_{t^\circ}^{\eta^\circ}$ UAT°, where η° is the unique typing of t° in D°, and $[\![op]\!](t)^\circ = [\![op^\circ]\!](t^\circ)$.*
4. *If $\overline{op}$ is a valid run with respect to D on T then $\overline{op}^\circ$ is valid with respect to $\mathcal{F}^\circ$ on T° and $[\![\overline{op}]\!](T)^\circ = [\![\overline{op}^\circ]\!](T^\circ)$.*
5. *If $op \in [\![S]\!](T)$ then $op^\circ \in [\![S^\circ]\!](T^\circ)$.*
6. *If $\overline{op}$ is a valid run allowed by P then $\overline{op}^\circ$ is valid run allowed by P°.*

Lemma 5 *If P is syntactically consistent then P° is syntactically consistent.*

Proof Straightforward, by unfolding definitions.

Lemma 6 *If P° is consistent then P is consistent.*

Proof We prove the contrapositive. If P is inconsistent, let T be a tree, $\overline{op}$ an allowed sequence, and op a forbidden update with $[\![\overline{op}]\!](T) = [\![op]\!](T)$. Then by Lemma 4, T° is a tree and $\overline{op}^\circ$ is an allowed sequence and op° is a forbidden operation on T°, and also $[\![\overline{op}^\circ]\!](T^\circ) = [\![\overline{op}]\!](T)^\circ = [\![op]\!](T)^\circ = [\![op^\circ]\!](T^\circ)$. Hence, P° is inconsistent.

Theorem 1 now follows from Lemmas 3, 5 and 6.

Remark 1 We considered a number of alternative generalizations including full DTDs or EDTDs before settling on CEDTDs. It turns out to be quite difficult to analyze policies with respect to general DTDs because of the fact that individual updates typically insert, delete or replace one node at a time, which can temporarily invalidate the document with respect to the DTD. In related work, Jacquemard and Rusinowitch show that consistency analysis for policies where the DTD can be temporarily violated is undecidable [15]. This suggests that there are likely to be subtle issues in increasing the expressiveness of tractable schema and policy languages.

4.3 Partial Policies

Partial policies may be smaller and easier to maintain than total policies, but are ambiguous because some permissions are left unspecified. Thus, there may be many ways to extend a partial policy to a consistent total one. In this section we consider the question of how to associate a consistent partial policy with a unique consistent total policy.

One (unsatisfactory) solution to this problem is to deny all of the operations that are not explicitly allowed by a partial policy. However, this can easily yield an inconsistent policy, because a UAT that is not mentioned by the policy could be simulated by its allowed UATs. Instead, we consider whether it is possible to find a total policy that only allows UATs that can already be simulated by the partial policy's allowed UATs. This approach does not add any privileges that the user did not already have according to the partial policy, in accordance with the *principle of least privilege* [26].

However, it is not obvious that a partial policy (even if consistent) has any consistent total extension. We will now show that a consistent partial policy does have a consistent extension, and in particular it has a unique least-privilege consistent extension. These 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 extends P. We say that a

partial policy P is *semi-consistent* if it has a consistent total extension. For example, a partial policy on the schema of Figure 5 which allows $(B, \mathsf{insert}(E))$, $(B, \mathsf{delete}(E))$, and denies $(H, \mathsf{replaceVal})$ is not semi-consistent, because any extension of the policy must allow $(H, \mathsf{replaceVal})$ to be consistent.

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 semi-consistent 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 7 *If P_1 and P_2 extend P_0 then $P_1 \wedge P_2$ extends P_0.*

Proof Since both P_1 and P_2 extend P_0, we have $\mathcal{A}_{P_1}, \mathcal{A}_{P_2} \supseteq \mathcal{A}_{P_0}$ and $\mathcal{F}_{P_1}, \mathcal{F}_{P_2} \supseteq \mathcal{F}_{P_0}$; hence

$$\mathcal{A}_{P_1 \wedge P_2} = \mathcal{A}_{P_1} \cap \mathcal{A}_{P_2} \supseteq \mathcal{A}_{P_0} \cap \mathcal{A}_{P_0} = \mathcal{A}_{P_0}$$
$$\mathcal{F}_{P_1 \wedge P_2} = \mathcal{F}_{P_1} \cup \mathcal{F}_{P_2} \supseteq \mathcal{F}_{P_0} \cup \mathcal{F}_{P_0} = \mathcal{F}_{P_0}$$

This completes the proof. □

Lemma 8 *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 By the previous lemma, $P_1 \wedge P_2$ extends P_0. Suppose $P_1 \wedge P_2$ is inconsistent. Then there exists an XML tree t, an atomic operation $op_0 \in [\![\mathcal{F}_{P_1 \wedge P_2}]\!](t)$, and a sequence $\overline{op}$ allowed on t by $P_1 \wedge P_2$, such that $[\![op_0]\!](t) = [\![\overline{op}]\!](t)$. Now $\mathcal{F}_{P_1 \wedge P_2} = \mathcal{F}_{P_1} \cup \mathcal{F}_{P_2}$, so op_0 must be forbidden (with respect to some typing η) by either P_1 or P_2. On the other hand, $\overline{op}$ must be allowed (with respect to some initial typing $\overline{\eta}$) 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 3 *Each semi-consistent partial policy P has a unique $\leq$-least consistent total extension $P^\dagger$.*

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

In the rest of this section, we show how to find the least-privilege consistent extension, or determine that none exists (and hence that the partial policy is not semi-consistent). The basic idea is to define an explicit transformation T on policies that adds allowed UATs that can be simulated by other allowed UATs. Define the operator $T : \mathcal{P}(\mathsf{valid}(D)) \to \mathcal{P}(\mathsf{valid}(D))$ as:

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

Clearly, T is monotonic, so has a least fixed point T^* on the finite lattice $\mathcal{P}(\mathsf{valid}(D))$. The aim of the next sequence of lemmas is to show that if P is consistent then $T^* = P^\dagger$, while if P is not semi-consistent then T^* overlaps with $\mathcal{F}_P$. Thus, computing T^* will give us a PTIME algorithm for either computing $P^\dagger$ or determining that P is not semi-consistent.

Since T^* calculates a fixed point, it is closed under applications of the rules (1)–(3) characterizing consistency. Hence, for any S, we obtain a consistent policy by allowing $T^*(S)$ and forbidding its complement. We define

$$(\mathcal{A}, \mathcal{F})^* = (T^*(\mathcal{A}), \mathsf{valid}(D) \setminus T^*(\mathcal{A})) \ .$$

Lemma 9 *Let S be a set of privileges. Then $T(S) = S$ if and only if $(S, \mathsf{valid}(D) \setminus S)$ is consistent. Moreover, P^* is a consistent policy over D.*

Proof For the first part, suppose $T(S) = S$. Then by inspection, S satisfies the conditions of Theorem 1 and so $(S, \mathsf{valid}(D) \setminus S)$ is consistent. Conversely if $(S, \mathsf{valid}(D) \setminus S)$ is consistent then the conditions of Theorem 1 imply that $T(S) = S$. The second part is immediate.

Corollary 1 *For any semi-consistent partial policy P, we have $P^\dagger = P^*$.*

Proof Assume P is semi-consistent and let $S = \mathcal{A}_P$. Let $P^\dagger = (\mathcal{A}^\dagger, \mathcal{F}^\dagger)$ be its least-privilege consistent extension. Since P^* is consistent, total and $P \leq P^*$, it follows that $P^\dagger \leq P^*$, since $P^\dagger$ is the least consistent total extension of P. Conversely, observe that $T^*(\mathcal{A}^\dagger) = \mathcal{A}^\dagger$ since $P^\dagger$ is syntactically consistent. Hence, $\mathcal{A}^\dagger$ is a fixed point of T so $T^*(\mathcal{A}^\dagger) \subseteq \mathcal{A}^\dagger$ since $T^*(\mathcal{A}^\dagger)$ is the least fixed point of T^* extending $\mathcal{A}^\dagger$. This implies $P^* \leq P^\dagger$. To conclude, $P^* = P^\dagger$ since $\leq$ is anti-symmetric on total policies.

Next, we observe that the updates licensed by $T^*(S)$ can always be simulated by sequences of updates in S.

Lemma 10 *If $uat \in T^*(S)$ then any operation op_0 matching uat on t can be simulated using a sequence of operations $\overline{op}$ that is valid and allowed on t by S (that is, such that $[\![op_0]\!](t) = [\![\overline{op}]\!](t)$).*

Proof First, assume $uat \in T(S)$. We prove the consequence by cases according to the definition of T. If $uat \in S$ then there is nothing to do.

If for some A, B we have $uat = (C, op)$ with $B \leq_D C$, with $B^* \in Rg_D(A)$ and $\{(A, \mathsf{insert}(B)), (A, \mathsf{delete}(B))\} \subseteq S$, then let $n = target(op_0)$, let m be the B-labeled node above m in t (there must be exactly one), and let t' be the subtree of t rooted at m. We can simulate op_0 by deleting the B-labeled subtree to which op_0 applies, then inserting the tree resulting from applying op_0; thus, the sequence $\overline{op} = \mathsf{delete}(m); \mathsf{insert}(n, [\![op_0]\!](t'))$ simulates op_0 and is allowed.

If for some A, B we have $uat = (C, op)$ with $B_i \leq_D C$, and $(B_1 + \ldots + B_n) \in RG_D(A)$, and $(B_i, B_i) \in \mathcal{G}_A^+(S)$, then let $B_{i_1}, \ldots, B_{i_k}$ be a cycle in $\mathcal{G}_A$ beginning and ending with B_i. Again let $n = target(op_0)$ and let m be the (unique) B_i-labeled node above n, and t' be the subtree of t rooted at m. Let $t_1, \ldots, t_{k-1}$ be arbitrary trees disjoint from t and satisfying $t_j \in I_D(B_{i_j})$. (The latter sets are always nonempty so such trees may be found.) Now consider the update sequence

$$\begin{aligned}\overline{op} = \;& \mathsf{replace}(m, t_1); \mathsf{replace}(rt_{t_1}, t_2); \\ & \vdots \\ & \mathsf{replace}(rt_{t_{n-2}}, t_{n-1}); \mathsf{replace}(rt_{t_n-1}, [\![op_0]\!](t'))\end{aligned}$$

This update sequence is allowed on t and simulates op_0.

Finally, if $uat = (C, \mathsf{replace}(B_i, B_j))$ where $(B_1 + \cdots + B_n) \in Rg_D(C)$ and $(B_i, B_j) \in \mathcal{G}_C^+(S)$ then let $n = target(op_0)$ and let t' be the subtree rooted at n. Let $B_{i_1}, \ldots, B_{i_k}$ be a sequence of nodes forming a path from $B_i = B_{i_1}$ to $B_j = B_{i_k}$ in $\mathcal{G}_C$, and choose $t_1, \ldots, t_{k-1}$ satisfying $t_l \in I_D(B_{i_l})$. Then the update sequence

$$\begin{aligned}\overline{op} = \;& \mathsf{replace}(n, t_1); \mathsf{replace}(rt_{t_1}, t_2); \\ & \vdots \\ & \mathsf{replace}(rt_{t_{n-2}}, t_{n-1}); \mathsf{replace}(rt_{t_n-1}, [\![op_0]\!](t'))\end{aligned}$$

again is allowed and simulates op_0.

Now, if $uat \in T^*(S)$, then the conclusion follows by induction. □

Finally, we show that we can determine (semi-)consistency using T^*. Along the way we show that semi-consistency and consistency are equivalent.

Theorem 2 *Let P be a partial policy. The following are equivalent: (1) P is semi-consistent, (2) P is consistent, and (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 10, 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 $(T(\mathcal{A}_P), \mathsf{valid}(D) \setminus T(\mathcal{A}_P))$ is consistent (by Lemma 9) and extends P (by Lemma 8) since $T^*(\mathcal{A}_P) \cap \mathcal{F}_P = \emptyset$ implies $\mathcal{F}_P \subseteq \mathsf{valid}(D) \setminus T^*(\mathcal{A}_P)$. □

Since each step of T is computable in PTIME and the number of elements of $\mathsf{valid}(D)$ is polynomial in the size of D, we can compute T^* in polynomial time using least fixed-point iteration. Hence, we can decide whether a partial policy is (semi-)consistent and if so find $P^\dagger$ in PTIME.

In fact, the T operator as defined above is idempotent, but this is not needed for the results above.

5 Policy Repairs for SEDTDs

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 allowed 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.

In this article 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 11 A policy $P' = (\mathcal{A}', \mathcal{F}')$ is a *repair* of a policy $P = (\mathcal{A}, \mathcal{F})$ defined over a CEDTD D if and only if:

1. P' is a policy defined over D,
2. P' is consistent, and
3. $P' \leq P$.

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 $k > 0$, 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 [30].

Theorem 3 *The total-repair and partial-repair problems for policies over SEDTDs are* NP*-complete.*

Proof We will concentrate on the total-repair problem. The proof for the partial-repair problem is analogous.

First we will prove that the total-repair is in NP. We can determine if there is a repair $P' = (\mathcal{A}', \mathcal{F}')$ of P such that $|\mathcal{A} \backslash \mathcal{A}'| < k$, by guessing a policy P', checking if $|\mathcal{A} \backslash \mathcal{A}'| < k$ and if it is consistent. Since consistency and the distance can be checked in polynomial time, the algorithm is in NP.

To prove that the problem is NP-hard, we reduce from the *edge-deletion transitive-digraph problem* which is NP-complete [29, 30]. The problem consists in, given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with $\mathcal{V} = \{v_1, \ldots, v_n\}$ and $\mathcal{E}$ a set of edges without self-loops, determine if there exists a set $\mathcal{G}' = (\mathcal{V}, \mathcal{E}')$ such that $\mathcal{E}' \subseteq \mathcal{E}$, $\mathcal{G}'$ is transitive and $|\mathcal{E} \setminus \mathcal{E}'| < k$. Now, let us define a DTD D and a policy P. The production rules of D are:

$$A \to v_1 + \cdots + v_n$$
$$v_i \to \mathsf{str} \qquad \text{for } i \in [1, n]$$

The policy $P = (\mathcal{A}, \mathcal{F})$ is such that

$$\mathcal{A} = \{(A, \mathsf{replace}(v_i, v_j)) | (v_i, v_j) \in \mathcal{E}\} \cup \{(v_i, \mathsf{replaceVal}) \mid v_i \in \mathcal{V}\}$$
$$\mathcal{F} = \mathsf{valid}(D) \setminus \mathcal{A}\ .$$

It is easy to see that $\mathcal{G}_A = \mathcal{G}$ and therefore finding a repair will consist of finding the minimal number of edges to delete from $\mathcal{G}$ to make the graph transitive. □

If the SEDTD has no disjunction in its production rules then the only type of inconsistencies arise when an operation is denied below an element type that is allowed to be inserted and deleted (case 1 of Definition 10). In this case the distance between the policy and its minimal repair will be equal to the number of such inconsistencies. Thus, the total-repair problem in this case is in PTIME.

5.1 Computing Minimal Repairs

For a policy P defined over a schema D we will construct a disjunctive logic program with weak negation such that there is a one-to-one correspondence between each model of it and the repairs of P. Satisfying models with minimal violations of weak constraints can be found by answer-set programming solvers such as DLV [19], and we will use DLV syntax. Upper and lower case letters denote variables and constants respectively. The program uses predicates: (i) UAT to store the valid UATs; (ii) a and d to store the allowed and denied UATs respectively; (iii) $desc$ to store the subelement relation between element types; (iv) rA and rD to contain the allowed and forbidden UATs in the repair.

Definition 12 Given a DTD D and a (partial) policy $P = (\mathcal{A}, \mathcal{F})$ let the repair program be $\Pi(D, P) = \Pi^f(D, P) \cup \Pi^r$ where:

1. $\Pi^f(D, P)$ contains the following facts:

$UAT(a, ins, b, null)$.	for every $(a, \mathsf{insert}(b)) \in \mathsf{valid}(D)$
$UAT(a, del, b, null)$.	for every $(a, \mathsf{delete}(b)) \in \mathsf{valid}(D)$
$UAT(a, rep, b, c)$.	for every $(a, \mathsf{replace}(b, c)) \in \mathsf{valid}(D)$
$a(a, ins, b, null)$	for every $(a, \mathsf{insert}(b)) \in \mathcal{A}$
$a(a, del, b, null)$	for every $(a, \mathsf{delete}(b)) \in \mathcal{A}$
$a(a, rep, b, c)$	for every $(a, \mathsf{replace}(b, c)) \in \mathcal{A}$
$a(a, repV, null, null)$	for every $(a, \mathsf{replaceVal}) \in \mathcal{A}$
$d(a, ins, b, null)$	for every $(a, \mathsf{insert}(b)) \in \mathcal{F}$
$d(a, del, b, null)$	for every $(a, \mathsf{delete}(b)) \in \mathcal{F}$
$d(a, rep, b, c)$	for every $(a, \mathsf{replace}(b, c)) \in \mathcal{F}$
$d(a, repV, null, null)$	for every $(a, \mathsf{replaceVal}) \in \mathcal{F}$
$desc(a, b)$.	for every type a and b such that $a \leq_D b$

2. Π^r contains the following rules:

$$rD(X, Op, E_1, E_2) \vee rA(X, Op, E_1, E_2) \leftarrow a(X, Op, E_1, E_2). \quad (1)$$
$$rD(X, Op, E_1, E_2) \leftarrow d(X, Op, E_1, E_2). \quad (2)$$
$$rA(Z, Op, E_1, E_2) \leftarrow rA(X, ins, Y, null), rA(X, del, Y, null), desc(Y, Z), UAT(Z, Op, E_1, E_2). \quad (3)$$
$$rA(X, rep, Y, W) \leftarrow rA(X, rep, Y, Z), rA(X, rep, Z, W). \quad (4)$$
$$rA(W, Op, E_1, E_2) \leftarrow rA(X, rep, Y, Z), rA(X, rep, Z, Y), desc(Y, W), UAT(W, Op, E_1, E_2). \quad (5)$$
$$\leftarrow rD(X, Op, E_1, E_2), rA(X, Op, E_1, E_2). \quad (6)$$
$$rD(Z, Op, E_1, E_2) \leftarrow UAT(Z, Op, E_1, E_2), \mathit{not}\ rA(Z, Op, E_1, E_2). \quad (7)$$
$$\Leftarrow a(X, Op, E_1, E_2), rD(X, Op, E_1, E_2). \quad (8)$$

Rules (1) and (2) are used to construct all possible policies obtained from policy P by keeping all denied operations as denied but letting allowed operations to be changed to either allowed or denied. This implements our choice of looking for repairs that are more restrictive than the policy that we are trying to repair. Rules (3), (4) and (5) make sure that the repairs have no inconsistencies. Indeed, rule (3) ensures that if it is possible to insert and delete a certain element type, then all the operations below it should be allowed; rule (4) forces the replace graphs to be transitive; and rule (5) checks that if an element type belongs to a cycle, then everything is allowed below it. The denial constraint (6) makes sure that no operation is both allowed and denied in the repair. Rule (7) computes a total policy if P was partial. Finally, rule (8) corresponds to a weak constraint that ensures that the number of permissions that are modified is minimized. A weak constraint is of the form $\Leftarrow \varphi$, where φ is a conjunction of atoms. The weak constraint is violated every time an assignment makes φ true in a model. The models of a program with weak constraints correspond to the answer sets that minimize the violations of the weak constraints. Note that only $\Pi^f(D, P)$ depends on the schema and the policy, since the rules in Π^r are independent.

Given a schema D and a policy P, the repair $P^{\mathcal{M}} = (\mathcal{A}^{\mathcal{M}}, \mathcal{F}^{\mathcal{M}})$ associated to an optimal answer sets model $\mathcal{M}$ of $\Pi(\mathcal{F}, P)$ is obtained from the rA and rD predicates. This

is, $\mathcal{A}^{\mathcal{M}}$ and $\mathcal{F}^{\mathcal{M}}$ contain all the UATs in rA and rD respectively. Every policy $P^{\mathcal{M}}$ obtained from an optimal answer set M of $\Pi(D,P)$ is a minimal repair of policy P. Furthermore, for every minimal repair P' of policy P there exists an optimal answer set M of $\Pi(D,P)$ such that $P^M = P'$.

Example 5 (example 4 continued) Program $\Pi^f(D,P)$ contains the following facts:

$UAT(r,rep,a,b).$	$a(c,del,f,null).$	$desc(a,r).$
$UAT(r,rep,b,k).$	$a(h,repV,null,null)$	$desc(c,r).$
$UAT(c,del,f,null).$	$\ldots$	$desc(c,a).$
$\ldots$	$d(r,rep,b,k).$	$desc(f,a).$
$a(r,rep,a,b).$	$d(g,rep,h,i).$	$\ldots$
$a(r,rep,b,j)$	$\ldots$	
$a(c,ins,a,null).$	$desc(a,a).$	

Program $\Pi(D,P)$ has 16 optimal answer set models that have a 1-1 correspondence with the minimal repairs which are obtained by the combination of denying:

1. $(B,\mathsf{insert}(E))$ or $(B,\mathsf{delete}(E))$;
2. $(J,\mathsf{insert}(G))$ or $(J,\mathsf{delete}(G))$;
3. $(E,\mathsf{insert}(G))$ or $(E,\mathsf{delete}(G))$; and
4. $\{(R,\mathsf{replace}(A,B)),(R,\mathsf{replace}(J,K))\}$ or $\{(R,\mathsf{replace}(B,J)),(R,\mathsf{replace}(J,K))\}$.

The first three resolve inconsistencies that correspond to violations of condition (1) of Definition 10. The fourth corresponds to violations of condition (2a). ■

Finally, note that a simple variation of the above approach can be used to test whether a (partial) policy is consistent, simply by replacing the rules Π^r with the following Π^c:

$$a(Z,Op,E_1,E_2) \leftarrow a(X,ins,Y,null), a(X,del,Y,null), desc(Y,Z), UAT(Z,Op,E_1,E_2). \quad (9)$$

$$a(X,rep,Y,W) \leftarrow a(X,rep,Y,Z), a(X,rep,Z,W). \quad (10)$$

$$a(W,Op,E_1,E_2) \leftarrow a(X,rep,Y,Z), a(X,rep,Z,Y), desc(Y,W), UAT(W,Op,E_1,E_2). \quad (11)$$

$$d(Z,Op,E_1,E_2) \leftarrow UAT(Z,Op,E_1,E_2), \mathit{not}\ a(Z,Op,E_1,E_2). \quad (12)$$

$$i(X,Op,E_1,E_2) \leftarrow a(X,Op,E_1,E_2), d(X,Op,E_1,E_2). \quad (13)$$

This is a stratified Datalog program that has a solution containing a tuple $i(X,Op,E_1,E_2)$ for every inconsistent UAT.

5.2 Approximate Algorithms for Minimal Repairs

In this section we discuss a polynomial repair algorithm that finds a repair of a total or partial policy which is not necessarily minimal but tries to minimize the number of changes.

The algorithm to compute a minimal repair of a policy relies on the independence property mentioned earlier, between insert/delete (Definition 10, condition (1)) inconsistencies and replace (Definition 10, conditions (2a) and (2b)) operations. As a result, 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* $MG_D = (G_D,\mu,\chi)$ μ is a function from nodes in $\mathcal{V}_D$ to {"+", "−"} and χ is a function from $\mathcal{V}_D$ to $\{-1,\perp\}$. In a marked graph for a DTD D and a policy $P = (\mathcal{A},\mathcal{F})$, 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.

Algorithm 1 markGraph

Input: DTD Graph G_D, Policy $P = (\mathcal{A},\mathcal{F})$
Output: Marked DTD Graph $MG_D = (G_D,\mu,\chi)$

1: Let $visited(A) = \mu(A) = \chi(A) = -1$ for every $A \in V_D$
2: $MG_D = (G_D,\mu,\chi)$
3: **for all** $(A,Op) \in \mathcal{F}$ **do**
4: $\quad\mu(A) = 0$
5: **markNode**$(MG_D, visited, G_D.root)$
6: **for all** $A \in V_D$ and $B \in V_D$ **do**
7: $\quad$**if** $(A,\mathsf{insert}(B)) \in \mathcal{A}$, $(A,\mathsf{delete}(B)) \in \mathcal{A}$ and $\mu(A) = 0$ **then**
8: $\quad\quad\chi(A) =$ '$\perp$'
9: **return** MG_D

Example 6 Consider the graph for DTD D in Fig. 10(a) and policy $P = (\mathcal{A},\mathcal{F})$, with $\mathcal{A}$ defined in Example 3. The result of applying **markGraph** to this DTD and policy is shown in Fig. 10(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 pur-

Algorithm 2 markNode

Input: Marked DTD Graph $MG_D = (G_D,\mu,\chi)$, function $visited$, Node A.
Output: The subtree rooted in A is marked

1: **if** $visited(A) < 0$ **then**
2: $\quad$**for all** $(A,B) \in E_D$ **do**
3: $\quad\quad$**markNode**(MG_D, B)
4: $\quad\quad$**if** $\mu(B) = 0$ **then**
5: $\quad\quad\quad\mu(A) = 0$
6: $\quad$**if** $\mu(A) = -1$ **then**
7: $\quad\quad\mu(A) = 1$
8: $\quad visited(A) = 1$

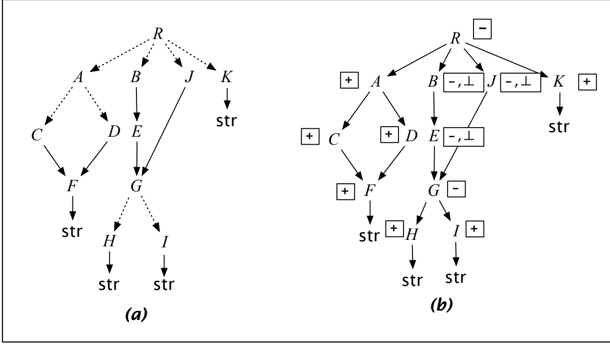

Fig. 10 DTD Graph (a) and Marked DTD Graph (b) for the DTD in Fig. 5

Algorithm 3 $\textbf{InsDelRepair}$

Input: DTD graph G_D, security policy P
Output: Set of *UATs* to remove from P to restore consistency in P *w.r.t.* insert/delete inconsistencies

1: $MG_D \leftarrow \textbf{markGraph}(G_D, P)$
2: $changes \leftarrow \emptyset$
3: **for all** $A \in \mathcal{V}_D$ and $(A, B) \in E_D$ **do**
4:     **if** $\chi(A) =$ "$\perp$" **then**
5:         Randomly choose either $(A, \mathsf{insert}(B))$ or $(A, \mathsf{delete}(B))$ and assign it to U
6:         $changes \leftarrow changes \cup U$
7: **return** $changes$

poses we do not show the multiplicities in the marked DTD graph. ■

5.2.1 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 Definition 10, 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 $\textbf{InsDelRepair}$ 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 7 Given the marked DTD graph in Fig. 10(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:

$\{(B, \mathsf{insert}(E)), (B, \mathsf{delete}(E))\}$
$\{(E, \mathsf{insert}(G)), (E, \mathsf{delete}(G))\}$
$\{(J, \mathsf{insert}(G)), (J, \mathsf{delete}(G))\}$

Algorithm 4 $\textbf{ReplaceNaive}$

Input: Node R, Marked Graph MG_D
Output: Set Sol containing pairs $(B, \mathcal{E}_{del})$ where B is a node reachable from R in MG_D, and $\mathcal{E}_{del}$ a set of edges to delete from $\mathcal{G}_B$ to make it consistent

1: **if** $Rg(R) := B_1 + B_2 \ldots + B_n$ **then**
2:     Let $\mathcal{G}_R$ be the replace graph for R
3:     $\mathcal{E}_{del} \leftarrow \emptyset$
4:     Let stack S contain all the nodes in c
5:     **while** S not empty **do**
6:         $B \leftarrow S.pop()$
7:         **for all** A in V_R, s.t. $(A, B) \in \mathcal{E}_R \setminus \mathcal{E}_{del}$ **do**
8:             **for all** $C \in V_R$, s.t. $(B, C) \in \mathcal{E}_R \setminus \mathcal{E}_{del}$ **do**
9:                 /* *If there is an edge missing for transitive or if there is a cycle over a node with a* UAT *forbidden below* */
10:                 **if** $A \neq C$ or $\mu(A) =$ "$-$" **then**
11:                     Let e be one of (A, B), (B, C) (chosen randomly)
12:                     $\mathcal{E}_{del} = \mathcal{E}_{del} \cup \{e\}$
13:                     **for all** $F \in \mathcal{V}_R$ s.t. F is reachable from the start vertex of e in $\mathcal{G}_R$ **do**
14:                         $S.push(F)$
15:     $Sol \leftarrow \{(R, \mathcal{E}_{del})\}$
16: **else**
17:     $Sol \leftarrow \emptyset$
18: **for all** $(R, B) \in \mathcal{E}_R$ **do**
19:     $Sol \leftarrow Sol \cup \textbf{ReplaceNaive}(B, MG_D)$
20: **return** Sol

■

5.2.2 Repairing Inconsistencies for Replace Operations

There are two types of inconsistencies related to replace operations (see Definition 10, conditions (2a)–(2b)): the first arises when some element 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 $\mathcal{F}$. 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 $\textbf{ReplaceNaive}$ 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 $\mathcal{G}_A$ for 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 $(B, D) \in \mathcal{F}_A$, *i.e.,* if there is forbidden-transitivity. In this way, the final graph will satisfy condition (2a) of Definition 10. 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 **ReplaceNaive** returns the set of edges to delete from each node to remove replace inconsistencies.

Example 8 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 $\mathcal{G}_R = (\mathcal{V}, \mathcal{E})$, shown in Fig. 11(a) is the source of many inconsistencies. A possible execution of **ReplaceNaive** (shown in Fig. 12) is: $(A, B), (B, J) \in \mathcal{E}$ but $(A, J) \in \mathcal{F}$, so (A, B) or (B, J) should be deleted, say (A, B). Now, (B, J), (J, K) $\in \mathcal{E}$ and $(B, K) \in \mathcal{F}$, therefore we delete either (B, J) or (J, K), say (B, J). Next, (K, J), $(J, K) \in \mathcal{E}$ and $\mu(J) =$ "$-$" in Fig. 10(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-transitive and 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. ■

The **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).

ReplaceSetCover is an alternative to **ReplaceNaive** that can find a solution closer to a minimal repair. This algorithm also uses a modified version of the Floyd-Warshall algorithm. In this case, the modification consists in computing the transitive closure of the replace graph $\mathcal{G}_A$ and labelling each newly constructed edge e with a set of *justifications* $\mathcal{J}$. Each justification contains 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 labelled 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 $O(2^{|\mathcal{E}|})$. To avoid the exponential number of justifications, **ReplaceSetCover** assigns at most $\mathfrak{J}$ justifications to each edge or node, where $\mathfrak{J} > 0$ is a given parameter. This new labelled graph is then used to construct an instance of the minimum set cover problem (MSCP) [24]. 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-transitive or negative-cycles. Thus, the justifications have to be recomputed and the set cover run again until there are no more replace inconsistencies.

Example 9 For $\mathfrak{J} = 1$, the first computation of justifications of **ReplaceSetCover** results in the graph in Fig. 11 (b) with the following justifications:

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

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 $\mathcal{J}$. ■

Algorithm 5 ReplaceSetCover

Input: Node R, marked DTD graph MG_D, integer $\mathfrak{J}$
Output: Set Sol containing pairs $(B, \mathcal{E}_{del})$ where B is a node reachable from R in MG_D, and $\mathcal{E}_{del}$ a set of edges to delete from $\mathcal{G}_B$ to make it consistent

```
1: Sol ← ∅, E_del ← ∅, done ← false
2: if Rg(R) := B1 + B2 ... + Bn then
3:     Let G_R = (V, E) be the replace graph for R
4:     G ← G_R
5:     while ¬done do
6:         G+ ← ComputeJustifications(G, ℑ)
7:         /* Algorithm setCoverAlg takes the graph G+ with the
           justifications and the set of forbidden edges and returns the
           edges to delete from G_A */
8:         F_R ← denied edges (edges that do not belong to G_R)
9:         E_sc ← setCoverAlg(G+, F_R)
10:        if E_sc ≠ ∅ then
11:            remove edges in E_sc from G
12:            E_del ← E_del ∪ E_sc
13:        else
14:            done = true
15:    Sol ← Sol ∪ {(R, E_del)}
16: for all (R, B) ∈ E_R do
17:    Sol ← Sol ∪ ReplaceSetCover(B, MG_D, ℑ)
18: return Sol
```

The previous example shows that, for each node 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 $\mathcal{G}_A^+ = (\mathcal{V}, \mathcal{E})$ and the set of forbidden edges $\mathcal{F}_A$ is $MSCP(\mathcal{G}_A^+, \mathcal{F}_A) = (\mathcal{U}, \mathcal{S})$ where

1. $\mathcal{U} = \{s \mid s \in \mathcal{J}(e), e \in \mathcal{F}_A\} \cup \{s \mid s \in \mathcal{J}(V), V \in \mathcal{V}\}$, and
2. $\mathcal{S} = \{\mathcal{I}(e) \mid e \in \mathcal{E}\}$ 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. In this instance of the MSCP, the $\mathcal{U}$ is a set of justifications, therefore, $\mathcal{S}$ is a set of sets of justifications.

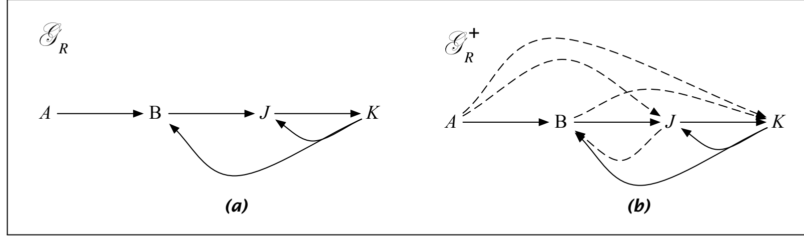

Fig. 11 Replace $\mathcal{G}_R$ (a) and Transitive Replace Graph $\mathcal{G}_R^+$(b)

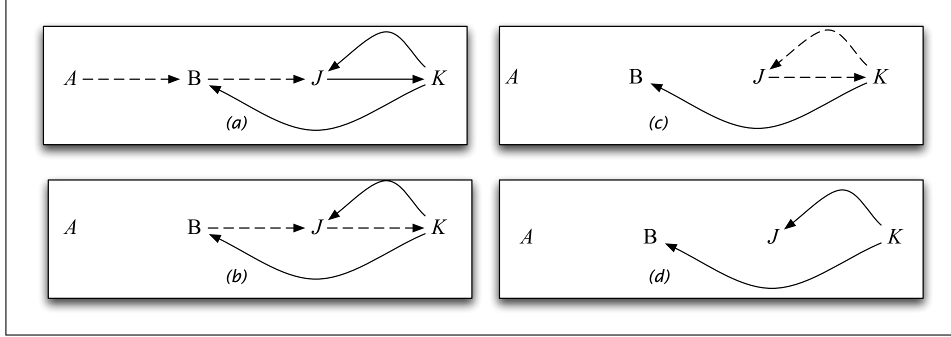

Fig. 12 Execution of $\mathbf{ReplaceNaive}$ on $\mathcal{G}_R$

Algorithm 6 ComputeJustifications

Input: Replace Graph $\mathcal{G}_R$, Maximum Number of Justifications $\mathfrak{J}$
Output: $\mathcal{G}_R^+$, *i.e.,* transitive closure of $\mathcal{G}_R$ where each edge and node is labelled with a set $\mathcal{J}$ containing at most $\mathfrak{J}$ justifications per edge

```
1: /* Initialize */
2: E ← ∅
3: for all A ∈ V_R and B ∈ V_R do
4:    if (A, B) ∈ E_R then
5:       J((A, B)) ← {{(A, B)}}
6:    else
7:       J(A, B) ← ∅
8: /* Compute justifications of cycles and missing edges */
9: for all A ∈ V_R do
10:    for all B ∈ V_R, s.t. (B, A) ∈ E_R ∪ E and A ≠ B do
11:       for all C ∈ V_R, s.t. (A, C) ∈ E_R ∪ E an A ≠ C do
12:          if (B, C) ∉ E_R then
13:             if (B, C) ∉ E and B ≠ C then
14:                E ← E ∪ {(B, C)}
15:          for all j1 ∈ J((B, A)) and j2 ∈ J((A, C)) do
16:             if |J((B, C))| < 𝔍 then
17:                J((B, C)) ← J((B, C)) ∪ {j1 ∪ j2}
18: /* Assign justifications to nodes involved in negative cycles */
19: for all A ∈ V_R do
20:    if μ(A) = "−" then
21:       J(A) ← J(A, A)
22:    else
23:       J(A) ← ∅
24: G_R^+ ← (V_R, E_R ∪ E)
25: return G_R^+
```

Example 10 The minimum set cover instance for the ongoing example $MSCP(\mathcal{G}_R^+, E) = (\mathcal{U}, \mathcal{S})$, is such that

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

$$\mathcal{S} = \{\mathcal{I}((A,B)),\mathcal{I}((B,J)),\mathcal{I}((J,K)),\mathcal{I}((K,J)),\mathcal{I}((K,B))\}\ .$$

The extensions of $\mathcal{I}$ are given in Table 3, 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 $MSCP(\mathcal{G}_R^+)$ 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. ■

The set cover problem is MAXSNP-hard [24], 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 $\mathcal{U}$ [8]. 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 $\mathbf{ReplaceNaive}$ algorithm.

Algorithm $\mathbf{ReplaceRepair}$ will compute the set of *UATs* to remove from $\mathcal{A}$, by using either $\mathbf{ReplaceNaive}$ (if $\mathfrak{J} = 0$) or $\mathbf{ReplaceSetCover}$ (if $\mathfrak{J} > 0$).

5.2.3 Computation of a Repair

Algorithm $\mathbf{Repair}$ 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 $\mathbf{InsDelRepair}$ and

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

Table 3 Set cover problem

Algorithm 7 $\mathbf{ReplaceRepair}$

Input: DTD graph G_D, security policy $P = (\mathcal{A}, \mathcal{F})$, Maximum Number of Justifications $\mathfrak{J}$
Output: Set of *UATs* to remove from $\mathcal{A}$ to restore consistency in P *w.r.t.* replace inconsistencies
1: $MG_D \leftarrow \mathbf{markGraph}(G_D, P)$
2: **if** $\mathfrak{J} = 0$ **then**
3: $\quad Sol \leftarrow \mathbf{ReplaceNaive}(r_D, MG_D)$
4: **else**
5: $\quad Sol \leftarrow \mathbf{ReplaceSetCover}(r_D, MG_D, \mathfrak{J})$
6: $\Delta \leftarrow \emptyset$
7: **for all** $(A, \mathcal{E}_{del}) \in Sol$ **do**
8: $\quad$ **for all** $(B_1, B_2) \in \mathcal{E}_{del}$ **do**
9: $\quad\quad \Delta \leftarrow \Delta \cup (A, \mathsf{replace}(B_1, B_2))$
10: **return** Δ

Algorithm 8 $\mathbf{Repair}$

Input: DTD graph G_D, security policy $P = (\mathcal{A}, \mathcal{F})$, boolean *total*, maximum number of justifications $\mathfrak{J}$
Output: A repair P' of P. The repair is total if parameter *total*= 1, partial otherwise.
1: $\Delta \leftarrow \mathbf{InsDelChecking}(G_D, P) \cup \mathbf{ReplaceRepair}(G_D, P, \mathfrak{J})$
2: $\mathcal{A}' \leftarrow \mathcal{A} - \Delta$
3: **if** *total* **then**
4: $\quad \mathcal{F}' \leftarrow \mathsf{valid}(D) - \mathcal{A}'$
5: **else**
6: $\quad \mathcal{F}' \leftarrow \mathcal{F} \cup \Delta$
7: $P' \leftarrow (\mathcal{A}', \mathcal{F}')$
8: **return** P'

$\mathbf{ReplaceRepair}$. If argument $total$ of algorithm $\mathbf{Repair}$ is $true$, then the repair returned by it will be total. If *false*, then a partial policy such that $\mathcal{F}' = \mathcal{F}$ will be returned.

Theorem 4 *Given a total (partial) policy P, algorithm* $\mathbf{Repair}$ *returns a total (partial) repair of P.*

Proof Given an inconsistent policy $P = (\mathcal{A}, \mathcal{F})$, let us assume, to obtain a contradiction, that the policy $P' = (\mathcal{A}', \mathcal{F}')$ returned by algorithm $\mathbf{Repair}$ is not a repair. Since P' is defined over D, and by construction $P' \leq P$, this implies that P' is not consistent. Then, it should be the case that either the changes returned by:

1. $\mathbf{InsDelRepair}$ *do not solve all the insert/delete inconsistencies*. This implies that there is a node A with production rule $A \to B*$ such that $(A, \mathsf{insert}(B)) \in \mathcal{A}'$, $(A, \mathsf{delete}(B)) \in \mathcal{A}'$ and there is at least one forbidden *UAT*, say (C, op), such that $B \leq_D C$. Since $P' \leq P$, we know $(A, \mathsf{insert}(B)) \in \mathcal{A}$ and $(A, \mathsf{delete}(B)) \in \mathcal{A}$. If we prove that there is always an operation $(G, op) \in \mathcal{F}$ such that $B \leq_D G$, the marked DTD graph would be such that $\chi(A) = \bot$. Then, either $(A, \mathsf{insert}(B))$ or $(A, \mathsf{delete}(B))$ would have been in the changes returned by $\mathbf{InsDelRepair}$ and one of them would not have belonged to P'. Now we will prove that such (G, op) always exists. If $(C, op) \in \mathcal{F}$, then $(G, op) = (C, op)$. On the other hand, if $(C, op) \notin \mathcal{F}$ then (C, op) is either one of the changes returned by $\mathbf{InsDelRepair}$ or $\mathbf{ReplaceRepair}$:
   (a) If (C, op) was a change returned by $\mathbf{InsDelRepair}$, then there was an insert-delete inconsistency, and there is another *UAT* $(F, op_2) \in \mathcal{F}$ such that $C \leq_D F$. As a consequence $B \leq_D F$, hence (G, op).
   (b) If (C, op) was a change returned by $\mathbf{ReplaceRepair}$ this would mean that (C, op) was either involved in a negative-cycle or forbidden-transitivity. The former implies there is another *UAT* $(F, op_2) \in \mathcal{F}$ such that $C \leq_D F$. Then, $B \leq_D F$, and we have found (G, op). The latter case implies there is at least one other $(C, op_2) \in \mathcal{F}$. We have found (G, op).
2. $\mathbf{ReplaceRepair}$ *do not solve all the replace inconsistencies*: This implies that there is a node A with production rule $A \to B_1 + \cdots + B_n$ such that one of the following holds:
   (a) There is an edge (B_i, B_j) in $\mathcal{G}_A^+$ for P', s.t. $(B_i, B_j) \in \mathcal{F}'_A$. If $(B_i, B_j) \in \mathcal{F}_A$, then $\mathbf{ReplaceRepair}$ would have deleted at least one edge from each justification of (B_i, B_j), and therefore, (B_i, B_j) could not be in $\mathcal{G}_A^+$ for P'. On the other hand, if $(B_i, B_j) \notin \mathcal{F}_A$, then $(A, \mathsf{replace}(B_i, B_j))$ implies that it was part of the changes returned by $\mathbf{ReplaceRepair}$. Since both $\mathbf{ReplaceNaive}$ and $\mathbf{ReplaceSetCover}$ check that the final graph has no forbidden-transitivity, this is not possible.
   (b) There is a B_i which is part of a cycle in $\mathcal{G}_A$ for P' and there is a *UAT* $(C, op) \in \mathcal{F}'$ s.t. $B_i \leq_D C$. Since B_i is in a cycle in $\mathcal{G}_A$ for P', it should be part of a cycle in $\mathcal{G}_A$ for P. If $(C, op) \in \mathcal{F}$, then the

inconsistency would have been solved already. On the other hand, if $(C, op) \notin \mathcal{F}$, then (C, op) is either one of the changes returned by $\mathbf{InsDelRepair}$ or $\mathbf{ReplaceRepair}$. By an analogous reasoning as in cases 1(a)-1(b), this is not possible either.

Therefore, P' is consistent and is a repair of P. □

6 Implementation and Experimental Evaluation

We implemented the consistency checking and repair algorithms, for partial and total policies, in a system called ACCon, short for Access Control CONsistency. In ACCon the user can load a DTD or XML Schema (which is internally converted to a CEDTD), and an associated write-access control policy (also expressed in XML). The implementation includes a user interface that displays the inconsistencies and proposes changes to the policy using $\mathbf{InsDelRepair}$, $\mathbf{RepairNaive}$, and $\mathbf{RepairSetCover}$. The user is then able to select among the suggested changes, and ACCon will apply the changes to the policy and check whether consistency is achieved. If the policy is not consistent, the process is repeated until no more inconsistencies are found.

6.1 Experimental setup

We evaluated the closure, consistency and repair algorithms described earlier in the article in comparison with a generic approach based on DLV. Specifically, we used DLV to check the consistency of policies and to calculate exact minimal repairs, when possible.

Our benchmark includes both real and synthetic (randomly-generated) schemas. The real schemas are based on existing schemas for IUPHAR-DB [14], SBML, XMark [27], and Sigmod Record and DBLP bibliography data. IUPHAR-DB is a database of pharmacological receptors maintained by colleagues at the University of Edinburgh; our schema is derived from an XML export format for their data. The SBML (Systems Biology Markup Language) schema is based on SBML level 1, version 2, obtained from the SBML.org website[1]. The Sigmod Record and DBLP schemas are based on the standard schemas for these sources, obtained from the UW XML dataset collection [2].

To generate the synthetic DTDs, we used the DTD Generator[3], a configurable Java program that takes into account a number of different parameters for the production of DTDs such as (a) the maximum depth of the DTD, (b) the number of distinct labels in the DTD, (c) the expected number of children per node (selected randomly by a Poisson distribution), and (d) the probability of '*' per node. The DTDs obtained from the generator were recursive, so we manually translated them into non-recursive ones. Schema nodes were annotated with '*' with probability 0.05. We produced DTDs with 10, 30, 50, 70, 100, 200 and 500 elements. For the DTDs with less than 100 elements we fixed the DTD depth to 4, the number of children of a node to 3, whereas for the DTDs with more than 100 elements, we fixed the number of children to 6 and the number of distinct children per node to 10. We write RANDOM-10, …, RANDOM-500 for these schemas, where RANDOM-n refers to the random schema with n elements.

For both real and synthetic schemas, we report on randomly-generated policies. A random policy is generated by allowing each possible UAT with probability p. For most experiments, we considered policies generated with $p = 0.5$; we also considered the effect of varying this probability on repair size. For most measurements we used justification limit $\mathfrak{J} = 10$. We also considered the effect of varying $\mathfrak{J}$.

All experiments were carried out on a MacBook Pro with 2.8GHz Intel Core 2 Duo CPU with 4GB RAM. We used a library called DLVWrapper to interface with the DLV executable (both are available from DLVSystem S.R.L.).

[1] `http://www.sbml.org`

[2] `http://www.cs.washington.edu/research/xmldatasets/`

[3] DTD Generator: `http://www.l3s.de/~papapetrou/dtdgen.html`

6.2 Experimental results

6.2.1 Completion, consistency and repair time

The experimental results for partial policy closure, consistency checking, and repair time are summarized in Figure 14. We first show the closure time and consistency checking times for our algorithms, then the DLV consistency checking time, then the times for repairing based on $\mathbf{ReplaceNaive}$, $\mathbf{ReplaceSetCover}$ and using DLV. The time for insert–delete repairs is added to both the naive and MSCP-based repair times, to ensure a fair comparison with DLV, which solves both repair problems at once. However, the time needed for insert–delete repairs was negligible compared to that needed for either naive or MSCP repairs (usually, it was less than a millisecond). DLV timed out when we attempted to use it to repair random policies of 100 elements or more. The times reported for DLV include overhead for calling DLV as an external process from our Java implementation, as well as time to construct the DLV program from the schema and policy. The randomly-generated IUPHAR policies were all consistent, so we do not report any repair times for them (in any case, these policies can never have any replace inconsistencies and, as noted above, insert–delete repair is fast).

Our implementations of both the policy closure and consistency algorithms are faster than the generic approach used by DLV by approximately an order of magnitude. Our naive

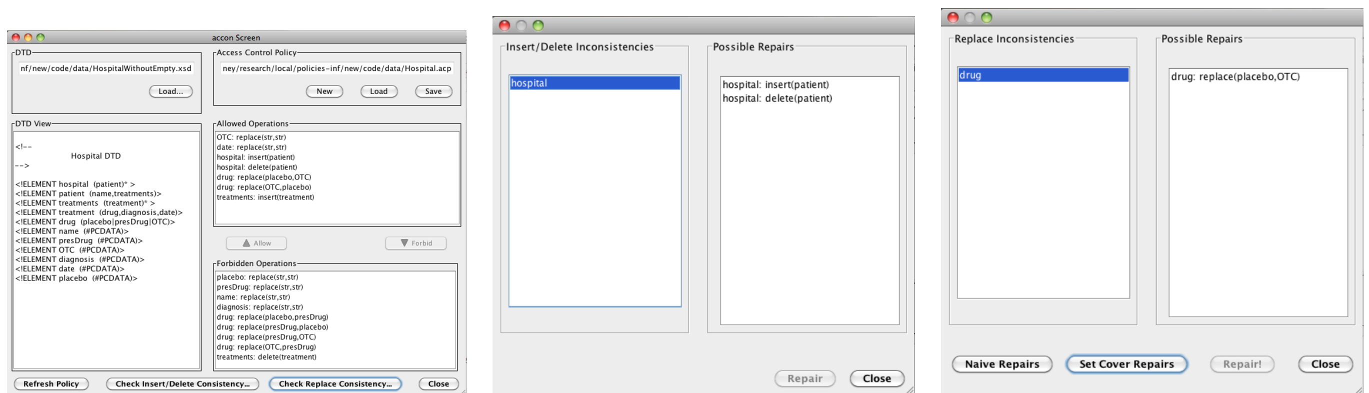

Fig. 13 GUI interface to ACCon

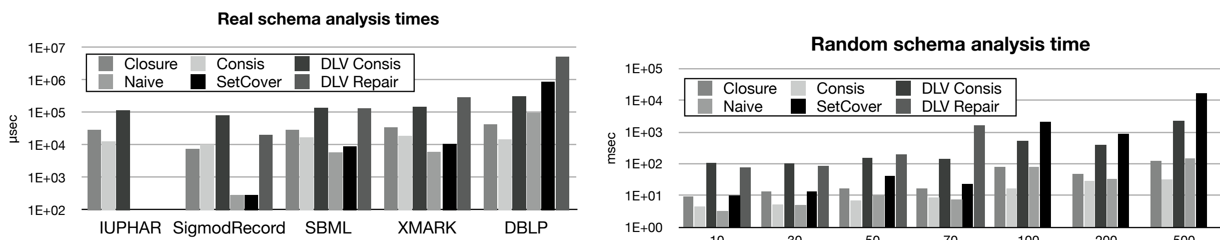

Fig. 14 Closure, consistency and repair times for real and random policies. Times are averaged over five random policies ($p = 0.5$). DLV repair did not finish within 5 seconds for larger random schemas. The left-hand figure is in microseconds (μsec) while the right-hand figure is in milliseconds (msec). Both are in logarithmic scale.

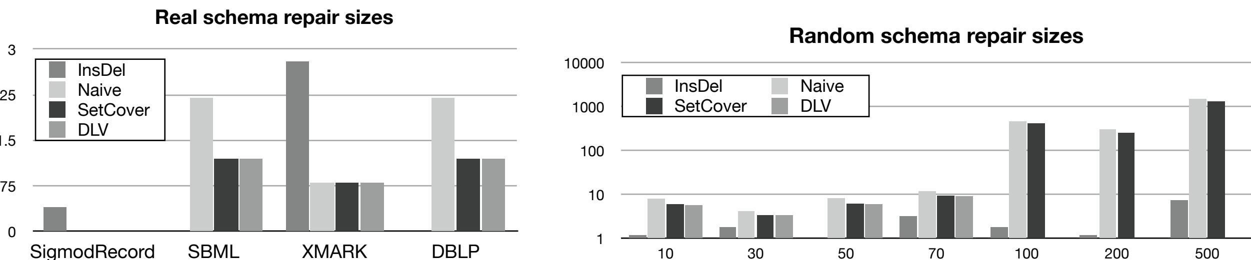

Fig. 15 Analysis time and repair sizes for random schemas. Repair sizes are averaged over five random policies ($p = 0.5$). Repair sizes for DLV exclude insert–delete repairs. DLV repair did not finish within 5 seconds for larger random schemas.

repair algorithm is also always faster than the MSCP-based algorithm, sometimes by more than an order of magnitude. Likewise, MSCP-based repair is always faster than DLV, often by over an order of magnitude.

6.2.2 Repair size

We measured the sizes of insert–delete, naive replace, and MSCP-based replace repairs found by the respective algorithms. We also measured the sizes of minimal repairs found by DLV, subject to a timeout of 60 seconds. The results are summarized in Figure 15. Both figures report the number of replace repairs found by DLV, since the number of insert–delete repairs found by either DLV or our algorithms is the same. Repair sizes are omitted for DLV on larger random policies where DLV timed out.

For real schemas, the number of repairs needed for a random policy was typically small: on average, less than three insert–delete repairs and less than two replace repairs. For all of the real schemas, the MSCP-based algorithm found a minimal repair, as of course DLV did. The naive repair was often significantly larger.

For random schemas, the number of insert–delete repairs was typically small, even for larger policies: on average there were under 10 insert–delete repairs. However, the number of replace repairs tends to grow as the size of the policy increases. Again, the naive repair is often significantly larger than the minimal repair found by DLV (when possible), and usually significantly larger than the MSCP-based repair. The MSCP-based repair is often not minimal but usually close to the minimal one found by DLV.

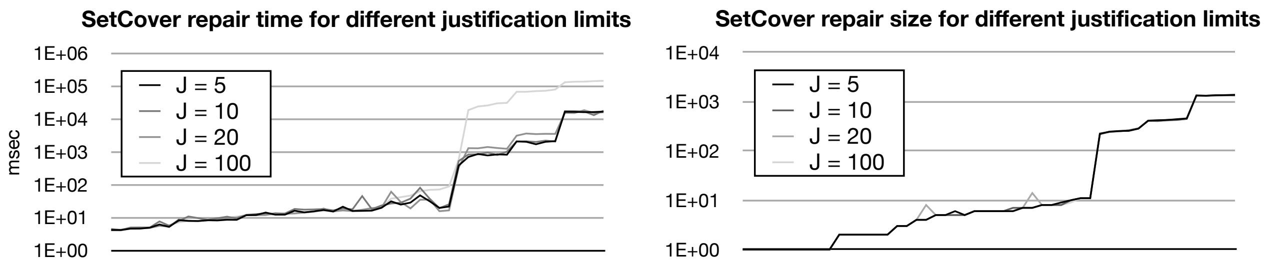

Fig. 16 Evaluation of the effect of justification limit. The figures report on 52 trials involving random and real policies with replace inconsistencies and $\mathfrak{J} \in \{5, 10, 20, 100\}$. The trials are sorted by their time or repair size for $\mathfrak{J} = 100$.

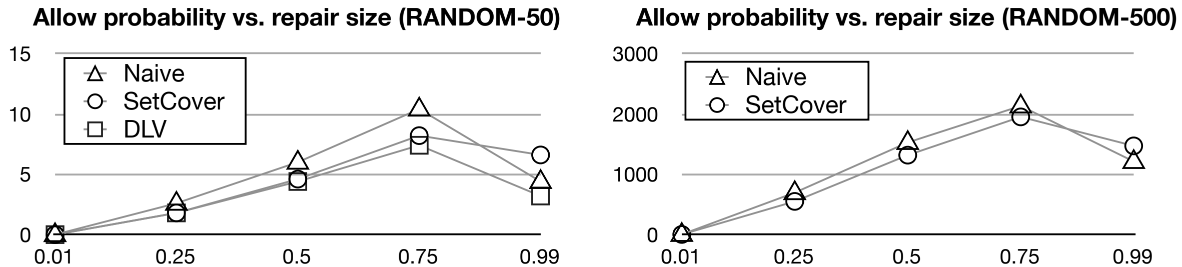

Fig. 17 Variation in repair sizes relative to probability p of allowing each UAT. Figures are averaged over five random policies ($p \in \{0.01, 0.25, 0.5, 0.75, 0.99\}$. Note that policies with all privileges allowed or denied are trivially consistent.

6.2.3 Varying justification limit

Recall that the $\mathbf{ReplaceSetCover}$ algorithm has a parameter $\mathfrak{J}$ limiting the number of justifications to consider per edge. The above experiments used a justification limit $\mathfrak{J} = 10$. To evaluate the sensitivity of the algorithm to this justification limit, we measured the repair times and policy sizes for 52 trials involving random policies for $\mathfrak{J} \in \{5, 10, 20, 50\}$. We obtained these trials by discarding those policies reported above that had no replace inconsistencies. Figure 16 shows the repair times and repair sizes for the different justification limits. These are sorted by the time or size for the largest justification limit $\mathfrak{J} = 100$; thus, large deviations from the performance for $\mathfrak{J} = 100$ are readily visible. Both plots use a logarithmic scale, so only large variations are visible.

These results suggest that increasing $\mathfrak{J}$ only has an effect for large policies, and this effect is to increase running time significantly, while decreasing repair size modestly. Thus, a choice of $\mathfrak{J} = 10$ or smaller is reasonable.

6.2.4 Varying allow probability

Our final experimental results concern the relationship between the probability p used to generate random policies and the degree of inconsistency of a policy. The relationship between p and the minimal repair size is hard to predict, since a policy must include both allowed and forbidden UATs in order to be inconsistent. We experimentally evaluated the repair sizes found by naive, MSCP, and DLV-based repair algorithms. The results are summarized in Figure 17.

We observe a peak in the number of inconsistencies for $p = 0.75$. For RANDOM-50, we can calculate the minimal repair exactly using DLV and it follows this pattern. For RANDOM-500, DLV times out, but the same pattern holds for the approximate algorithms. Interestingly, both sets of results also show that for policies with few forbidden UATs ($p = 0.99$), the naive repair is often smaller than the MSCP-based repair. This suggests that for policies with few forbidden UATs, it the naive algorithm may be better.

6.3 Discussion

Our experiments show that consistency checking, partial policy completion and policy repair can be implemented efficiently, compared to a generic implementation using an external solver such as DLV. Our algorithms often return minimal repairs for small examples, while providing answers much faster than the generic DLV solver on larger examples. Our algorithms also run fast enough to be useful in an interactive policy editor for realistic schemas and policies.

Our experiments varying $\mathfrak{J}$ show that a small value of $\mathfrak{J}$ is reasonable since increasing the justification limit slows down repair for larger policies while not finding significantly smaller repairs. Our experiments varying p (the probability show that repair size peaks near $p = 0.75$), and suggest that the naive repair strategy is preferable for policies in which most operations are already allowed, since it is much faster and produces a smaller repair for $p = 0.99$.

For many applications, the relative ease of implementation using DLV (or any other Datalog implementation)

would be a substantial advantage, but to repair larger policies, our algorithms offer superior performance.

7 Related Work

De Capitani di Vimercati et al. [10] give an overview of access control techniques for XML, focusing on enforcement mechanisms. Consistency of XML update access control policies was first considered by Fundulaki and Maneth [13], who showed the problem is undecidable for policies defined using XPath path expressions in the absence of a schema. They introduced the schema-based policy model XAcU_{annot} which we have extended in this article.

Cautis, Abiteboul and Milo in [7] 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 [2] the set cover problem was used to find repairs of databases *w.r.t.* denial constraints.

Moore [22] studies the problem of whether an XML document can be generated by a sequence of operations allowed by an access control policy. He shows that this problem is undecidable in general, and identifies some decidable special cases (such as monotonically increasing updates).

Jacquemard and Rusinowitch [15] model XML updates using rewriting rules and apply techniques from tree automata to prove decidability and (mostly) undecidability results for policy consistency in the absence of a schema. They give a PTIME algorithm for consistency with respect to a schema, similar to Theorem 1, but do not investigate partial policies or the repair problem. They also show that consistency is undecidable if the allowed sequence can temporarily violate the schema. They do not consider the repair problem, which is likely more difficult in the absence of a schema that constrains the search space of policy changes.

Boneva et al. [4] study the problem of safely propagating updates to XML security views back to uniform, side-effect free updates to the original source document. This technique is complementary to our approach. Their technique assumes that the user is allowed to update any data in the view, but may constrain the updates that can be performed on the source. Our update access control policies can be used to make some parts of the view read-only, and our algorithms can then check that the view policy is consistent. It may be interesting to study the problem of translating policies expressed on the view to policies on the underlying source data (or vice versa) or checking that allowed view updates translate to allowed souce updates.

8 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 article we have studied such *inconsistency* vulnerabilities and shown how to check consistency and repair inconsistent policies. We also considered consistency and repair problems for partial policies that are more convenient to write since many privileges may be left unspecified. We showed that the repair problem is intractable, but developed approximate algorithms and showed that they yield reasonable results in practice. Finally, we evaluated the algorithms and showed that the consistency and insert/delete repair algorithms are fast in practice, while the approximate replace repair algorithms are more computationally intensive but still much faster than finding exact solutions using a state-of-the-art answer-set programming solver.

There are a number of possible directions for future work, including 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 developing consistent policy languages based on larger classes of DTDs or schemas.

Acknowledgments: We would like to thank Sebastian Maneth and Floris Geerts for insightful discussions and comments. We also thank Francesco Ricca (University of Calabria) for help with the DLVWrapper library. Loreto Bravo acknowledges support from FONDECYT through grant 11080260 and CONICYT through grant PSD-57. Cheney is supported by a Royal Society University Research Fellowship and this work was supported in part by the UK Engineering and Physical Sciences Research Council and a Google Research Award.

References

1. Arenas M, Bertossi L, Chomicki J (1999) Consistent Query Answers in Inconsistent Databases. In: PODS, ACM Press, pp 68–79
2. Bertossi L, Bravo L, Franconi E, Lopatenko A (2008) The complexity and approximation of fixing numerical attributes in databases under integrity constraints. Inf Syst 33:407–434

3. Bex GJ, Neven F, Schwentick T, Vansummeren S (2010) Inference of concise regular expressions and DTDs. ACM Trans Database Syst 35:11:1–11:47
4. Boneva I, Caron AC, Groz B, Roos Y, Tison S, Staworko S (2011) View update translation for XML. In: ICDT, pp 42–53
5. Bravo L, Cheney J, Fundulaki I (2007) Repairing inconsistent XML write-access control policies. In: DBPL, Springer-Verlag, Vienna, Austria, no. 4797 in LNCS, pp 98–112
6. Bravo L, Cheney J, Fundulaki I (2008) ACCOn: Checking consistency of XML write-access control policies. In: Proceedings of the 11th International Conference on Extending Database Technology (EDBT 2008), pp 715–719, demonstration.
7. Cautis B, Abiteboul S, Milo T (2009) Reasoning about xml update constraints. J Comput Syst Sci 75(6):336–358
8. Chvatal V (1979) A Greedy Heuristic for the Set Covering Problem. Mathematics of Operations Research 4:233–235
9. Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to Algorithms. MIT Press and McGraw-Hill
10. De Capitani di Vimercati S, Foresti S, Paraboschi S, Samarati P (2008) Access control models for XML. In: Gertz M, Jajodia S (eds) Handbook of Database Security, Springer US, pp 27–53
11. Fan W, Chan CY, Garofalakis M (2004) Secure XML Querying with Security Views. In: ACM SIGMOD, pp 587–598
12. Floyd R (1962) Algorithm 97: Shortest path. Communications of the ACM 5(6):345
13. Fundulaki I, Maneth S (2007) Formalizing XML access control for update operations. In: Proceedings of the 12th ACM symposium on Access control models and technologies, ACM, New York, NY, USA, SACMAT '07, pp 169–174
14. Harmar A, Hills R, Rosser E, Jones M, Buneman O, Dunbar D, Greenhill S, Hale V, Sharman J, et al TB (2009) IUPHAR-DB: the IUPHAR database of G protein-coupled receptors and ion channels. Nucleic Acids Res 37:D680–5
15. Jacquemard F, Rusinowitch M (2010) Rewrite-based verification of XML updates. In: PPDP '10: Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming, ACM, New York, NY, USA, pp 119–130
16. Jiang M, Fu AWC (2005) Integration and efficient lookup of compressed XML accessibility maps. IEEE TKDE 17(7):939–953
17. Koromilas L, Chinis G, Fundulaki I, Ioannidis S (2009) Controlling access to XML documents over XML native and relational databases. In: Jonker W, Petkovic M (eds) Secure Data Management, Springer, Lecture Notes in Computer Science, vol 5776, pp 122–141
18. Kuper G, Massacci F, Rassadko N (2005) Generalized XML security views. In: Proceedings of the tenth ACM symposium on Access control models and technologies, ACM, New York, NY, USA, SACMAT '05, pp 77–84
19. Leone N, Pfeifer G, Faber W, Eiter T, Gottlob G, Koch C, Mateis C, Perri S, Scarcello F (2006) The DLV System for Knowledge Representation and Reasoning. ACM Trans on Comp Logic 7(3):499–562
20. Lim CH, Park S, Son SH (2003) Access control of XML documents considering update operations. In: ACM Workshop on XML Security, pp 49–59
21. Martens W, Neven F, Schwentick T, Bex GJ (2006) Expressiveness and Complexity of XML Schema. ACM Trans Database Syst 31(3):770–813
22. Moore N (2011) Computational complexity of the problem of tree generation under fine-grained access control policies. Inf Comput 209(3):548–567
23. Murata M, Tozawa A, Kudo M, Hada S (2006) XML Access Control Using Static Analysis. ACM TISSEC 9(3):290–331
24. Papadimitriou C (1994) Computational Complexity. Addison-Wesley
25. Robie J, Chamberlin D, Dyck M, Florescu D, Melton J, Simeon J (2011) XQuery Update Facility 1.0. http://www.w3.org/TR/xquery-update-10/, W3C Recommendation
26. Saltzer J, Schroeder M (1975) The protection of information in computer systems. Proceedings of the IEEE 63(9):1278–1308
27. Schmidt A, Waas F, Kersten M, Carey MJ, Manolescu I, Busse R (2002) XMark: a benchmark for XML data management. In: VLDB, pp 974–985
28. Stoica A, Farkas C (2002) Secure XML Views. In: IFIP WG 11.3, Kluwer, vol 256
29. Yannakakis M (1978) Node-and Edge-deletion NP-complete Problems. In: STOC, ACM Press, pp 253–264
30. Yannakakis M (1981) Edge-Deletion Problems. SIAM Journal on Computing 10(2):297–309
31. Yu T, Srivastava D, Lakshmanan LVS, Jagadish HV (2004) A Compressed Accessibility Map for XML. ACM Trans Database Syst 29(2):363–402