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

Gabriel Kuper E-mail: kuper@acm.org ·

Fabio Massacci E-mail: fabio.massacci@unitn.it ·

Nataliya Rassadko E-mail: rassadko@dit.unitn.it

Generalized XML Security Views

Abstract We investigate a generalization of the notion of XML security view introduced by Stoica and Farkas [45] and later refined by Fan et al. [23]. The model consists of access control policies specified over DTDs with XPath expressions for data-dependent access control. We provide the notion of *security views* characterizing information accessible to authorized users. This is a transformed DTD schema that can be used by users for query formulation. We develop an algorithm to materialize an authorized version of the document from the view and an algorithm to construct the view from an access control specification. We show that our view construction combined with materialization produces the same result as the direct application of the DTD access specification on the document. We also propose a number of generalizations of possible security policies and show how they affect view construction algorithm. Finally, we provide an evaluation of our system.

Keywords XML, security view, XPath, materialization, authorized view

1 Introduction

XML [12] has become the prime standard for data representation and exchange on the Web. In light of the sensitive nature of many business data applications, this also raises the important issue of security

The University of Trento

via Sommarive, 14

Povo (Trento), 38050

Italy

in XML and the selective exposure of information to different classes of users based on their access privileges.

To address such security issues, there is a need for a generic, flexible *security model* that can effectively support multiple policies for controlling access to XML content at various levels of granularity. Perhaps even more importantly, enforcing such access-control policies should not imply any drastic degradation in either performance or functionality for the underlying XML query-execution engine. In addition, access-control enforcement should not complicate the maintenance of the consistency and integrity of the data when either the XML data or the access policies are updated.

An XML security model must support:

1. a simple and powerful fine grained authorization mechanism that can control access to both content and structure (e.g., restricting access to entire subtrees or specific elements in the document tree based on their content or location);
2. efficient mechanisms for the enforcement of security policy without fully annotating the underlying document to decouple data management from policy management;
3. schema information, characterizing all and only those elements accessible to each authorized user, in the same way that a relational database offers security views to their users.

An access control should not inhibit *schema availability*, i.e., the availability of necessary schema information (e.g., DTD [12] or an XML Schema [22]) specifying the structure of the accessible data. Furthermore, XML documents are typically accompanied by a schema that specifies the internal structure of the data. For the same reasons that a database schema is needed for query formulation and processing for traditional databases, XML schemas are also important for XML query formulation and optimization; and schemas are critical for XML data exchange and integration [4].

While specifications and enforcement of access control are well understood for traditional databases [21], [36], [42], [44], the study of security for XML is less established. Early security models that have been proposed for XML do not meet criteria 3 above and, to a lesser extent, criterion 1 and 2. In particular, cryptographically enforced access control to XML documents [38], [10] considers only protection of XML data but not a schema; different parts of the XML tree are encrypted by different keys (typically one key for a particular combination of access control rules applicable to an XML element); finally, these kinds of protection make querying difficult since a large amount of decryption is needed. Run-time policy evaluation scenarios [33], [10], requires policy propagation from the root to the requested node. This may result in calculation of accessibility decisions for every node test of the user query. Such a time consuming query answering can be improved only partly in the presence of DTD [40], [14], but it happens not for all queries. Other optimization techniques rely on compressed accessibility map [52], [31], [53], or combining access control policy with user queries [37]. However, the user still does not have the schema of accessible data. This problem was slightly resolved in [9], [17], where a "loosened" variant (i.e., DTD where every forbidden element is "optional") of the schema is delivered. The problem with the loosened version is that it still reveals all element names, including sensitive ones, of the initial DTD. Moreover, the overall security policy enforcement is performed at the document level by fully annotating the entire XML document/database; this requires possibly expensive view materialization (see Sec. 10), and complicates consistency and integrity maintenance.

In addition, fixing the access control policies at the instance level without providing or computing a schema makes it difficult for the security officer to understand how the authorized view of a document for a user or a class of users actually looks like. Hence, such a solution is hardly practical for large and complex documents. Revelation of excessive schema information might lead to security breaches: an unauthorized user can deduce or infer confidential information via multiple queries (essentially if the authorization specifications are not closed under intersection) and analyze the schema even if only accessible nodes are queried.

To overcome the limitations outlined above, the notion of an XML security views was initially proposed by Stoica and Farkas [45] and later refined by Fan et al. [23]. The basic idea is to provide a schema that describes the data that can be seen by the user, as well as a (hidden) set of XPath expressions that describes how to compute the data in the view from the original data.

1.1 Our Contribution

This paper is an extension of our previous work [34]. In this section, we list our contribution not only w.r.t. the existing proposals of an XML access control research, but also w.r.t. [34].

Our first contribution is an investigation of different alternatives for policy definition and enforcement at the level of an XML tree (Sec. 4). Our analysis shows that not all combinations of policy options satisfy the properties of *completeness* and *consistency*, i.e. some policy settings do not result in a single fully annotated tree. We provide a classification of policies using different options of security label propagation and conflict resolution. The comparison of our policy framework with those of the existing proposals is done in Sec. 11.

Second, we generalize the notion of XML security views from [23], [45] to arbitrary DAG DTDs and to conditional constraints defined in a very expressive XPath fragment. For each view, a security specification is a simple extension of the DTD document D with security annotations and security policies exploited to automatically obtain a full annotation from a partial one. The security views proposed in [45] in some sense are smarter than ours since they can preserve semantic associations among some XML tags. For this purpose, corresponding cover stories are selected by a security officer, but they need a human intervence. However, neither full DTD labelling derivation nor querying is discussed in [45].

Our third contribution is related to view derivation algorithm. Namely, we show a generic algorithm that constructs a fully annotated DTD D_F (from the partial security specification) for different policies so that D_F reflects a full annotation of a corresponding XML document. From this full specification, we derive a security view $\mathcal{S}$ consisting of a *view DTD* D_v and a function σ defined via XPath queries. The view DTD D_v shows only the data that is accessible according to the specification. The view is provided to the users so that they can formulate their queries over the view. The function σ is withheld from the users, and is used to extract accessible data from the actual XML documents to populate a structure conforming to D_v. The formal proof of correctness of view derivation algorithm is also given.

In contrast to [23], we consider general XML DTDs defined in terms of regular expressions rather than normalized DTDs. Furthermore, we do not allow dummy element types in the definition of security views. The latter are equivalent of optional elements in a loosened DTD of [17]. In addition, algorithm presented in [23] is only top-down with a different semantics for qualifier, while our solution can compute views both for top-down and for bottom-up policies. Next, Fan et al. did not show how to construct fully annotated DTD D_F. The latter can be extensively exploited, for example, in run-time

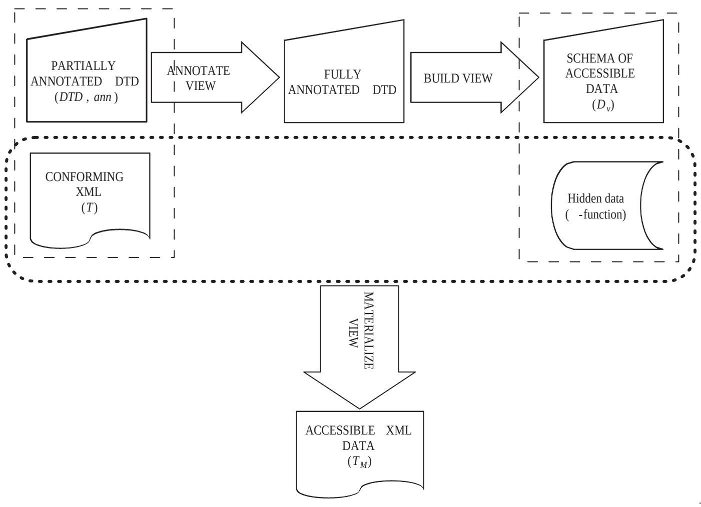


Fig. 1 Schema of materialization of accessible data

policy evaluation scenarios when D_F is calculated for a requesting user once for session and all further accessibility checks are done on D_F.

Finally, in [23], it was claimed that view materialization is not needed since it is time-consuming and can be avoided by means of query rewriting technique. However, in many applications, views can remain unchanged for a long period of time and, hence, can be queried directly without query rewriting. Moreover, query rewriting may introduce complicated qualifiers, and thus evaluation of them may lead to exponential response time. Guided by these observations, we provide an algorithm for view materialization. Our approach is depicted summarily in Fig. 1. We also show that materialized XML views conform to D_v and are isomorphic to authorized ones. This is our fourth contribution.

Finally, as the fifth contribution, we provide an experimental evaluation of our system.

1.2 Plan of the paper

The paper is organized as follows. First we present preliminary notions on XML and XPath in Sec. 3. In Sec. 2 we provide a motivating example. After classification of security policies with respect to consistency and completeness properties in Sec. 4, we provide a formalization of the motivating example for top-down policy in Sec. 5, followed by the algorithm of view materialization in Sec. 6, and DTD view derivation in Sec. 7. An extension of view construction algorithm is given in Sec. 8. Next, investigation on theoretical aspects of our algorithms are provided in Sec. 9 and evaluation of the system is done in Sec. 10. Related work is reviewed in Sec. 11. Finally, we conclude the paper in Sec. 12.

2 A Motivating Example

We start with a running example assuming an intuitive understanding of XML documents as trees and DTDs as DAGs.

Example 1 We describe a DTD database containing the information on *applications* for PhD/MS program. Each application is initiated by a student described via *student-data* with an element *id* uniquely identifying the student and representing the student's login name. *Student-data* is composed of *name*, desired *degree* (PhD or MS) *department*, and *waiver*. The latter field may take values "true" or "false"[1] and means that student does (does not) waive his/her right to inspect the content of the recommendation letters. The application is supported by several letters of recommendation (*recomm-letter*); some of them can be classified as for *letter* body and is provided by a separate *evaluator* having *name*, *title* and *institution* attributes. The evaluator places comments on the applicant's skills in *free-text* field, which is either a *PDF* or a *TXT* file, and rates applicant's *English* proficiency, and possible contributions to *PhD* or *MS* program. Letters of recommendation are reviewed by the admission committee and are assigned to a category *favorable* or *unfavorable* depending on the context.

The corresponding DTD graph is depicted on Fig. 2, where solid lines represent concatenation (i.e., AND-relation between a node and its children), dashed lines represent disjunction (i.e., OR-relation), and combined line, called *nesting* represent mix of relations between a node and its child nodes (in Fig. 2, `rating` has child `English` and either `MS` or `PhD`). Stars on lines represent zero-to-many cardinality between a node and its child of a certain type. Fig. 3(a) shows an XML document *conforming* to this DTD, where boxed leaves of the tree represent text values. Gray labels on some edges of this XML will be explained later.

The need to provide users with a schema-level security view is illustrated by the access control requirements in Example 2.

[1] A domain of some value, like *waiver*, may be restricted by means of `ENTITY` declaration of DTD and is not considered in the current thesis.

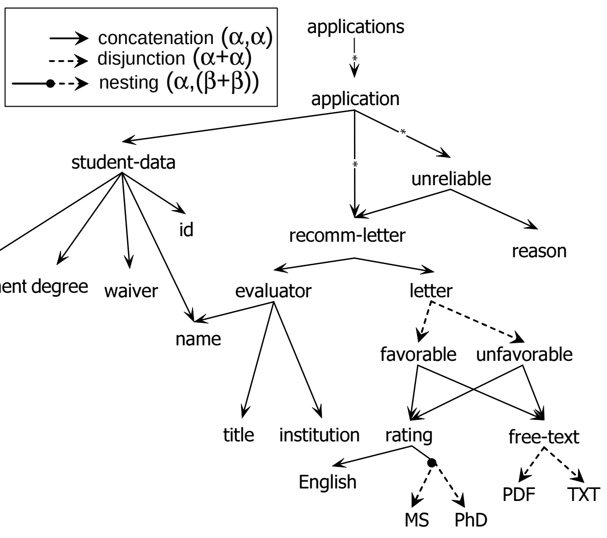
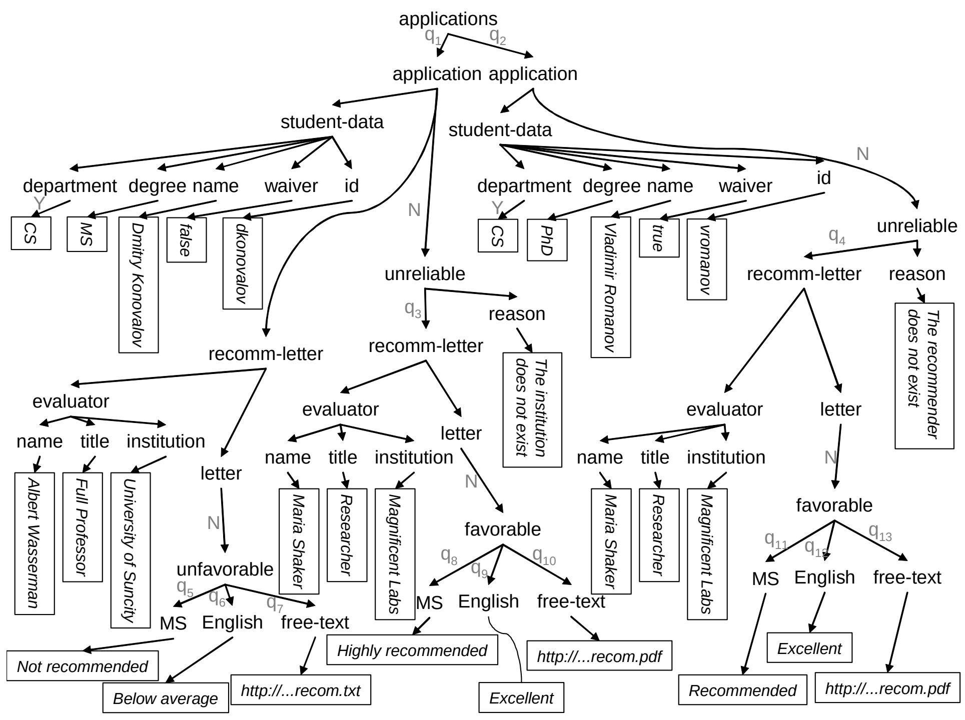


Fig. 2 The graph representation of the DTD document D



(a) Initial XML document

$q_1, q_2 \doteq$ `student-data/id` $=$ `$login`

$q_3, q_4 \doteq ../../$`student-data/id` $=$ `$login`;

$q_5, \dots, q_{13} \doteq$ *ancestor*::`application/student-data[id = $login]/waiver` $=$"*false*";

(b) The meaning of qualifiers

Fig. 3 Partial annotation of the XML document conforming to the DTD from Example 1

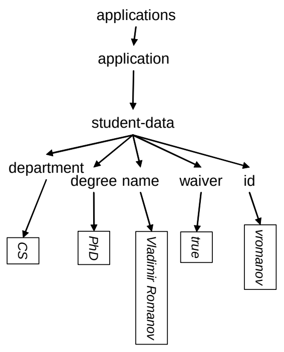
Example 2 An applicant can access his/her personal record located under the field `student-data`. Access to fields `favorable` and `unfavorable` is forbidden, while the visibility of `rating` and `free-text` is possible if the `waiver` is *false* (data-dependent access). Moreover, the applicant should not be aware of the reliability of the recommendation letters as the leakage of this information to recommenders might lead to diplomatic incidents. Finally, we require the field `department` to be visible in any case: either from a proper student's application or from external one. Thus, the student will be able to see how many applications are submitted to different departments of the university. Hence, a ranking of departments can be inferred.

Corresponding qualifiers are shown in Fig. 3(a) as gray labels on edges. The meaning of the labels is the following: Y means always accessible, N means always forbidden, while $q_1, \ldots, q_{13}$ are encoded in XPath and presented in Fig. 3(b). In these qualifiers, `$login` is a variable that is instantiated dynamically during the login into the system. So, if `$login` is "dkonovalov" then q_1, q_3, $q_5 - q_{10}$ are evaluated to true, q_2, q_4, $q_{11} - q_{13}$ are evaluated to false. If the value of `$login` is "vromanov" the situation is reverse.

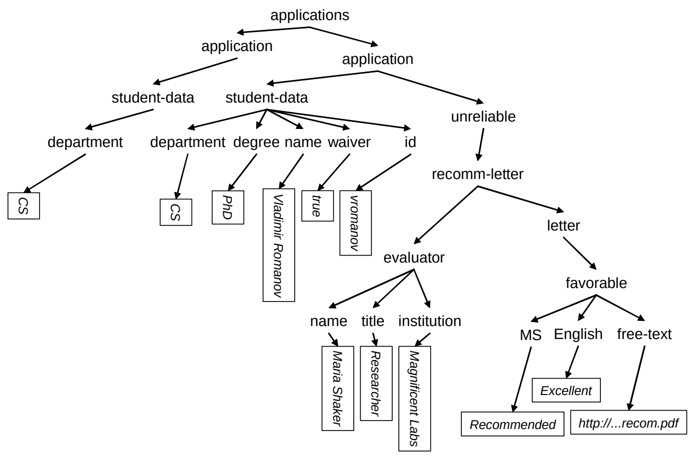
Existing cryptographical proposals like [38], [9] as well as run-time policy evaluation scenarios [33], [10], [40], [14], [37] and view-based approaches [9], [17] enforce such security constraints directly on the XML document. The DTD is usually used only for typing of the XML document with security labels which are then propagated over the entire document through corresponding document portions. These systems specify how to restrict access at the *data level* and how to obtain *authorized view* of data.

Example 3 Let us construct an authorized view for Vladimir Romanov using some other well-known access control models.

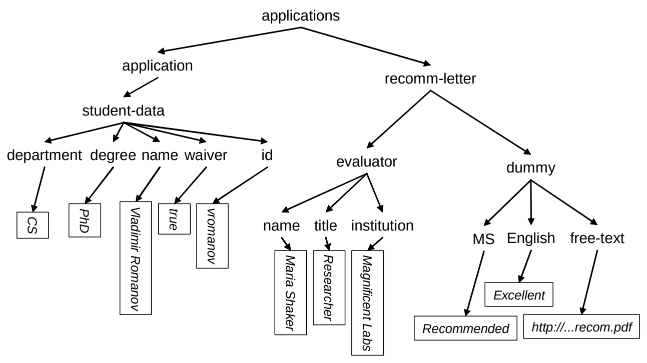
– **Author-X by Bertino et al. [7]** does not support qualifiers, so we assume that a preprocessing step of their algorithm evaluating qualifier with the following semantics: if q is evaluated to true, the label is Y (i.e. permitted for access), otherwise N (i.e. forbidden for access). In the case of Vladimir Romanov, q_1, q_3, $q_5, \ldots, q_{10}$ are evaluated to false, and q_2, q_4, $q_{11}, \ldots, q_{13}$ are evaluated to true. Next, we assume that the propagation option is `CASCADE` [7] (i.e., top-down to all subelements) and *deny takes precedence* in the case of conflicts. The approach also relies on *denial downward consistency* rule that deletes everything that is below Nlabeled node. The view for Vladimir Ro-



(a) Bertino et al. [7]



(b) Damiani et al. [17]

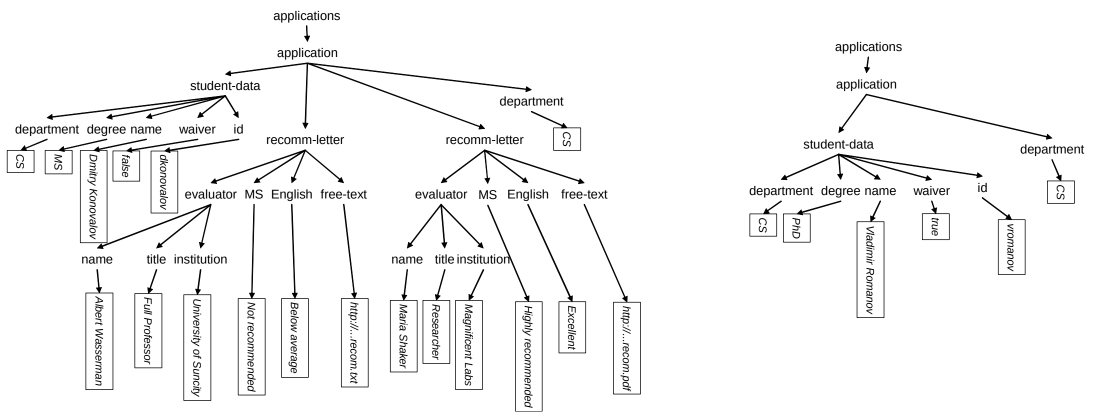


(c) Fan et al. [23]

Fig. 4 Authorized XML view for student Vladimir Romanov

manov is shown in Fig. 4(a). It contains only the student's personal information and looks fine. However, if Vladimir Romanov didn't wave his right to see the content of recommendation letters, he would't have an access to this information anyway, because $\mathsf{ann}(recomm - letter, letter)=\mathsf{N}$. **Selective dissemination** [9] is a crypto-version of the Author-X which promotes a *push architecture*. The difference is that negative authorizations are not supported. However, everything that is not permitted is forbidden which is, basically, local *closed* policy. This means that if some XML subtree rooted at node n cannot be decrypted by a set of keys owned by the user, neither can any of its subtrees. Hence, the same view can be extracted as in the previous case. Ranking among departments is missing because of denial downward consistency rule as well.

– **Access control processor by Damiani et al. [17].** The preprocessing step mentioned previously is already included in the algorithm. The rule *most specific takes precedence* (which simply



(a) Authorized XML view for student Dmitry Konovalov (b) and for student Vladimir Romanov

Fig. 5 Security annotation for competing student

says that an unlabelled node takes the security label of its first labelled ancestor/descendant) is used for resolving conflicts. Finally, forbidden nodes are deleted if they do not have permitted children, otherwise only their attributes are cleaned. Such a policy enforcement results in Vladimir's view depicted in Fig. 4(b). Sensitive field `unreliable`is visible because it has the permitted child `recomm-letter`. If the waiver had value *true*, sensitive fields `favorable` and `unfavorable` would be revealed as well.

– **Security views by Fan et al. [23].** Although [23] does not discuss XML view materialization, we tried to emulate their schema-level enforcement at the XML instance. The view for Vladimir Romanov is the same as in the first case. If we imagine that Vladimir Romanov does not waive his right to see the content of recommendation letters, the XML view presented in Fig. 4(c) has the element `dummy` that may suggest to the user an idea that something is hidden. Ranking among departments is still missing.

As we have seen above, none of these methods produces a correct XML view. Author-X hides too much information that should have been available for the user and sometimes reveals sensitive information in a meaningless way. The access control processor by Damiani et al., on the contrary, reveals too much information. The last methods is better, but some information is still missing due to inappropriate qualifier semantics that deletes a subtree rooted at a node where the qualifier does not hold.

Example 4 Fig. 5(a) and Fig. 5(b) are the authorized views retrieved from the document in Fig. 3(a). In particular, Dmitry Konovalov has login `dkonovalov` and does not waive his right to see recommendation letters supporting his application (i.e., `waiver`=*false*), while Vladimir Romanov has login `vromanov` and `waiver`=*true* in his case. Both users also may infer the ranking among departments of the university since their views include `department` elements of all the other applications.

An important question remains unanswered: what schema information should be provided to the user? To formulate and process queries, the user needs a schema describing the accessible data. One solution, suggested by Damiani et al. [17], is to *loosen* the original DTD (make forbidden nodes optional). In some cases, it is unacceptable to expose even the loosened DTD to final user. To illustrate this, consider two permissible XPath queries about a letter of recommendation:

Q_1 : `/applications/application//evaluator`

Q_2 : `/applications/application/recomm-letter/evaluator`

The query Q_1 finds all elements of type `evaluator` that are associated with recommendation letter (including unreliable ones), while Q_2 returns only `evaluator`s of reliable `recomm-letter`s. Although most of the unreliable data is hidden, a look at the DTD document allows one to infer which letters are considered as unreliable: the `evaluators` in Q_1 that are not returned by Q_2; thus, we have a security breach. This is because all evaluators are visible, but in different ways.

3 A Primer on DTD, XML and XPath

We first review DTDs (Document Type Definitions [12]) and XPath [15] queries. It is well-known, that non-recursive DTDs may be modelled as a DAG.

Definition 1 A DTD D is a triple (Ele, P, `root`), where Ele is a finite set of *element types*; `root` is a distinguished type in Ele called "root", and P is a function defining element types such that for each A in Ele, $P(A) = \alpha$, where α is a regular expression, defined as follows:

$$\alpha := \texttt{str} \mid Ele \mid \epsilon \mid \alpha + \alpha \mid \alpha , \alpha \mid \alpha *$$

where `str` is a special type denoting PCDATA, ϵ is the empty word, and "+", ",", and "$*$" denote disjunction (or choice container), concatenation (or sequence container), and the Kleene star, respectively. We call $A \to P(A)$ as the *DTD production rule* of A. For all element types B occurring in $P(A)$, we refer to B as a *child type* of A and to A as a *parent type* of B.

In the spirit of [45], our DTD definition allows mixed containers (e.g., choice container may include sequence subcontainer, etc) and do not require any normal form as in [23], where it was claimed that any DTD may be normalized. However, no algorithm on normalization/unnormalization were provided [2].

Example 5 According to Def. 1, the formal representation of the database from Example 1 is the following. The DTD D is defined as (Ele, P, db), where

db = `applications`

Ele = {`applications, application, student-data, department, degree, waiver, name, recomm-letter, evaluator, title, institution, id, letter, rating, English, MS, PhD, free-text, PDF, TXT, unreliable, reason, favorable, unfavorable`};

and the function P is the following (we omit the definition of elements whose type is `str`):

```
P(applications)    = (application*)
P(application)     = (student-data, recomm-letter*, unreliable*)
P(student-data)    = (department, degree, waver, name, id)
P(recomm-letter)   = (evaluator, letter)
P(evaluator)       = (name, title, institution)
P(letter)          = (favorable+unfavorable)
P(favorable)       = (rating, free-text)
P(unfavorable)     = (rating, free-text)
P(rating)          = (English, MS+PhD)
P(free-text)       = (PDF+TXT)
P(unreliable)      = (recomm-letter, reason)
```

An XML document is typically modelled as a node-labelled tree.

[2] Although we may assume that normalization can be carried via additional intermediate nodes (e.g., `rating` has two children: `English` and a normalization node `normalizing_node` with two children `PhD` and `MS`), the role of these nodes during policy enforcement, DTD view querying and query rewriting presented in [23] is not clear.

Definition 2 [23] An XML tree T *conforms to* a DTD D iff

1. the root of T is the unique node labelled with `root`;
2. each node in T is labelled either with an *Ele* type A, called an A *element*, or with a value of type `str`, called a *text node*;
3. each A element has a list of children of elements and text nodes such that their labels form a word in the regular language defined by $P(A)$;
4. each text node carries a `str` value and is a leaf of the tree.

We call T an *instance* of D if T conforms to D.

Next, we consider a class of XPath queries that corresponds to the CoreXPath of Gottlob et al. [27] augmented with the union operator and atomic tests and which is denoted by Benedict et al. [5] as $\mathcal{X}$.

The XPath axes we consider as primitive are `child`, `parent`, `ancestor-or-self`, `descendant-or-self`, `self`. Gottlob et al. [27] show how the semantics of such axes can be computed in polynomial time. In the sequel we denote by θ one of the primitive axes and by θ^{-1} its inverse. Notice that each primitive axis has its inverse within the same set of primitives. For instance, $\texttt{descendant-or-self}^{-1} = \texttt{ancestor-or-self}$.

Definition 3 An XPath expression in $\mathcal{X}$ is defined by the following grammar:

$$\begin{aligned}
\langle xpath\rangle &::= \text{`/`?}\,\langle path\rangle \;\mid\; \langle path\rangle\,(\text{`}\cup\text{`}\,\langle path\rangle)* \\
\langle path\rangle &::= \langle step\rangle\,(\text{`/`}\,\langle step\rangle)* \\
\langle step\rangle &::= \langle test\rangle \;\mid\; \langle test\rangle\,(\text{`[`}\,\langle qual\rangle\,\text{`]`})* \\
\langle test\rangle &::= \theta\text{`}::\text{`}A \;\mid\; \theta\text{`}::*\text{`} \\
\langle qual\rangle &::= \langle path\rangle\;\, op\; c \;\mid\; \langle qual\rangle\; \texttt{and}\; \langle qual\rangle \;\mid \\
&\qquad \langle qual\rangle\; \texttt{or}\; \langle qual\rangle \;\mid\; \texttt{not}\; \langle qual\rangle \;\mid\; \text{`(`}\,\langle qual\rangle\,\text{`)`}
\end{aligned}$$

where θ stands for an axis, c is a `str` constant, A is a label, *op* stands for one of $=$, $<$, $>$, $\leq$, $\geq$. The result of the *qual* filtering is called *qualifier* and is denoted by q. We denote by $\mathcal{X}_{NoTest}$ the fragment built without the $\langle path\rangle$ *op* c test.

For the sake of readability, we ignore the difference between *xpath* and *path*; we denote both with p. We also abbreviate `self` by ϵ, `child` $:: A/p$ with A/p, `descendant-or-self` $:: A/p$ by $//A/p$, and $p = p_1/p_2$ with $p_2 = //p_2'$ is written p as $p_1//p_2'$. The parent axis is also abbreviated as $../$.

The semantics of XPath is obtained by adapting to our fragment the $\mathcal{S}_{\rightarrow}$, $\mathcal{S}_{\leftarrow}$, $\mathcal{E}$ operators proposed by Gottlob et al. [27] and is identical to the proposal of Benedickt et al. [5]. Intuitively $\mathcal{S}_{\rightarrow}[|p|](N)$ gives all nodes that are reachable from a node in N using the path p. The $\mathcal{S}_{\leftarrow}[|p|]$ functions gives all nodes from which a path p starts to arrive at the queried node. The $\mathcal{E}[|q|]$ function evaluates qualifiers and returns all nodes that satisfy q.

For the sake of readability we overload the θ-symbol to stand for both the semantics and the syntax of axes. So, given a set of nodes N of a document T we have $\theta(N) = \{m \mid n\,\theta\,m \text{ for } n \in N\}$. In other words, $\theta(N)$ returns the nodes that are reachable according to the axis from a node in N. By $\mathcal{T}(A)$ we denote the set of nodes that have element type A. By $\mathcal{T}(*)$ we denote all nodes of a document. By $\mathcal{O}[|c|](N)$ we denote the function that returns all nodes of a set $\mathcal{T}(N)$ whose `str` value is *op c*.

The semantics of the other operators are shown in Fig. 6.

4 Document Access Control

Definition 4 Let (T, ann) be an authorization specification, where T is an XML tree, ann is an annotation of T with Y, N, and $\mathsf{Q}[q]$ expressed in XPath fragment from Def. 3. The *authorized version* T_A of T according the authorization specification is obtained from T as follows:

1. Evaluate qualifiers top down starting from the root and replace annotations by Y or N depending on the result;
2. For each unlabelled node, label it with
   – the annotation of its nearest labelled ancestor; or
   – the annotation of its nearest labelled descendants applying value conflict resolution policy; or
   – the *local propagation* value;
3. Delete all nodes labelled with N from the result, making all children of a deleted node v into children of v's parent.

The annotation of the document, before deleting nodes in the last step, is called the *full annotation* of T.

We present different policies to extend a partial annotation of the XML document to a full one. There are a number of alternatives. Although top-down propagation is considered to be the most

$$
\begin{aligned}
\mathcal{S}_{\rightarrow}[|\theta :: \lambda|](N) &= \theta(N) \cap \mathcal{T}(\lambda)\\
\mathcal{S}_{\rightarrow}[|/\theta :: \lambda|](N) &= \theta(\texttt{root}) \cap \mathcal{T}(\lambda)\\
\mathcal{S}_{\rightarrow}[|p/\theta :: \lambda|](N) &= \theta(\mathcal{S}_{\rightarrow}[|p|](N)) \cap \mathcal{T}(\lambda)\\
\mathcal{S}_{\rightarrow}[|/p|](N) &= \mathcal{S}_{\rightarrow}[|p|](\{\texttt{root}\})\\
\mathcal{S}_{\rightarrow}[|p[q]|](N) &= \mathcal{S}_{\rightarrow}[|p|](N) \cap \mathcal{E}[|q|]\\
\mathcal{S}_{\rightarrow}[|p_1 \cup p_2|](N) &= \mathcal{S}_{\rightarrow}[|p_1|](N) \cup \mathcal{S}_{\rightarrow}[|p_2|](N)\\
\mathcal{S}_{\rightarrow}[|(p_1 \cup p_2)/p|](N) &= \mathcal{S}_{\rightarrow}[|p_1/p|](N) \cup \mathcal{S}_{\rightarrow}[|p_2/p|](N)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}_{\leftarrow}[|\theta :: \lambda|] &= \theta^{-1}(\mathcal{T}(\lambda))\\
\mathcal{S}_{\leftarrow}[|\theta :: \lambda[q]|] &= \theta^{-1}(\mathcal{T}(\lambda) \cap \mathcal{E}[|q|])\\
\mathcal{S}_{\leftarrow}[|\theta :: \lambda/p|] &= \theta^{-1}(\mathcal{S}_{\leftarrow}[|p|] \cap \mathcal{T}(\lambda))\\
\mathcal{S}_{\leftarrow}[|\theta :: \lambda[q]/p|] &= \theta^{-1}(\mathcal{S}_{\leftarrow}[|p|] \cap \mathcal{T}(\lambda) \cap \mathcal{E}[|q|])\\
\mathcal{S}_{\leftarrow}[|/p|] &= \begin{cases} \{n \in T\} & \text{if } \texttt{root} \in \mathcal{S}_{\leftarrow}[|/p|]\\ \emptyset & \text{otherwise} \end{cases}\\
\mathcal{S}_{\leftarrow}[|p|] &= \{x \mid \mathcal{S}_{\rightarrow}[|p|](\{x\}) \neq \emptyset\}\\
\mathcal{S}_{\leftarrow}[|p_1 \cup p_2|] &= \mathcal{S}_{\leftarrow}[|p_1|] \cup \mathcal{S}_{\leftarrow}[|p_2|]\\
\mathcal{S}_{\leftarrow}[|(p_1 \cup p_2)/p|] &= \mathcal{S}_{\leftarrow}[|p_1/p|] \cup \mathcal{S}_{\leftarrow}[|p_2/p|]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[|A|] &= \mathcal{T}(A)\\
\mathcal{E}[|q_1 \texttt{ and } q_2|] &= \mathcal{E}[|q_1|] \cap \mathcal{E}[|q_2|]\\
\mathcal{E}[|q_1 \texttt{ or } q_2|] &= \mathcal{E}[|q_1|] \cup \mathcal{E}[|q_2|]\\
\mathcal{E}[|\texttt{not } q|] &= \{n \in T\} \setminus \mathcal{E}[|q|]\\
\mathcal{E}[|p|] &= \mathcal{S}_{\leftarrow}[|p|]
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{O}[|c|](N) &= \{n \in \mathcal{T}(N)\} \text{ if } \texttt{str} \text{ of } n \text{ is } op\ c\\
\mathcal{E}[|p\ op\ c|] &= \mathcal{O}[|c|](\mathcal{E}[|p|])
\end{aligned}
$$

Fig. 6 The semantics of operators

natural in many applications [7], [17], [20], [37], there are other proposals that consider also bottom-up propagation of security labels [33], [32].

Our security model is based on a specific policy, used for determining a complete authorization specification of a document based on a partial specification. This is the *most-specific-takes-precedence*

(MSTP) policy [21]. Different applications may have different requirements, and we now look at alternative approaches.

We can classify security policies using two orthogonal classifications that focus on *completeness* and *consistency* [21]. The first classification is based on how one handles *unassigned values*, while the second is based on the handling of *conflicting assignments* and how one restores consistency.

We are interested only in policies that are complete and consistent:

Definition 5 A policy is *complete* and *consistent* if every partially annotated tree can be extended to a fully annotated tree.

To capture the variety of policy propagation and conflict resolution options we have identified the following framework:

Local Propagation Policy (LP): "open", "closed", or "none";

Hierarchy Propagation Policy (HP): "topDown" (td), "bottomUp" (bu), or "none";

Structural Conflict Resolution (SC): "localFirst" (lf), "hierarchyFirst" (hf), or "none";

Value Conflict Resolution (VC): "denialTakesPrecedence" (dtp), "permissionTakesPrecedence" (ptp), or "none".

The LP option is similar to traditional policies for access control: in the case of "open", if a node is not labelled N then it is labelled by Y; in the case of "closed", a node not labelled Y is labelled by N; finally, the "none" option says that a node is not labelled.

The HP option specifies node annotation inheritance. In the case of "td", an unlabelled node with a labelled parent inherits the label of its parent. In the case of "bu" an unlabelled node inherits the label from labelled children. The "none" option says that no hierarchy propagation is applied. Note that the "bu" case can result in conflicts, and they should be addressed by the VC Resolution Policy.

The SC option specifies whether the local or the hierarchy rule takes precedence ("lf" or "hf" respectively); while "none" means that the choice depends on the values and on the VC option. The latter specifies how to resolve conflicts for unlabelled nodes that are assigned different labels by the preceding rules: N always has precedence over Y ("dtp"); Y always has precedence over N ("ptp"), and no choice ("none").

Table 1 Policy alternatives

	HP	LP	SC	VC	additional condition
1	td	≠none	hf	∗	none
2	td	none	∗	∗	root is annotated
3	bu	≠none	hf	≠none	none
4	bu	none	∗	≠none	all leaves are annotated
5	∗	≠none	lf	∗	none
6	none	≠none	∗	∗	none
7	≠none	≠none	none	≠none	none
8	none	none	∗	∗	none
9	≠none	≠none	none	none	none
10	bu	∗	hf	none	none
11	bu	none	≠hf	none	none

We list here several possible policies. These are variations of classical security policies that are used in other settings ([21]):

– either *permission-takes-precedence* or *denial-takes-precedence* together with either the *closed* or *open* policy;
– *most-specific-takes-precedence* with *top-down* policy and root node labelled either Y or N by default.
– *most-specific-takes-precedence* with *top-down* policy and either the *closed* or *open* policy.

In the sequel, we show some sufficient conditions for complete and consistent policy combinations. We represent all the possible policy options in Table 1, where symbol "$*$" means "any", i.e. any possible value from a related domain (see column headers of Table 1). Note that Table 1 reflects all 81 possible combination of security options, since symbols $*$ and $\neq$ in columns HP, LP, SC, and VC means, respectively, three and two possible values for the corresponding policy option.

Definition 6 A policy is called *top-down/bottom-up/local policy* if it satisfies the conditions in lines 1-2/3-4/5-6 respectively of Table 1.

Proposition 1 *The top-down, bottom-up and local policies are complete and consistent.*

Algorithm Policy Class

Input: Policy combinations: {HP, LP, SC, VC}

Output: Policy class

```
1: if $(HP \neq none\ \wedge\ LP \neq none \wedge\ SC = hierarchyFirst) \vee (HP \neq none\ \wedge\ LP = none)$ then
2:     if $HP = topDown$ then
3:         return topDown policy;
4:     else
5:         if $VC \neq none$ then
6:             return bottomUp policy;
7:         else
8:             return unresolvable policy;
9: else if $(HP \neq none\ \wedge\ LP \neq none \wedge\ SC = localFirst) \vee (HP = none\ \wedge\ LP \neq none)$ then
10:    return local policy;
11: else if $HP \neq none\ \wedge\ LP \neq none \wedge\ SC = none$ then
12:    if $VC \neq none$ then
13:        return multilabel policy
14:    else
15:        return unresolvable policy
16: else if $HP = none\ \wedge\ LP = none$ then
17:    return unresolvable policy;
```

Fig. 7 Algorithm Policy Class

In some cases both HP and LP policies are defined, but SC policy is "none". Hence, we apply both HP and LP thus obtaining for each node more than one security annotation. The result is defined by means of VC policy which should defined, i.e. the conditions in line 7 of Table 1 should be satisfied.

Definition 7 A policy is called a *multilabel policy* if it satisfies the conditions in line 7 of Table 1.

Proposition 2 *The multilabel policy is complete and consistent.*

Proof The proof follows from the completeness and consistency of the hierarchical and local policies along with *value conflict resolution* option defined.

All the other policies are classified as *unresolvable*. Indeed, policies following the condition 8 are incomplete because neither HP nor LP is applied which results in the fact that unlabelled nodes are not assigned any label; policies in lines 9 and 10 are inconsistent because either the "winning" label in multilabel case is not provided (line 9) or value conflict which arises in the case of "bu" policy propagation is not resolved (line 10); policy in line 11 may be either inconsistent (some XML leaves are not defined; therefore, "bu" propagation is inconsistent) or incomplete (all leaves are defined but have different labels; therefore, value conflict arises and cannot be resolved).

So far, we have defined five classes of policies: *local policy*, *top-down policy*, *bottom-up policy*, *multilabel policy*, and *unresolvable policy* class. These classes were identified based on particular combinations of policy options for value propagation (hierarchy or local) and conflict resolution (structural and value).

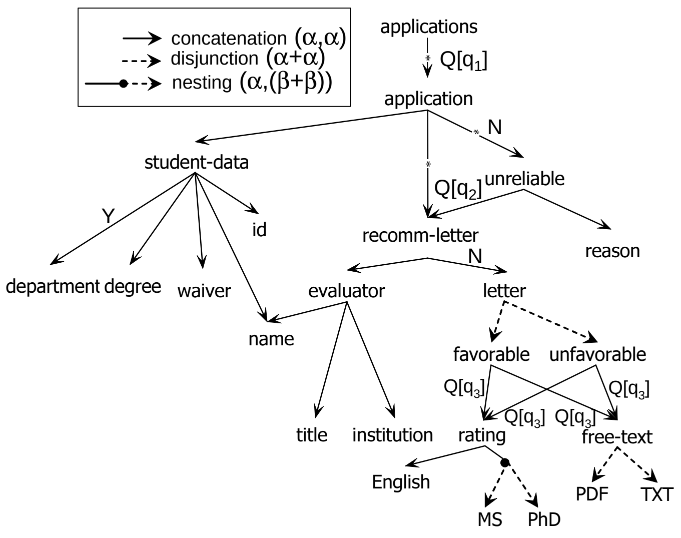
The algorithm of policy class identification is presented in Fig. 7).

In the next sections, we will develop a method that enforces access control on DTD schema emulating XML-based propagation.

5 Schema Access Control for Top-Down Policies

For each user group, an *access specification* is defined to be a partial mapping `ann` such that for each production $A \rightarrow \alpha$ in D and each element type B in α, $\mathtt{ann}(A, B)$ is either Y or N or an XPath qualifier $[q]$, denoting that the B child of an A element is accessible, inaccessible, or conditionally accessible depending on $[q]$, respectively.

Example 6 In Fig. 8, we show an example of security specification described in Example 2: paths to unconditionally allowed (forbidden) element types from their corresponding parents are marked with `Y`(`N`), and conditionally accessible element types are marked by qualifiers q_1, q_2 and q_3 (Fig. 8(b)). In particular, elements `unreliable` and `letter` are forbidden to everybody, while information on department is unconditionally allowed in spite of conditional accessibility (q_1) of its ancestor `application` that is permitted if the students's login is the same as the `id` field of the underlying `student` node. Next, q_2 permits access to `recomm-letter` element if it is a descendant of a permitted `application`, q_3 reveals subtrees rooted at `rating` and `free-text` if the latter are descendants of a permitted `application` having `waiver`="*true*". In all three qualifiers, `$login` is a dynamic variable that is assigned at run time and equals the student's login name.



(a) Security annotation defined at DTD level

$q_1 \doteq$ `student-data/id` $=$ `$login`

$q_2 \doteq ../../$`student-data/id` $=$ `$login`;

$q_3 \doteq$ *ancestor*::`application/student-data`[`id` $=$ `$login`]/`waiver` $=$“*false*”;

(b) Meaning of security annotation qualifiers

Fig. 8 Security annotation for the applicant

Definition 8 An *authorization specification* S is a pair (D, ann), where D is a DTD, ann is a partial mapping such that, for each top-down edge (A, B), $\mathsf{ann}(A, B)$, if defined, is an annotation of the form:

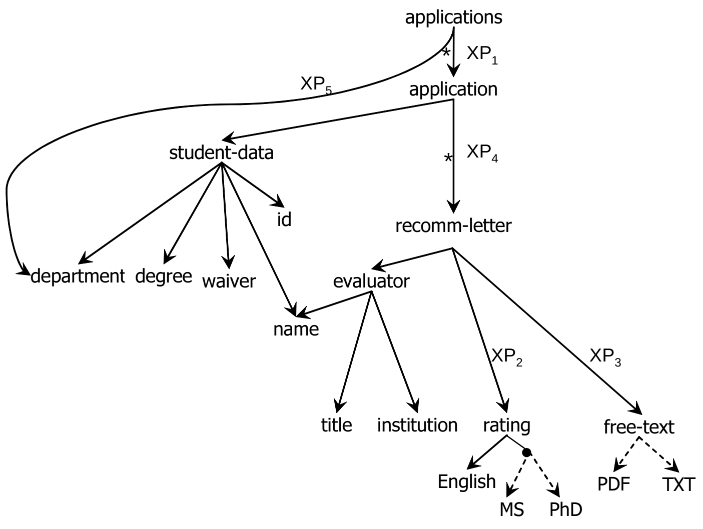
$$\mathsf{ann}(A, B) \quad ::= \quad \mathsf{Q}[q] \;\mid\; \mathsf{Y} \;\mid\; \mathsf{N}$$

where $[q]$ is a qualifier in our fragment $\mathcal{X}$ of XPath. A special case is the root of D, for which we define $\mathsf{ann}(root) = \mathsf{Y}$ by default.

Every $\mathsf{ann}(A, B)$ defines a *source element type* A denoted as s, a *destination element type* B denoted as d, and a *generator of a security label for* B (or simply *generator*) (A, B) denoted g. Thus, we can write $\mathsf{ann}(A, B)$ as $\mathsf{ann}(s, d)$ or $\mathsf{ann}(g)$.

Example 7 In Fig. 8, $\mathsf{ann}(application, unreliable)$=$\mathsf{N}$ defines a source `application`, destination `unreliable` and a generator $(application, unreliable)$ of a security label N for `unreliable`.

Intuitively, labelling an edge (A, B) with an unconditional annotation is a security constraint expressed at the schema level: Y or N indicates that, in the case of top-down propagation, the corresponding B child of an A element in an XML document is always accessible (Y) or always inaccessible (N),



(a) Security DTD view

$XP_1 = \texttt{application}[q_1]$

$XP_2 = \texttt{letter}/(\texttt{favorable} \cup \texttt{unfavorable})/\texttt{rating}[q_3]$

$XP_3 = \texttt{letter}/(\texttt{favorable} \cup \texttt{unfavorable})/\texttt{free-text}[q_3]$

$XP_4 = ./(\epsilon \cup \texttt{unreliable})/\texttt{recomm-letter}[q_2]$

$XP_5 = \texttt{application}[\neg q_1]/\texttt{student-data/department}$

(b) Meaning of XPath expressions

Fig. 9 Security view for the applicant

no matter what the actual values of these elements in the document are. If $\mathsf{ann}(A, B)$ is not explicitly defined, then B *inherits* the accessibility of A or obtains a label from the *default* policy. On the other hand, if $\mathsf{ann}(A, B)$ is explicitly defined it *overrides* the accessibility of B obtained via propagation.

6 Security Views for Top-Down Policies

From a specification as presented in the previous subsection, we would like to infer a *DTD view* D_v which represents a schema of available data for the user. We use a set of XPath queries showing how to construct D_v from the initial DTD D. We call this set of XPath expressions σ*- function.*

At this point, it would be interesting to continue the comparison of our vision of view with those of proposals in [7], [18], and [23]. Namely, the first one does not consider any schema information and reveals the intial DTD to the user as it is. Obviously, this strategy will leak too much sensitive information and hence is not reliable. Damiani's loosened version of the DTD is the original DTD with optional

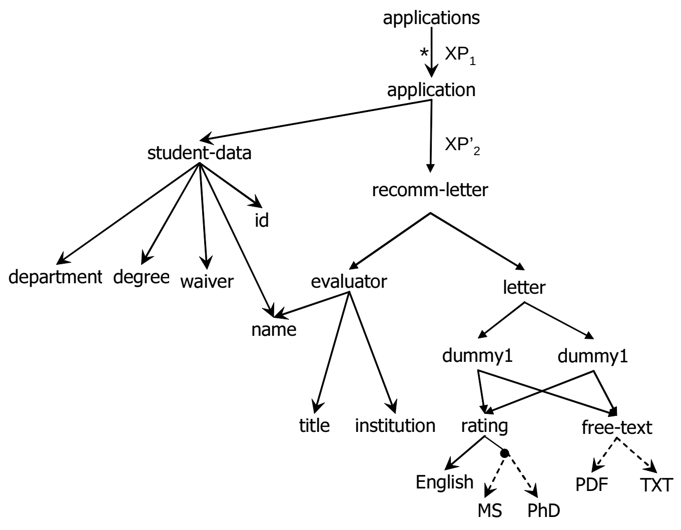


Fig. 10 Authorized DTD view for student Vladimir Romanov

cardinality “?” on every edge that was not of cardinality zero-or-many “$*$”. It slightly alleviates the problem of the previous approach but not enough. Finally, the DTD view constructed according to the algorithm in [23] is shown in Fig. 10. Namely, edge $(applications, department)$ is missing, two dummy elements are introduced instead of `favorable` and `unfavorable`. Finally, `recomm-letter` is added to the view with $XP_2' = (unreliable \cup \epsilon)/recomm - letter$. However, if $\mathsf{ann}(application, recomm - letter)$ were N then `recomm-letter` would have been missing from the DTD view at all although q_4 allowed an access to it. This is because, in DFS traversal of the DTD graph in [23], the negative authorization would have arrived to `recomm-letter` before q_4 from `unreliable`. It means that `recomm-letter` would have been immediately substituted with its children.

In the following, the view is provided to the users so that they can formulate their queries over the view. This means that the users can only access data via D_v. At the same time, the function σ is withheld from the users, and is used by the system to extract accessible data from the actual XML document.

Example 8 Fig. 6 shows the view DTD D_v which represents accessible data according to Example 2, and σ-function expressed by means of XPath expressions XP_1-XP_5. In particular, XP_1 says that only nodes of type `application` with student's `id` equal to student's login are included in the view. XP_2 and XP_3 skip all forbidden elements on the path from `recomm-letter` to, respectively, `rating` and `free-text` that are the children of accessible `application` (condition q_3), XP_4 extracts all `recomm-letter`, including unreliable ones. Finally, XP_5 collects all `departments`, even those that are located in forbidden parts of the tree.

Definition 9 Let D be a DTD. A *security view* for D is a pair (D_v, σ) where D_v is a DTD schema of accessible information and σ is a function from pairs of adjacent element types such that for each element type A in D_v and its child element type B, $\sigma(A, B)$ is an expression in $\mathcal{X}$ defining accessibility of B from A.

Drawing an analogy between relational databases and XML, we elaborate a method of XML view materialization using the σ-function so that the XML view conforms to the DTD view D_v.

Example 9 It is easy to see that the views for Dmitry Konovalov and Vladimir Romanov, presented in Example 2 conform to the DTD view of Example 8.

The derivation of a materialized XML view is explained in the next definition:

Definition 10 Let $\mathcal{S} = (D_v, \sigma)$ be a security view. The semantics of $\mathcal{S}$ is a mapping from documents T conforming to D to documents $T_{\mathcal{S}}$ such that:

1. $T_{\mathcal{S}}$ conforms to D_v
2. The nodes of $T_{\mathcal{S}}$ are a subset of the nodes of T, and their element type is unchanged.
3. For any node n of T which is in $T_{\mathcal{S}}$, let A be the element type of n, and let $B_1, \ldots, B_m$ be the list of element types that occur in $P(A)$. Then the children of n in $T_{\mathcal{S}}$ are

$$\bigcup_{1 \le i \le m} \mathcal{S}_{\rightarrow} [|\sigma(A, B_i)|] (\{n\}) \ .$$

and the order of nodes of type B_i for each $i = 1, \ldots, m$ is the same as in T.

$T_{\mathcal{S}}$ is called the *materialized version* of T w.r.t. the view $\mathcal{S}$.

A classical question for relational database research, namely whether a view produced by the MATERIALIZE algorithm is actually populated by some instances is true. Since the root of the document is always labelled Y, the materialized view always has at least one node. We can show that for XPath fragment, the algorithm is efficient. Let $f(n, d)$ be the complexity of evaluating an XPath expression of size n on a document of size d. Gottlob et al. [27] have shown that for CoreXPath (i.e., $\mathcal{X}$ without union and test) it is $f(|\sigma|, |T|) = O(|\sigma| \times |T|)$. We extend their result to $\mathcal{X}$ without test and with a factor of T to the full $\mathcal{X}$ fragment. Let $|\sigma|$ be the size of the largest XPath expression in the range of σ. Then:

Lemma 1 *Every XPath query $p \in \mathcal{X}_{NoTest}$ over a document T can be evaluated in time $O(|p| \times |T|)$.*

Proof The proof follows the line of Gottlob, Koch and Pichler [27] for the CoreXPath fragment (that is without union of paths): we use the functions $\mathcal{S}_{\rightarrow}$, $\mathcal{S}_{\leftarrow}$, and $\mathcal{E}$ to compute a query tree which is then evaluated bottom-up to yield the desired complexity result.

For the full fragment considered here, the naive implementation of union would lead to an exponential blow up because the processing of p_1 is duplicated in $\mathcal{S}_{\rightarrow}[|p_1/(p_2 \cup p_3)|](N) = \mathcal{S}_{\rightarrow}[|p_1/p_2|](N) \cup \mathcal{S}_{\rightarrow}[|p_1/p_3|](N)$.

To avoid this blow-up we use a query DAG instead of a query tree. Each path of the form $\mathcal{S}_{\rightarrow}[|p_1/(p_2 \cup p_3)|](N)$ is mapped into a (single source) rooted DAG in which the root is labelled $\cup$ with two children, one corresponding to the root of $\mathcal{S}_{\rightarrow}[|p_2|](X)$ and one corresponding to the root of $\mathcal{S}_{\rightarrow}[|p_3|](X)$. The shared X leaf node is the root of the $\mathcal{S}_{\rightarrow}[|p_1|](N)$ node.

Formally, this is equivalent to say that $\mathcal{S}_{\rightarrow}[|p_1/(p_2 \cup p_3)|](N)$ is evaluated using the symbolic rightmost lazy evaluation. In other words,

```
let   $X_1 = \mathcal{S}_{\rightarrow}[|p_1|](N);$
let   $X_{21} = \mathcal{S}_{\rightarrow}[|p_2|](X_1);$
let   $X_{31} = \mathcal{S}_{\rightarrow}[|p_3|](X_1);$
then  $\mathcal{S}_{\rightarrow}[|p_1/(p_2 \cup p_3)|](N) = (X_{21} \cup X_{31}).$
```

For the evaluation of the $\mathcal{S}_{\leftarrow}[|(p_1 \cup p_2)/p|]$ function, a similar strategy can be applied:

```
let  $X_1 = \{x \mid \mathcal{S}_{\rightarrow}[|p_1|](x) \neq \emptyset\};$
let  $X_2 = \{x \mid \mathcal{S}_{\rightarrow}[|p_2|](x) \neq \emptyset\};$
let  $X = \mathcal{S}_{\rightarrow}[|p_1|](X_1) \cup \mathcal{S}_{\rightarrow}[|p_2|](X_2);$
then  $\mathcal{S}_{\leftarrow}[|(p_1 \cup p_2)/p|] = \mathcal{S}_{\rightarrow}[|p|](X).$
```

With this construction each XPath expression can be transformed in time $O(|p|)$ into a query DAG of size $O(|p|)$ in which each operation is a set operation that can be computed in time $O(|T|)$ thus yielding the desired upper bound.

The addition of the test operation increases slightly the complexity because the computation of the $\mathcal{O}\left[|c|\right](N)$ operator requires the comparison of the `str` value c with the `str` value at every node of a set $\mathcal{T}(N)$ which may include all nodes of the tree. This yields a quadratic increase in data complexity. Once the $\mathcal{O}\left[|c|\right](N)$ has been computed at the appropriate leaves of the query DAG, all other operations can be done in time linear in the size of the document. Hence, the following takes place:

Lemma 2 *Every XPath query $p \in \mathcal{X}$ over a document T can be evaluated in time $O(|p| \times |T|^2)$.*

Corollary 1 *Every valid DTD view whose annotations are in $\mathcal{X}$, respectively in $\mathcal{X}_{NoTest}$, can be materialized in $O(|\sigma| \times |T|^3)$, resp. $O(|\sigma| \times |T|^2)$, by Algorithm* Materialize.

Proof The first step of the algorithm takes up only $O(|\sigma| \times |T|^3)$, resp. $O(|\sigma| \times |T|^2)$, by using the construction in Lemma 1, resp. Lemma 2, for the evaluation of XPath queries. For the subsequent processing the number of iteration is bounded by the number of nodes in T and each step can be performed in $O(|\sigma| \times |T|)$ steps.

From Lemmas 1, 2 and Corollary 1, we immediately prove the next theorem:

Theorem 1 *Algorithm* Materialize *computes a materialized view in time $O(f(|\sigma|, |T|) \times |T|)$.*

Definition 11 A *valid* security view is one for which the semantics are always well-defined, i.e., if for every document T, its materialized version conforms to the security view DTD.

Not all views are valid: wrong typing, violated cardinality constraints, and other problems could be all causes of a view to be invalid. For example, in the case of loosened DTD, authorized XML view is not valid since the loosened DTD may contain (optional) element types that in XML document should be deleted and their children should be attached to permitted parents. However, the semantics of an optional element assumes that the absence of the element means the absence of the subtree rooted at that element. Consequently, the authorized XML view does not conform to the loosened DTD. To resolve this problem, Damiani et al. [17] proposed to delete only those forbidden nodes that do not have permitted descendants. This means that, sometimes, the user is allowed to see the information that is not permitted. The views that we construct from an annotated DTD are valid (see Sec. 7) and reveal all and only permitted information.

Security specification and views are related as follows.

Definition 12 Let (D, ann) be an authorization specification, and let $\mathcal{S} = (D_v, \sigma)$ be a security view for D. $\mathcal{S}$ is *data equivalent* to (D, ann) iff for every document T conforming to D, the materialized version $T_{\mathcal{S}}$ is isomorphic to the authorized version T_A.

Two weaker characterizations are based on the notion of *data secrecy* and *data availability* [3].

Definition 13 Let (D, ann) be an authorization specification, and $\mathcal{S} = (D_v, \sigma)$ a security view for D.

1. $\mathcal{S}$ guarantees *data secrecy* iff for every T conforming to D, and for every node n of T, if n occurs in $T_{\mathcal{S}}$ then n must also occur in the authorized tree T_A.
2. $\mathcal{S}$ guarantees *data availability* iff for every T conforming to D, and every node n of T, if n occurs in the authorized tree T_A then n occurs in the materialized version $T_{\mathcal{S}}$.

Intuitively a secrecy-preserving view insures that no forbidden node is leaked, whereas an availability-preserving view is a guarantee that no permitted node is withheld from legitimate users. Obviously, data equivalence implies secrecy and availability, but the converse does not hold, since a data equivalent view also "preserves the structure" of the original document.

Given a security view $\mathcal{S} = (D_v, \sigma)$ and document T conforming to a DTD D, we give an algorithm constructing $T_{\mathcal{S}}$ in Fig. 11.

Proposition 3 *If $\mathcal{S} = (D_v, \sigma)$ is a valid view for D, then the result of Algorithm* Materialize *is a document $T_{\mathcal{S}}$ that is the materialized version of T.*

Proof To proof the proposition, we must show that all three conditions of Def. 10.

In lines 7-12 algorithm evaluates $\sigma(A, B_i)$, $i = 1, \ldots, m$ at any non-visited node n of type A in $T_{\mathcal{S}}\dot{}$The result of this evaluation is a set of nodes of types $B_1, \ldots, B_m$ that become children of node n.

[3] Sometimes these notions are also termed "consistency" and "completeness" in the literature [21] but that terminology can be misleading in our context.

Algorithm MATERIALIZE

Input: a document T conforming to DTD D, a DTD View (D_v, σ)

Output: a materialized view $T_{\mathcal{S}}$ of T or $\perp$ (there is no such view)

```
1: Set the root of $T_{\mathcal{S}}$ to be the root of $T$;
2: for all nodes $n$ of type $A$ in $T$ do
3:   let $A \to P(A)$ the corresponding rule in $D_v$
4:   for all $B$ occurring in $P(A)$ do
5:     precompute $\mathcal{S}_{\to} [|\sigma(A,B)|] (\{n\})$
6: assign to $T_{\mathcal{S}}$ the root of $T$ and mark it as unprocessed
7: while there are unprocessed nodes in $T_{\mathcal{S}}$ do
8:   select an unprocessed node $n$ of type $A$ with rule $A \to P(A)$ in $D_v$
9:   mark the nodes in
       $$\bigcup_{B \text{ occurs in } P(A)} \mathcal{S}_{\to} [|\sigma(A,B)|] (\{n\})$$
     in $T$ as unprocessed children of $n$ in $T_{\mathcal{S}}$
10:  if a child of $n$ already occurs as a processed node in $T_{\mathcal{S}}$ then
11:    return $\perp$ (invalid view)
12:  mark $n$ as processed
```

Fig. 11 Algorithm MATERIALIZE

The order of children of type B_i is the same as in T for every $i = 1, \ldots, m$. This is exactly what the third condition of Def. 10 says.

The second condition holds obviously, because the algorithm constructs $T_{\mathcal{S}}$ from T. Consequently, the nodes of $T_{\mathcal{S}}$ are a subset of T. In addition, the algorithm neither changes the names of element types nor adds new ones.

Finally, since we assume that D_v is valid, and any node n of type A in $T_{\mathcal{S}}$ has children of types $P(A)$, it is clear that $T_{\mathcal{S}}$ conforms to D_v. Thus, the first condition holds.

7 View Construction for Top-Down Policies

We now show how to construct a security view, given a DTD document and an authorization specification. The derivation of D_v has two parts: (i) computing D_F such that Y/N labels are assigned to *every* element type, and (ii) restructuring D_F so that N-labelled elements are deleted and their permitted children are attached to their nearest permitted ancestors. In this manner, the view DTD D_v shows all, and only, the accessible data. The first part may use various propagation techniques (top-down, bottom-up, among siblings, etc.) and conflict resolution rules (denial or permission takes precedence, qualifier over Y takes precedence and vice versa, explicitly defined label takes precedence over propagated one, priorities among access control rules applicable to the element, etc.) or may impose some default value on unlabelled elements.

Since we put annotations on DTD edges (generators), the idea behind our algorithm is to "push" security labels from generators to destination types.

Definition 14 If $\mathsf{ann}(s, d)$, is defined and equals a, we say that *s transmits* (or *propagates*) annotation a to d via g.

Having obtained an annotation, a destination type, in its turn, becomes a source type and may transmit its annotation to its children (new destination types) via corresponding generators that were initially unlabelled (thus preserving *most specific takes precedence* condition).

Example 10 In Fig. 8, there is an annotation $\mathsf{ann}(recomm{-}letter, letter)$=$\mathsf{N}$. It means that `recomm-letter` is a source transmitting N to a destination `letter`. The type `letter` obtains annotation N, becomes a source for `favorable` and `unfavorable`, and transmits N to them.

Since an annotation can be presented as a qualifier, the algorithm, first of all, eliminates qualifiers. For this purpose, it expands each of them into a union of two element types: one is the original element type, which is annotated Y, and the other is a *new* type, which is annotated N. Since the tag of an element uniquely determines the type, it follows that new type names cannot match any nodes in a document that conforms to the original DTD. This is not a serious problem, as all these new type names will be deleted in the final security view.

Definition 15 The semantics of $\mathsf{ann}(A, B) = \mathsf{Q}[q]$ is to split of node type B in the DTD into two nodes having the following meanings: *visible* node instance of type B is a child of a node instance of type A if $B[q]$ holds, and *invisible* otherwise.

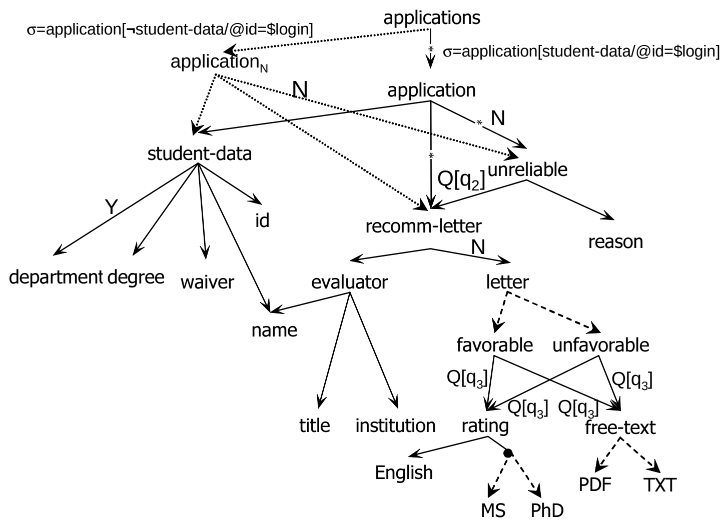


Fig. 12 Removing qualifier $\mathtt{ann}(\mathit{applications}, \mathit{application}) = \mathsf{Q}[\mathtt{student\text{-}data/id} = \mathtt{\$login}]$

Basically, Def. 15 is an emulation of security policy enforcement at XML instance: if $B[q]$ holds at the current concrete node of type B, this node is visible in authorized view; otherwise, it is deleted from the final view. Obviously, the expression $B[q]$ cannot be evaluated at the DTD level.

We must emphasize that the semantics of qualifiers in [23] is different from ours. In particular, qualifiers in [23] have precedence over Y labels. This means that it is impossible to express "global" accessibility of a DTD element (like the accessibility of `department` fields in the current motivating example). In other words, every Y label guarantees accessibility if positive result of evaluating existing qualifiers for ancestors takes place. The advantage of such semantics is a succinctness of policy representation, it is enough to assign Y to (`unreliable`, `recomm-letter`) instead of $\mathsf{Q}[q_2]$ as in motivating example. However, we allow true overriding of qualifiers while authors of [23] don't. Another issue is related to the emulation of security policy in the XML document: the method of [23] simply prunes XML at the node where qualifier does not hold. This is somewhat similar to *denial downward consistency* of [24], [41]. On the other hand, unconditional Y may still override unconditional N in [23]. We believe that our interpretation of qualifiers and the notion of visible/invisible nodes we emulate is a step beyond of this confusion.

Example 11 In Fig. 12, we demonstrate what happens to the edge (`applications`, `application`) with an annotation $\mathsf{Q}[\mathtt{student\text{-}data/id} = \mathtt{\$login}]$. Namely, we split `application` into a visible

$application_Y$, which is a normal `application`, and invisible $application_N$, which should be deleted afterwards. Both newly created elements have the same set of parents and the same set of children. For any child, the newly created element transmits the same annotation as the old one. For each parent, we construct a σ-function describing the situation when qualifier holds for the visible node, and the situation when qualifier does not hold for the invisible. The latter is equivalent to the situation when a negation of the initial qualifier holds.

To avoid an overloading of Fig. 12, we don't show the elimination of qualifiers $\mathsf{Q}[q_2]$ and $\mathsf{Q}[q_3]$. Regarding the last one, elements `rating` and `free-text` should be split twice because of the sources `favorable` and `unfavorable` transmitting qualifier. However, there will be always two kinds of newly created nodes, visible and invisible. Hence, we can either reuse elements created previously, or merge multiple elements of the same visibility into a unique element.

After removing qualifiers, the next step expands the annotation to a "full annotation" by propagating the remaining Y and N labels. The idea is simple: if all incoming edges of some destination element have the same annotation Y or N this element becomes visible or invisible respectively and transmits this annotation to its children. It is easy to see that every XML document has a unique full annotation [23]. At the schema level, however, this is not the case, as there may be several "paths" in the DTD that reach the same element type, each of which results in a different annotation. We use a similar technique when we handle qualifiers, i.e., we introduce new element types, and label the original one with Y (emulation of a visible node), which is connected with parents transmitting to it Y, and the "copy" with N (emulation of an invisible node), which is connected to parents transmitting to it N. The function σ between parents and newly created nodes are simply the name of the split element. This is because invisible copy will be deleted, while the visible one will be considered as the original one. The newly created nodes transmit their visibility to their children.

Example 12 Elements $application_Y$ and $application_N$ transmit, respectively, Y and N to `student-data`. The latter should be split to visible and invisible copies, i.e. $student\_data_{\mathsf{Y}}$ and $student\_data_{\mathsf{N}}$, children of $annotation_Y$ and $annotation_N$ respectively. The σ-function in both cases is equal to $student - data$. The newly created elements are connected to all children of `student-data` and transmit to them cor-

Algorithm Annotate View

Input: A authorization specification (D, ann)

Output: Fully annotated DTD D

```
1: Initialize $D_v := D$ where ann is defined on $D_v$ as on $D$;
2: for all production rules $A \to P(A)$ in $D_v$ and all $B \in P(A)$ do
3:   initialize $\sigma(A, B) := B$
4: for all $A \to P(A)$ and all $B \in P(A)$ with $\mathsf{ann}(A, B) = \mathsf{Q}[q]$ do
5:   add to $D_v$ a new element type $B'$ and a production rule $B' \to P(B)$
6:   replace $B$ by $B + B'$ in $P(A)$
7:   set $\sigma(A, B) := B[q]$; $\sigma(A, B') := B[\neg q]$;
8:   set $\mathsf{ann}_{\mathsf{data}}(B) = \mathsf{Y}$ and $\mathsf{ann}_{\mathsf{data}}(B') = \mathsf{N}$;
9:   for all element types $C$ occurring in $P(B)$ do
10:    set $\sigma(B', C) := \sigma(B, C)$;
11:    set $\mathsf{ann}(B', C) := \mathsf{ann}(B, C)$;
12: while $\mathsf{ann}_{\mathsf{data}}(B)$ of some element types $B$ is undefined do
13:   if all generators $A$ of $B$ have defined $\mathsf{ann}(A, B)$ then
14:     if all $\mathsf{ann}_{\mathsf{data}}(A) = \mathsf{Y}$ then
15:       set $\mathsf{ann}_{\mathsf{data}}(B) := \mathsf{Y}$;
16:     else if all $\mathsf{ann}_{\mathsf{data}}(A) = \mathsf{N}$ then
17:       set $\mathsf{ann}_{\mathsf{data}}(B) := \mathsf{N}$;
18:     else
19:       add to $D_v$ a new element type $B'$ and a production rule $B' \to P(B)$
20:       set $\sigma(A, B') := B$;
21:       set $\mathsf{ann}_{\mathsf{data}}(B) = \mathsf{Y}$, $\mathsf{ann}_{\mathsf{data}}(B') = \mathsf{N}$;
22:       for all element types $C$ occurring in $P(B)$ do
23:         set $\sigma(B', C) := \sigma(B, C)$;
24:         set $\mathsf{ann}(B', C) := \mathsf{ann}(B, C)$;
25:       for all generators $A$ of $B$ do
26:         if $\mathsf{ann}(A) = \mathsf{N}$ then
27:           replace $B$ with $B'$ in $P(A)$
```

Fig. 13 Algorithm Annotate View

responding visibility. However, since $\mathsf{ann}(student - data, department)$=$\mathsf{Y}$, which is more specific, Y overrides negative visibility transmitted by $student - data_N$.

```
Algorithm BUILD VIEW
Input: Fully annotated DTD $D$
Output: A security view ($D_v$, $\sigma$)
1: for all element types $B$ with $\mathsf{ann}(B) = \mathsf{N}$ do
2:     for all production rules $A \to P(A)$ do
3:         if $B$ occurs in $P(A)$ then
4:             for all $C$ that occurs in $P(B)$ do
5:                 set $\sigma(A, C) := \sigma(A, B)/\sigma(B, C) \cup \sigma(A, C)$
6:             replace $B$ by $P(B)$ in $P(A)$ if $B \to P(B)$ exists and by $\epsilon$ otherwise
7: each element $B_{\mathsf{Y}}$ rename to $B$;
```

Fig. 14 Algorithm BUILD VIEW

An overall algorithm for top-down policy propagation is shown in Fig. 13. In particular, lines 4–11 of ANNOTATE VIEW algorithm perform splitting nodes when removing qualifier (Def. 15). Steps 12–27 perform a top-down propagation with splitting (lines 18–27). The result of ANNOTATE VIEW execution is a fully annotated DTD.

Definition 16 DTD is called *fully annotated* if for every DTD node A, there is a defined function $\mathsf{ann}_{\mathsf{data}}(A) ::= \mathsf{Y} \mid \mathsf{N}$. The function $\mathsf{ann}_{\mathsf{data}}$ is called *full annotation* of the DTD document.

Having obtained a fully annotated DTD, we delete all the element types that are labelled N, modifying the regular expressions and the σ functions accordingly. This is shown in the BUILD VIEW algorithm in Fig. 14.

Example 13 In Example 12, `department` is reachable via two paths:

$$p_1 = \texttt{applications}/application_Y[\texttt{student-data/id} = \$login]/student - data_Y/\texttt{department}$$

$$p_2 = \texttt{applications}/application_N[\neg(\texttt{student-data/id} = \$login)]/student - data_N/\texttt{department}$$

In the path p_2, there are two elements that should be deleted: $application_N$ and $student - data_N$. Since all permitted children should be connected to all permitted parents, an additional edge between `applications` and `department` is created and

$$\sigma(applications, department) = \texttt{application}[\neg(\texttt{student-data/id} = \$login)]/\texttt{student-data/department}$$

that is exactly XP_5 from Example 8.

8 Schema-Level Access for Bottom-Up Policies

In the previous section, we showed how to construct the view for the top-down policy. In the case of local policy, we suppose that $\mathsf{ann}(A, B)$ is an annotation between *parent* A and its *child* B. Therefore, pushing of security labels is performed in a top-down manner. This approach assures that there will not be any conflicts at XML tree since every node B will have only one parent A, i.e. only one generator. Hence, we can consider *local* policy to be subsumed by *top-down* policy. Note that we can push annotation bottom-up from children to a parent. However, in this case, the VC option must be defined. Consequently, *local* policy class will be subsumed by *bottom-up* policy. Finally, the multilabel policy requires application of both local and hierarchical (top-down or bottom-up) policies. Therefore, we say that multilabel policy is also subsumed by top-down and bottom-up policies and in this section, we can restrict the attention to bottom-up policy only.

First, the definition of *authorization specification* is extended as follows: all generators of the authorization specification are of one type: either top-down or bottom-up edges.

Definition 17 Element types A and B of DTD D are called *adjacent* if one of the following statements is true: (i) $B \in P(A)$ of a production rule $A \to P(A)$, or (ii) $A \in P(B)$ of a production rule $B \to P(B)$.

In the case (i), a DTD edge (A, B) is a *top-down* edge; otherwise, it is a *bottom-up* edge.

We then can generalize the semantics of qualifiers at the DTD level as follows:

Definition 18 The semantics of $\mathsf{ann}(s, d) = \mathsf{Q}[q]$ is a splitting of node type d into two ones having the following meaning: *visible* node instance of type d is visible if $child(s, d)[q]$ holds, and *invisible* otherwise, where $child(s, d)$ is a function that for generator (s, d) that returns a child element type $ch = s \vee d$ with respect to a DTD structure, i.e., for some element type $U \neq ch, U \in \{s, d\}$, DTD should have a production rule $U \to P(U)$ such that $ch \in P(U)$ [4].

From the top-down view point, the destination d is visible if $d[q]$ (d is a child) holds. On the other hand, from the bottom-up view point, d is visible if $s[q]$ (s is a child) holds. The last interpretation of qualifier seems strange but it is required in such a form because σ-function evaluation and view

[4] In symmetric way we may introduce function $parent(s, d)$ that returns parent element type w.r.t. DTD structure for a pair (s, d)

materialization is held in a top-down manner. Namely, $\sigma(A, B) = Q$ means that Q should be evaluated at B; in the XML view, B children of A are extracted according to Q.

As in the top-down approach we push security annotations along edges, but from children to parents. In this case, we must take into account, that an analogous operation in the XML tree for any destination may have multiple sources.

Example 14 Consider the DTD from Example 1. Generators $(\texttt{student-data}, \texttt{name})$and $(\texttt{evaluator}, \texttt{name})$ differently transmit annotation to `name` in the sense that a node of type *name* may have a parent of type $student{-}data$ or *evaluator*. Hence, these generators cannot influence on $\mathsf{ann}_{\mathsf{data}}(name)$ simultaneously. If we consider a bottom-up case, generators $(\texttt{title}, \texttt{evaluator})$ and $(\texttt{institution}, \texttt{evaluator})$ influence on $\mathsf{ann}_{\mathsf{data}}(evaluator)$ simultaneously because any node of type *evaluator* has both children of types *institution* and *title* in any XML instance. However, this is not a case for generators $(\texttt{MS}, \texttt{rating})$ and $(\texttt{PhD}, \texttt{rating})$ influencing on $\mathsf{ann}_{\mathsf{data}}(rating)$ non-simultaneously as long as a node of type *rating* has either a child of type MS or a child of type PhD.

Suppose, we are given an authorization specification (D, ann). We denote the set of all generators of d as $G(d)$.

Definition 19 We say that a subset of $G(d)$ denoted as $\overline{G}(d)$ has a *simultaneous influence* on $\mathsf{ann}_{\mathsf{data}}(d)$ if there exists T conforming to D such that every instance of type d has a set of either outgoing or incoming edges corresponding to the set $\overline{G}(d)$. We call $\overline{G}(d)$ a *set of simultaneous influence* (SSI).

As we have seen above, a node may have several SSIs. This means that different nodes of the same element type in the same XML document may have different full annotation $\mathsf{ann}_{\mathsf{data}}$. Hence, we split an element w.r.t. a set of sources rather than a single source.

Definition 20 We say that d obtains a *preliminary full annotation* (PFA) from SSI $\overline{G}(d)$, denoted $\mathsf{ann}_{\mathsf{data}}(d)_{\overline{G}(d)}$, if for every $g \in \overline{G}(d)$, $\mathsf{ann}(g)$ it is the same for every $g \in \overline{G}(d)$, $\mathsf{ann}(g) \neq \emptyset$, and $\mathsf{ann}(g) \neq \mathsf{Q}[q]$.

The notion of *PFA* is introduced (i) to reflect the possibility of a node obtaining either a positive or a negative full annotation, and (ii) to take into account the fact that any node may have multiple

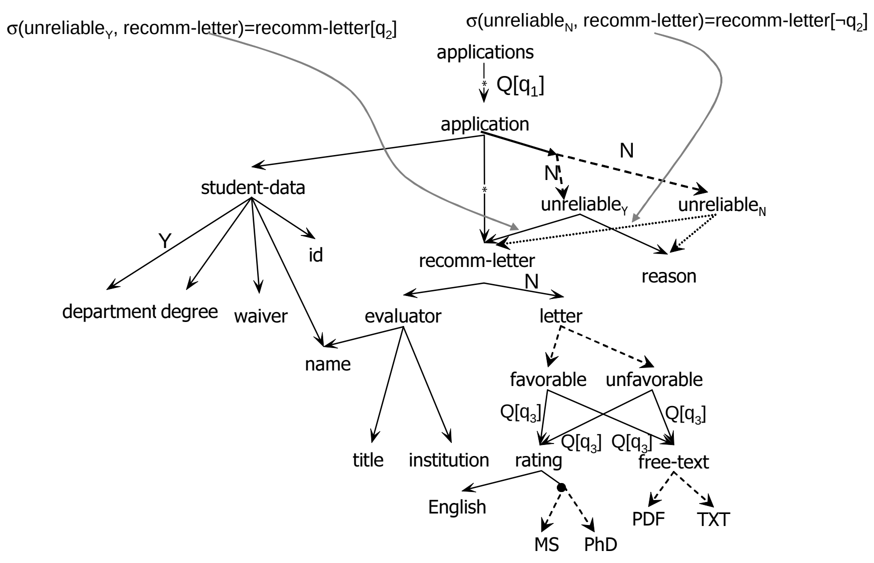


Fig. 15 Removing qualifier $\mathsf{ann}(recomm - letter, unreliable) = \mathsf{Q}[q_2]$

sources. Obviously, if $\mathsf{ann}(g)$ is the same for all $g \in \overline{G}(d)$ then $\mathsf{ann}_{\mathsf{data}}(d)_{\overline{G}(d)} = \mathsf{ann}(g)$. Otherwise, to resolve a conflict within an SSI, we use VC resolution option if it is defined [5]. Value conflict may arise only in the case of bottom-up policy class because every XML instance usually contains a node having more than one child.

In the definition of PFA, we required that $\mathsf{ann}(g) \neq \mathsf{Q}[q]$. This is because the first step of the algorithm should be removing qualifier according to Def. 18.

Let us consider an example of removing qualifiers while propagating security annotations bottom-up. First of all, as in top-down case, d_{Y} and d_{N} transmit, respectively, Y and N to destinations d' of d if $\mathsf{ann}(d, d') = \emptyset$; otherwise, they transmit $\mathsf{ann}(d, d')$ to d'. As Fig. 15 shows, `unreliable` is split into $unreliable_Y$ and $unreliable_N$ both transmitting N to `application` because of the original element `unreliable` transmitting N. Otherwise, $\mathsf{ann}(unreliable_N, application)$ and $\mathsf{ann}(unreliable_Y, application)$ should have been N and Y respectively.

Next, if policy is `bottom-up`, d is substituted with $d_{\mathsf{Y}} + d_{\mathsf{N}}$ in production rule of every d's parent, because d is split conditionally under all its parents. On the other hand, this substitution is done in only one production rule $s \rightarrow P(s)$ if the policy is `top-down`. This is because only s has a child d for which $d[q]$ either holds or does not. Regarding the other sources s' different from a considered s,

[5] If VC option is not defined, the view does not exist

there is a common rule for top-down and bottom-up propagations: d_{N} should be connected with the sources transmitting N while d_{Y} with all the others. After refining generators transmitting qualifiers or nothing, we may need to split d_{Y} again.

Example 15 In Fig. 15, $unreliable_{\mathsf{Y}}$ is connected to both its sources, since $recomm - letter$ transmits Y due to qualifier evaluation, $reason$ transmits nothing. On the other hand, $unreliable_{\mathsf{N}}$ is connected only to $recomm - letter$ that transmits N.

The σ-function between d_{Y} and d_{N} and their source transmitting the qualifier should be $d[q]$ and $d[\neg q]$ respectively. For the other sources s', $\sigma = child(s', d)$. The σ-function for (d, d') is $child(d, d')$.

Example 16 In Fig. 15, we show in boxes a σ between the newly created nodes and their sources that are children. Note that now $\mathsf{ann}(recomm{-}letter, unreliable_Y)$=$\mathsf{Y}$ and $\mathsf{ann}(recomm{-}letter, unreliable_N)$=$\mathsf{N}$ as in top-down case.

The procedure for removing qualifier is depicted in Fig. 16. Cycle FOR starting in line 3 implements the attachment of d_{Y} and d_{N} to parents of d. In lines 8 and 5, the choice $d_{\mathsf{Y}} + d_{\mathsf{N}}$ substitutes d in the case of `top-down` and `bottom-up` policy respectively. Lines 10 and 11 are the beginning of subroutines to connect d_{Y} and d_{N} with source s and other sources $p \neq s$ respectively.

Having removed qualifiers, we can define SSIs. For top-down propagation, SSIs contain only one generator (parent-child DTD edge), and the number of SSIs is equal to the number of parents in DTD graph. The situation is more complicated in bottom-up propagation. First of all, every destination element type d may have several children in the DTD graph transmitting their security labels to d. Secondly, the number of SSIs and their components depend on the presence of choices $(\alpha + \alpha)$ in $P(A)$. More precisely, if we present every sequence (α, α) of $P(A)$ as an arithmetic product $(\alpha \times \alpha)$, and every choice $(\alpha + \alpha)$ of $P(A)$ as an arithmetic sum in parenthesis, then the precise number of SSIs and their configuration is, respectively, the number of components and multipliers in every component in the resulting arithmetic expression after removal of parenthesis.

Example 17 Arithmetic representation of production rule $rating \rightarrow \ (English, (MS + PhD))$ is $rating = (English \times (MS + PhD))$ which is equal to $English \times MS + English \times PhD$ after re-

Algorithm Qualifier Removing

Input: Partially annotated DTD with qualifiers, policy class PC

Output: Partially annotated DTD without qualifiers

```
 1: for every generator $(s, d)$ such that $\mathsf{ann}(s, d) = \mathsf{Q}[q]$ do
 2:   Create element types $d_{\mathsf{Y}}$ and $d_{\mathsf{N}}$;
 3:   for all destinations $d'$ of $d$ do
 4:     Connect $d_{\mathsf{Y}}$ and $d_{\mathsf{N}}$ with $d'$:
          $\sigma(parent(d', d_{\mathsf{Y}}), child(d', d_{\mathsf{Y}})) = \sigma(parent(d', d), child(d', d))$;
          $\sigma(parent(d', d_{\mathsf{N}}), child(d', d_{\mathsf{N}})) = \sigma(parent(d', d), child(d', d))$;
          $\mathsf{ann}(d_{\mathsf{Y}}, d') = \mathsf{ann}(d, d') = \mathsf{ann}(d, d'),\ if\ \mathsf{ann}(d, d') \neq \emptyset$;
          $\mathsf{ann}(d_{\mathsf{Y}}, d') = \mathsf{Y}; \mathsf{ann}(d_{\mathsf{N}}, d') = \mathsf{N},\ if\ \mathsf{ann}(d, d') = \emptyset$;
 5:     if policy is bottom-up then
 6:       for every parent $d'$ of $d$ do
 7:         substitute $d$ with $d_{\mathsf{Y}} + d_{\mathsf{N}}$ in production rule $d' \rightarrow P(d')$;
 8:   if policy is top-down then
 9:     Substitute $d$ for $d_{\mathsf{Y}} + d_{\mathsf{N}}$ in production rule $s \rightarrow\ P(s)$;
10:   Connect $d_{\mathsf{Y}}$ and $d_{\mathsf{N}}$ with $s$:
          $\sigma(parent(s, d_{\mathsf{Y}}), child(s, d_{\mathsf{Y}})) = child(s, d)[q]$;
          $\sigma(parent(s, d_{\mathsf{N}}), child(s, d_{\mathsf{N}})) = child(s, d)[\neg q]$;
          $\mathsf{ann}(s, d_{\mathsf{Y}}) = \mathsf{Y};\ \mathsf{ann}(s, d_{\mathsf{N}}) = \mathsf{N}$;
11:   Connect $d_{\mathsf{Y}}$ ($d_{\mathsf{N}}$ respectively) with other sources $s' \neq s$ transmitting $\mathsf{Y}|\mathsf{Q}[]|nothing$ ($\mathsf{N}$ respectively)
          $\sigma(parent(s', d_{\mathsf{Y}}), child(s', d_{\mathsf{Y}})) = \sigma(parent(s', d), child(s', d))$
          $(\sigma(parent(p, d_{\mathsf{N}}), child(p, d_{\mathsf{N}})) = \sigma(parent(p, d), child(p, d))\ respectively)$;
          $\mathsf{ann}(s', d_{\mathsf{Y}}) = \mathsf{ann}(s', d);\ (\mathsf{ann}(s', d_{\mathsf{N}}) = \mathsf{ann}(s', d) respectively)$;
```

Fig. 16 Algorithm Qualifier Removing

moval of parenthesis. Therefore, in the case of bottom-up propagation, *rating* has two SSIs: $S_1 = \{English, MS\}$ and $S_2 = \{English, PhD\}$.

Next, for every SSI, we calculate a preliminary full annotation using the VC option if necessary (e.g., in the case of `bottom-up` policy class). If different SSIs deliver to d different annotations, we perform the same splitting operation as in the case of qualifier removed.

Algorithm SPLIT

```
Input: DTD element type $d$ having generators with different annotations
1: Create element types $d_{\mathsf{Y}}$ and $d_{\mathsf{N}}$;
2: for every SSI $\overline{G_k}(d)(k = \overline{1,n})$ having sources $\{s_1, \ldots, s_{m_k}\}$ and resulting in a preliminary full annotation
   $\mathsf{Y}$ or $\mathsf{N}$ of $d$ do
3:     Connect source $s_i$ of every generator $g_i \in \overline{G_k}(d), i = \overline{1, m_k}$, respectively, with $d_{\mathsf{Y}}$ or $d_{\mathsf{N}}$ setting:
           $\sigma(parent(s_i, d_{\mathsf{Y}}), child(s_i, d_{\mathsf{Y}})) = child(s_i, d) = \sigma(parent(s_i, d_{\mathsf{N}}), child(s_i, d_{\mathsf{N}}))$
           $\mathsf{ann}(s_i, d_{\mathsf{Y}}) = \mathsf{ann}(s_i, d)(= \mathsf{Y}); \mathsf{ann}(s_i, d_{\mathsf{N}}) = \mathsf{ann}(s_i, d)(= \mathsf{N})$;
4: for every generator $g' = (d, d')$ where $d$ is a source do
5:     Connect $d_{\mathsf{Y}}$ and $d_{\mathsf{N}}$ with $d'$ setting:
           $\sigma(parent(d', d_{\mathsf{Y}}), child(d', d_{\mathsf{Y}})) = child(d', d) = \sigma(parent(d', d_{\mathsf{N}}), child(d', d_{\mathsf{N}}))$
           $\mathsf{ann}(d_{\mathsf{Y}}, d') = \mathsf{ann}(d, d') = \mathsf{ann}(d_{\mathsf{N}}, d')$;
```

Fig. 17 Algorithm SPLIT

Example 18 Suppose $\mathsf{ann}(English, rating)$=$\mathsf{ann}(MS, rating)$=$\mathsf{Y}$, $\mathsf{ann}(PhD, rating)$=$\mathsf{N}$. Then, following the previous example, S_1 results in a positive PFA (i.e., Y), while PFA of S_2 is conflicting (both Y and N are transmitted to `rating`) and depends on VC.

Finally, if all PFAs are the same, then $\mathsf{ann}_{\mathsf{data}}$ of the destination node is clearly defined. Otherwise, as we said above, we perform a splitting operation w.r.t. SSIs transmitting different PFAs.

Example 19 If in the previous example VC=*denialTakesPrecedence*, we have different PFAs: Y from S_1 and N from S_2. Consequently, we split `rating` into a visible and an invisible version. Otherwise, we assign Y as $\mathsf{ann}_{\mathsf{data}}(rating)$.

The generic splitting algorithm (valid for top-down and bottom-up polices) is shown in Fig. 17.

We assume that every DTD element type e that is required to be initially annotated (like `root` or all leaves for bottom-up propagation) automatically retransmits its annotation to all generators $g = (e, d')$ such that $\mathsf{ann}(g) = \emptyset$.

The generic algorithm ANNOTATE VIEW is shown in Fig 18. It starts with a preprocessing procedure which is needed only for the local policy. After preprocessing and qualifier removal steps, we use a *queue*: if the next considered element type d has a full annotation $\mathsf{ann}_{\mathsf{data}}(d)$, there is no need to process it;

Algorithm Annotate View

```
Input: Partially annotated DTD $D$
Output: Fully annotated DTD
 1: Preprocessing;
 2: Qualifier Removing;
 3: Create empty queue, initialize it with all DTD element types;
 4: while queue is not empty do
 5:     $d$ :=Dequeue(queue);
 6:     if $\mathsf{ann}_{\mathsf{data}}(d) = \emptyset$ then
 7:         if $d$ belongs to all generators with defined ann then
 8:             Calculate SSIs $\{\overline{G_1}(d), \overline{G_2}(d), \ldots, \overline{G_n}(d)\}$;
 9:             for every $\overline{G_i}(d)$ do
10:                 Calculate $\mathsf{ann}_{\mathsf{data}}(d)_{\overline{G_i}(d)}$ (applying value conflict resolution policy option if not for all $g \in \overline{G_i}(d)$
                    $\mathsf{ann}(g)$ is the same);
11:             if all PFAs of $d$ are the same (Y or N) then
12:                 Assign any $\mathsf{ann}_{\mathsf{data}}(d)_{\overline{G_i}(d)}$ to $\mathsf{ann}_{\mathsf{data}}(d)$;
13:             else
14:                 Split($d$);
15:             For every $d' \in P_{\mathsf{ann}}(d)$ such that $\mathsf{ann}(d,d') = \emptyset$, set $\mathsf{ann}(d,d') = \mathsf{ann}_{\mathsf{data}}(d)$;
16:         else
17:             Enqueue(queue, $d$);
```

Fig. 18 Algorithm Annotate View

otherwise, the algorithm returns to line 2. If all generators of d are annotated, then we decide $\mathsf{ann}_{\mathsf{data}}(d)$. Otherwise, we place d back to *queue* (step 17).

9 Theoretical Results

Theorem 2 *Let (D, ann) be a security specification where D is non-recursive. Algorithms terminate and produce valid security views.*

Proof First, we prove that the algorithms. Indeed, in Annotate View, there are two sources of iteration: the first is step 2 which terminates because the number of qualifiers is finite, the second one is the cycle **while** starting in line 4 that extends the annotation to a "full" one where $\mathsf{ann}_{\mathsf{data}}$ is defined as either Y or N for every element type. Suppose that the algorithm never reaches the state when *queue* is empty (i.e., does not terminate). It may happen if every element of queue is *expecting* a full annotation [6]. Consider one such an element type e having a source s_{k+1} such that $\mathsf{ann}_{\mathsf{data}}(s_{k+1}) = \emptyset$ which is also *expecting*. Inductively, we suppose that s_{k+1} has an expecting source s_{k+2}, etc. Therefore, in the DTD graph, there exists either an infinite path or a cycle $s_{k+n}, \ldots, s_{k+2}, s_{k+1}, e, s_{k+n}$ of expecting element types that is a contradiction. Therefore, this **while** terminates.

The algorithm Build View always terminates because a fully annotated DTD contains a finite set of N-labelled nodes. Hence, the first step in Build View always reduces the number of element types in the DTD by one.

Secondly, we show that D_v is a DTD. D_v would fail to be a DTD only if, for some element type $A \in D_v$, there exists $B \in P(A)$ such that B was deleted in step 6 of Build View. Since B is deleted by Build View $\mathsf{ann}_{\mathsf{data}}(B)$ in a fully annotated DTD D_F must be equal to N, and therefore B is replaced by $P(B)$ in step 6 of Build View. Hence, B cannot occur in $P(A)$, a contradiction. As we are considering only non-recursive DTDs, we must also show that the new DTD is non-recursive. But this follows immediately, as any cycle D_v can be traced back to a cycle in D.

Finally, we prove that the resulting security view is valid. For this purpose, we must show that $T_{\mathcal{S}}$ conforms to D_v. To do this, we first examine T_F which is the fully annotated version of T, and D_F which is the fully annotated DTD. At this point, we would like to show that T_F conforms to D_F, but there is a problem, namely that some of the nodes in T_F should be typed by element types in D_F because of removing qualifier and splitting. To have them typed appropriately, we extend the notion of typing so that the new types will also match the corresponding old type from which they are generated. Namely, we allow each new element type B_{Y} or B_{N} to type the same nodes that were typed by B. With this modified definition of typing, a node in T_F that is annotated N (resp. Y) will be typed by a type in D_F that is annotated N (resp. Y). Since all the new nodes are deleted at step 6 of Build View the new definition of typing reduces to the standard definition, completing the proof.

Now we need a technical lemma that will be used to proove Theorem 3.

[6] Without the loss of generality, we may consider the state of *queue* when all nodes having $\mathsf{ann}_{\mathsf{data}}$ are deleted at step 5

Lemma 3 *Let (D, ann) be a security specification where D is a not-recursive DTD and (D_v, σ) is the security view that is constructed by Algorithms* Annotate View *and* Build View*, for any sequence of element types $B_0 \ldots B_n$ in the fully annotated D such that (i) B_{i+1} is a child type of B_i for $i = 0 \ldots n-1$, and (ii) each B_i for $i = 1 \ldots n-1$ is annotated N, there exists an XPath expression p and $q_1 \ldots q_n$ XPath qualifiers such that the following equation holds for all set of nodes N:*

$$\mathcal{S}_{\rightarrow}\left[|\sigma(B_0, B_n)|\right](N) = \mathcal{S}_{\rightarrow}\left[|p|\right](N) \cup \mathcal{S}_{\rightarrow}\left[|B_1[q_1]/\cdots/B_n[q_n]|\right](N) \quad .$$

Proof The proof is by a nested induction on n and the number of iterations of step 5 of algorithm Build View.

Base case: $n = 1$, then B_1 is a child of B_0. There are two cases: (1) $\mathsf{ann}(B_0, B_1)$=N, and (2) $\mathsf{ann}(B_0, B_1)$=$\mathsf{Q}[q]$. In the first case, $\sigma(B_0, B_1)$=B_1 because of initialization step of Annotate View algorithm. In the second case, before step 5 of Build View is executed, algorithm Annotate View would set $\sigma(B_0, B_1) = B_1[q_1]$ for a suitable qualifier q_1. Therefore, up to this point, the theorem holds by setting $p = B$, $q_1 = \emptyset$ in the first case and $p = \emptyset$ in the second case. During step 6 of algorithm Build View it is possible that the elimination of some N-children of B_0 would modify $\sigma(B_0, B_1)$. Namely, it may happen if B has a child C which, in its turn has a child B_1. In this case, we get

$$\begin{aligned}
&\mathcal{S}_{\rightarrow}\left[|\sigma(B_0, B_1)|\right](N) = \mathcal{S}_{\rightarrow}\left[|\sigma(B_0, C)/\sigma(C, B_1) \cup \sigma(B_0, B_1)|\right](N) = \\
&\mathcal{S}_{\rightarrow}\left[|\sigma(B_0, C)/\sigma(C, B_1)|\right](N) \cup \mathcal{S}_{\rightarrow}\left[|\sigma(B_0, B_1)|\right](N) \qquad = \\
&\mathcal{S}_{\rightarrow}\left[|\sigma(B_0, C)/\sigma(C, B_1)|\right](N) \cup \mathcal{S}_{\rightarrow}\left[|B_1[q_1]|\right](N) \qquad = \\
&\mathcal{S}_{\rightarrow}\left[|p_1|\right](N) \cup \mathcal{S}_{\rightarrow}\left[|B_1[q_1]|\right](N)
\end{aligned}$$

where q_1 may be $\emptyset$. If B_1 itself is eliminated from $P(B_0)$ this would not change the selection function constructed so far for B_1.

For the inductive case, let $B_0 \ldots B_n$ be the sequence of nodes and let B_i for $i \in \{1 \ldots n-1\}$ be the last node that is eliminated by step 5 of the algorithm Build View. Since the DTD is not recursive neither $\sigma(B_0, B_i)$, nor $\sigma(B_i, B_n)$ can be changed by this step. Without the loss of generality, let $\mathsf{ann}(B_i, B_{i+1})$=$\mathsf{Q}[q]$ for some $0 \leq i < n$. We put $q_{i+1} = \emptyset$ if needed. Then, by evaluating the $\mathcal{S}_{\rightarrow}$

operator and by induction hypothesis we get:

$$
\begin{aligned}
&\mathcal{S}_{\rightarrow}[|\sigma(B_0,B_n)|](N) &&=\\
&\mathcal{S}_{\rightarrow}[|\sigma(B_0,B_i)/\sigma(B_i,B_n)\cup\sigma(B_0,B_n)|](N) &&=\\
&\mathcal{S}_{\rightarrow}[|\sigma(B_0,B_i)/\sigma(B_i,B_n)|](N)\cup\mathcal{S}_{\rightarrow}[|\sigma(B_0,B_n)|](N) &&=\\
&\mathcal{S}_{\rightarrow}[|\sigma(B_i,B_n)|](\mathcal{S}_{\rightarrow}[|\sigma(B_0,B_i)|](N))\cup\mathcal{S}_{\rightarrow}[|p_0|](N) &&=\\
&\mathcal{S}_{\rightarrow}[|\sigma(B_i,B_n)|](\mathcal{S}_{\rightarrow}[|p_{1,i}|](N)\cup\mathcal{S}_{\rightarrow}[|B_1[q_1]/\cdots/B_i[q_i]|](N))\cup\mathcal{S}_{\rightarrow}[|p_0|](N) &&=\\
&\mathcal{S}_{\rightarrow}[|\sigma(B_i,B_n)|](\mathcal{S}_{\rightarrow}[|p_{1,i}|](N))\cup\\
&\qquad\mathcal{S}_{\rightarrow}[|\sigma(B_i,B_n)|](\mathcal{S}_{\rightarrow}[|B_1[q_1]/\cdots/B_i[q_i]|](N))\cup\mathcal{S}_{\rightarrow}[|p_0|](N) &&=\\
&\mathcal{S}_{\rightarrow}[|p_1|](N)\cup\mathcal{S}_{\rightarrow}[|\sigma(B_i,B_n)|](\mathcal{S}_{\rightarrow}[|B_1[q_1]/\cdots/B_i[q_i]|](N))\cup\mathcal{S}_{\rightarrow}[|p_0|](N) &&=\\
&\mathcal{S}_{\rightarrow}[|p_2|](N)\cup\mathcal{S}_{\rightarrow}[|\sigma(B_i,B_n)|](\mathcal{S}_{\rightarrow}[|B_1[q_1]/\cdots/B_i[q_i]|](N)) &&=\\
&\mathcal{S}_{\rightarrow}[|p_2|](N)\cup\mathcal{S}_{\rightarrow}[|p_{i+1,n}|](\mathcal{S}_{\rightarrow}[|B_1[q_1]/\cdots/B_i[q_i]|](N))\cup\\
&\qquad\mathcal{S}_{\rightarrow}[|B_{i+1}[q_{i+1}]/\cdots/B_n[q_n]|](\mathcal{S}_{\rightarrow}[|B_1[q_1]/\cdots/B_i[q_i]|](N)) &&=\\
&\mathcal{S}_{\rightarrow}[|p_2|](N)\cup\mathcal{S}_{\rightarrow}[|p_3|](N)\cup\mathcal{S}_{\rightarrow}[|B_1[q_1]/\cdots/B_i[q_i]/B_{i+1}[q_{i+1}]/\cdots/B_n[q_n]|](N) &&=\\
&\mathcal{S}_{\rightarrow}[|p|](N)\cup\mathcal{S}_{\rightarrow}[|B_1[q_1]/\cdots/B_n[q_n]|](N)
\end{aligned}
$$

The case $i = n$ is similar to the above one by combining the reasoning for the base case and the intermediate case above.

Remark 1 In this lemma, there is no condition on the labelling of either B_0 or B_n as this would make the induction hypothesis needed for the proof not strong enough. Equally we need to quantify over all sets N or the composition of two intermediate sequences during the induction step would not have an inductive hypothesis strong enough.

Theorem 3 *Let (D, ann) be an authorization specification, where D is non-recursive, and (D_v, σ) is the security view constructed by algorithms* Annotate View *and* Build View. *Let T be a document, T_A the authorized version of T and $T_{\mathcal{S}}$ the materialized version of T with respect to (D_v, σ). Then T_A is isomorphic to $T_{\mathcal{S}}$.*

Proof The proof is by top-down induction on T. The root of T is clearly in both T_A and $T_{\mathcal{S}}$. By induction, assume that n is of element type A, and is in both T_A and $T_{\mathcal{S}}$. We must show that each child n in T_A is also a child of n in $T_{\mathcal{S}}$, and vice versa.

$\Longrightarrow$ Let m, of type B, be a child of n in T_A. Assume, first, that m is a child of n in the original document T. Consider the fully annotated DTD $(D_F, \mathsf{ann}_{\mathsf{data}})$. Since n is in T_A, $\mathsf{ann}_{\mathsf{data}}(A) = \mathsf{Y}$. Since m is in T_A, it follows that $\mathsf{ann}_{\mathsf{data}}(B) = \mathsf{Y}$ as well, and so element type B is in D_v; hence it is in $T_{\mathcal{S}}$. Note, that in the case of top-down propagation if $\mathsf{ann}(A, B) = \mathsf{Q}[q]$, then q must hold at m.

We must show that m is in $\mathcal{S}_{\rightarrow} [|\sigma(A, B)|] (\{n\})$. Let p be a path between n and m in T. At the step of initialization, the algorithm Annotate View sets $p = B$. At the step of removing qualifier, $p = B$ may be replaced $p = B[q]$. Finally, step 5 of algorithm Build View may add additional disjuncts to p. In all cases m is clearly in the result.

Now consider the case where m is not a child, but a descendant, of n in T. Let n, n_1, $\ldots$, n_k, m $(k \geq 1)$ be the sequence of nodes in T from n to m, of element types B_1, $\ldots$, B_k. (Next, we suppose that $B_0 = A, B_{k+1} = B$.) Since these nodes are not present in T_A, each $\mathsf{ann}(B_{i-1}, B_i)$, $1 \leq i \leq k$ must be either undefined, N or $\mathsf{Q}[q_i]$, with the qualifier in the latter case evaluated to false at n_i. Furthermore, $\mathsf{ann}_{\mathsf{data}}(B)$ must be either Y or a qualifier $\mathsf{Q}[q]$ that is evaluated to true at m, which implies that B is in D_v.

To show that m is accessible from n in T via path $\sigma(A, B)$, observe first that D_F contains artificial element types that were obtained from B_j by splitting whenever. For this part of the proof, we shall write B_j' and B_j in the case when $\mathsf{ann}_{\mathsf{data}}(B_j) = \mathsf{N}$ and $\mathsf{ann}_{\mathsf{data}}(B_j) = \mathsf{Y}$ respectively. Whenever B_i inherits qualifier $\mathsf{Q}[q_i]$ via its sources, the step of qualifier removing of algorithm Annotate View initially sets $\sigma(B_{i-1}', B_i')$ to $B_i[\neg q]$; when B_i has a generator with ann N or undefined, $\sigma(B_{i-1}', B_i')$ is initially set equal to B_i' in initialization step of Annotate View. Finally, step 6 of Build View deletes elements types B_1', $\ldots$, B_k', replacing $\sigma(A, B)$ by a disjunction of paths, and by Lemma 3 we get:

$$\mathcal{S}_{\rightarrow} [|\sigma(A, B)|] (\{n\}) = \mathcal{S}_{\rightarrow} [|p \cup B_1[\neg q_1]/B_2[\neg q_2]/\cdots/B_k[\neg q_k]/B|] (\{n\})$$

with some of the q_i's absent, when $\mathsf{ann}(B_{i-1}, B_i)$ is N or undefined. It follows that $m \in \mathcal{S}_{\rightarrow} [|\sigma(A, B)|] (\{n\})$ (i.e., m is accessible from n in T via path $\sigma(A, B)$), as desired.

$\Longleftarrow$ For the converse, let m be a child of n in $T_{\mathcal{S}}$. We must show that m is a child of n in T_A.

From the definition of $T_{\mathcal{S}}$, m must be in the result of evaluating $\sigma(A, B)$ at n. Let $n = n_0$, n_1, $\ldots$, n_k, $m = n_{k+1}$ $(k \geq 0)$ be the shortest path from n to m that is used in the evaluation of the σ

function. We claim that n_{i+1} is accessible from n_i in T via the path $\sigma(B_i, B_{i+1})$ ($0 \leq i \leq k$, $B_0 = A$, $B_{k+1} = B$). Indeed, B_i is deleted in step 6 of Build View so that $\sigma(B_{i-1}, B_{i+1})$ is replaced by

$$\sigma(B_{i-1}, B_i)/\sigma(B_i, B_{i+1}) + \sigma(B_{i-1}, B_{i+1}) \ .$$

By our induction hypothesis, $n_{i+1} \in \mathcal{S}_{\rightarrow} [|\sigma(B_{i-1}, B_{i+1})|] (\{n_{i-1}\})$. If n_{i+1} were in the second disjunct above, we would have a contradiction with the assumption that our path was the shortest. Therefore n_{i+1} must be accessible from n_i in T by the path $\sigma(B_i, B_{i+1})$. Therefore for $0 < i < k+1$, $\sigma(B_{i-1}, B_i)$ is

1. B_i when $\mathsf{ann}(B_{i-1}, B_i)$ ($\mathsf{ann}(B_{i+1}, B_i)$ respectively) is either N or undefined. The case $\mathsf{ann}(B_{i-1}, B_i) = \mathsf{Y}$ ($\mathsf{ann}(B_i, B_{i-1}) = \mathsf{Y}$ respectively) is impossible except when $i = k+1$, as the element type B_i is absent in $T_{\mathcal{S}}$ and, consequently, is deleted in step 6 of Build View.
2. $B_i[\neg q]$ when $\mathsf{ann}(B_{i-1}, B_i)$ is $\mathsf{Q}[q]$.

In both case, it follows that n, n_1, $\ldots$, n_k, m is a path in T. It remains to show that n_1, $\ldots$, n_k are deleted in T_A. For nodes inheriting a qualifier via some source, this is immediate; for the other nodes it follows from the fact that the algorithm used to define a complete annotation is the same in the definition of T_A and in Algorithm Annotate View.

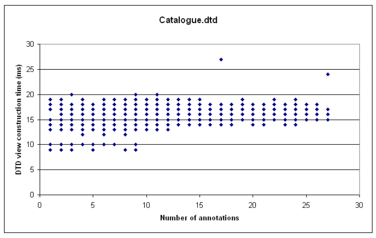
10 Experimental Results

The overall experimentation schema consists of two parts. The first part is to measure scalability an degradation of performance of DTD view construction. The second part shows advantages of schema-based policy enforcement over instance-based one.

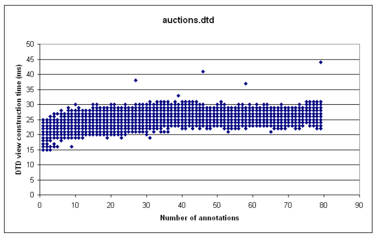
All experiments were held on a working station with 4 CPU Intel Xeon CPUs of 3.20GHz, 4GB of RAM, and RedHat Linux ES 3 as an operating system.

10.1 Scalability

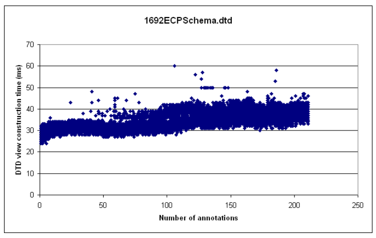
We selected 5 DTDs from http://www.xml.org/xml/schema/ of different size: 24, 79, 211, 802, 2303 edges. For every DTD, we generated 100 times a file containing 1, 2, ..., {24, 79, 211, 802, 2303} Y



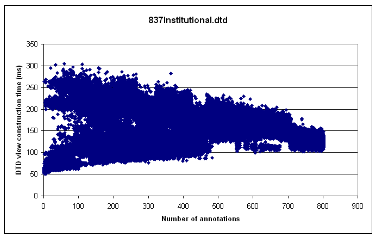
(a)



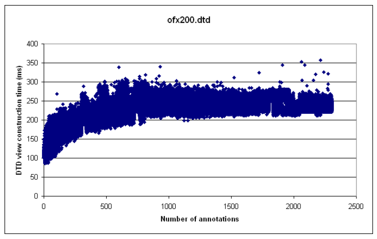
(b)



(c)



(d)



(e)

Fig. 19 DTD view construction degradation for Y/N labels

and N labels assigned randomly. For every generated annotation file, we constructed a corresponding DTD view and measured wall-time required for the construction. Scatter-plot diagrams are shown in Fig. 19. Most of scatter plots demonstrate the same behavior: increasing at the beginning and stabilization (constant time required for view construction) to the end. Increasing is expected: more security labels we have, more splitting and deleting operations may be required. In particular, there may be an effect of “cascading” splitting, when a subgraph of the DTD should be split at every node that goes below the node that was split first. For example, if element type `recomm-letter` in the DTD from our motivating example is split and no other security label goes below that node, the whole subgraph rooted at `recomm-letter` will be split in a cascading manner because both original element

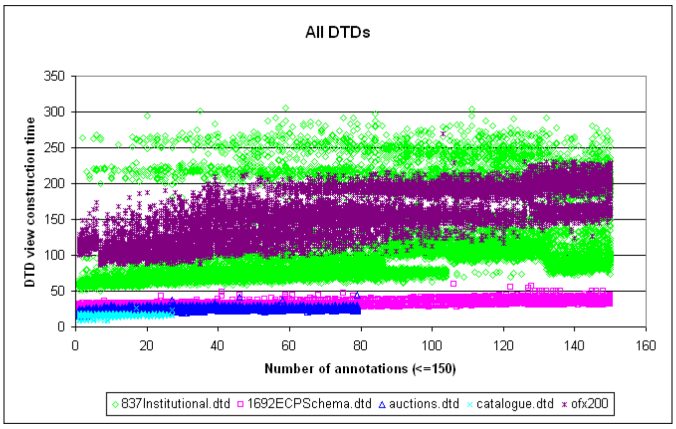
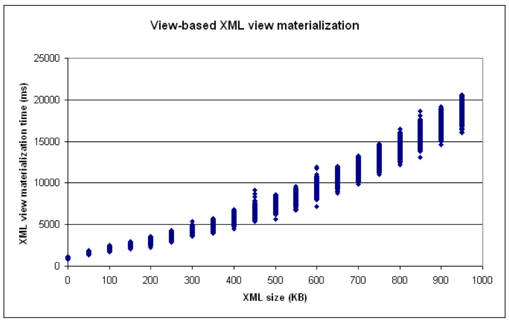


Fig. 20 Scalability of DTD view construction algorithm

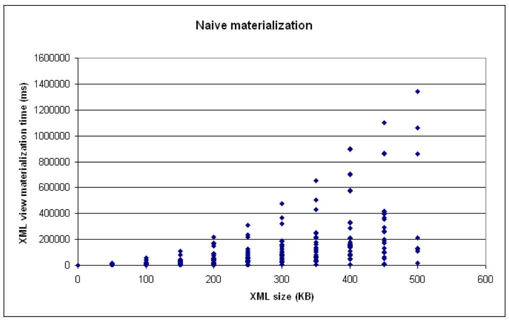
type and its copy are connected to the children of the original element type while transmitting different security annotations (Y and N respectively).

However, at some point, DTD view construction time obtains a constant nature, i.e., it does not increase with the growth of the number of security labels. Moreover, the time variation becomes smaller. It may be explained by the fact that the number of splitting operations reduces in spite of possible conflicts between annotations that require splitting. If we imagine again `recomm-letter` that should be split, but there are other security labels assigned to edges below that element type, it becomes clear that cascading splittings take place not for all underlying element types.

To this point, we have to mention a particular scatter plot on Fig. 19(d). At the beginning there are two branches. The lower branch has the same behavior as in the other scatter plots. The upper branch corresponds to, at least, two times bigger values than the lower one. However, the upper branch tends to descend to the level when there are no difference between two branches. Such a strange scatter plot is explained by a complicated structure of a DTD, i.e., with multiple destinations having many sources. In this case, the lower branch corresponds to cascading splittings in a subgraph that has a smaller number of multi-source element types, while the upper branch represents label propagation in more complicated parts of DTD which require extremely time consuming cascading splits in the presence of a small number of security labels. However, as we have seen above, cascading splits diminish if we



(a) View-based materialization (1-79 annotations)



(b) Naïve materialization (1 annotation)

Fig. 21 Schema-based vs instance-based materialization

have more security labels. As the scatter plot on Fig. 19(d), this also holds for graphs with complicated structure.

Finally, the scalability scatter plot is shown in Fig. 20. As it can be seen, if the number of DTD edges increases 1000 times, DTD view construction time increases only 10-15 times. Thus, we may conclude that our method is scalable. Moreover, the degradation of performance is not significant with respect to the number of either DTD edges or security labels.

10.2 Policy enforcement performance: schema-based vs. instance-based

The next step was to measure a performance of a view-based XML materialization and its comparison with a naïve one (corresponds to Def. 4). For this purpose, we used XMark benchmark [1] to generate 20 different XML documents of size from 1MB to 9.5MB. For all these XML documents, we applied

MATERIALIZE algorithm that was invoked for every DTD view constructed for auctions.dtd above (total: 7900 views with a different number of security labels in the initial annotation). The scatter-plot of such a materialization is shown in Fig. 21(a). Here we have to note that, in order to make experiments repeatable, we used a standard Xalan XPath evaluator by Apache which leads to exponential blow up. Although there have been provided polynomial algorithms of XPath evaluation [28], their implementation is done as a C++ tool [7] rather than a Java library.

We tried to run the same set of experiments for a naïve materialization, when annotations are propagated directly on XML document. However, around 30 hours were taken to run 100 experiments only for the case of a single security label in the initial DTD annotation. The result is shown in Fig. 21(b). It can be noticed a significant difference between two approaches from the view point of "compactness" of materialization time. In other words, the view-based materialization time for the same XML document does not vary much from experiment to experiment. On the other hand, naïve materialization time has an enormous variation. This variation is caused by the presence of qualifiers of the form [$parent$::$parentName$] that are inevitable in the presence of destinations with multiple sources (which is true for auctions.dtd).

There is a big gap between performance of naïve materialization experiments and view-based materialization experiments due to the fact about qualifiers described above. However, in rare cases, naïve materialization is faster than view-based one. It happens in the cases when XML tree has (one) Y label, or (one) N label assigned to a leaf. In other words, these fast results take place because of easy annotations. However, we must note that, in the case of faster naïve rather than view-based materialization, the difference between results is not more than 3-5 seconds which can be considered as negligible from the view point of the overall performance comparison.

10.3 Query answering issues

We started this paper with an argument that DTD view is required for the user in order to alleviate query formulation over accessible data. In this section, we briefly provide different options of query

[7] http://www.dbai.tuwien.ac.at/research/xmltaskforce/

answering over materialized view from the view point of materialization performance and space required for a storage of materialized views.

Option 0: Materialize XML views on the fly by a naïve method. In this option, we store only DTD and security annotations corresponding to different classes of users. Security labels are expressed usually as $\langle xpath, \mathsf{Y}|\mathsf{N}\rangle$ (see [7], [10], [17], [24], [26], [33], [37]) where *xpath* is an XPath expression (possibly with qualifiers) that after evaluation returns all the nodes to which Y or N label should be assigned. It means that simple XML labelling can be extremely time consuming operation because of (i) multiple XPath evaluations, and (ii) the presence of qualifiers in XPaths.

Option 1: Store materialized XML views. This option is very attractive for query evaluation performance: the user simply submits a query which is evaluated over a related XML view. The big disadvantages of this approach are (i) integrity maintenance of all views in the case of updates in the original XML document is error-prone and unfeasible; (ii) XML view is usually 20-40 % smaller than the original XML document. In the presence of a large number of user classes, there will be required a very large disk space to store all those views. The situation can be aggravated in the case of views that contain personal data. It means that in such a situation, we cannot even define classes of views: every user will require a personal view.

Option 2: Store DTD views and materialize XML views on the fly. As we have seen above, there are not so many cases when view-based materialization is slower than a naïve one. Moreover, even such a case takes place, the difference in results is negligible. On the other hand, the overall time for view-based materialization is not more than 20 seconds in all our experiments. Certainly, this result will grow exponentially with an increase of XML size. However, if we apply polynomial algorithms of XPath evaluation, this measure will be decreased significantly. And thus, we can consider this option as acceptable even in the case of interactive web-applications. To this end, we have to remember about space consumption which is reduced drastically because (i) DTD view is much smaller than XML view (while $D_v+\sigma$ is more or less of the same size as the initial DTD document), (ii) the same DTD view can express conditions that allows to extract different personal data for each single user.

Option 3: Construct DTD view and materialize XML view on the fly. At this point, we recall that DTD view construction can be considered as of constant time. Hence as in Option 0, we store a DTD with a set of security annotations which requires small space. On the other hand, DTD view construction plus XML materialization by Materialize algorithm on the fly has an acceptable performance.

Finally, the overall query answering can be improved by introducing a cache that stores frequently asked views. Another approach may exploit query rewriting technique when a user query over the DTD view is rewritten into an equivalent query over the initial XML document [23]. Both these issues are under investigation.

11 Related Work

A typical approach to specifying and enforcing access control for traditional databases is to define views on which security permissions should be applied (e.g., multi-level security view for a relational database [36], [42], [44], [49], discretionary access control over relational databases [6] and object-oriented databases [11]). For instance, Lunt et al. [35] showed how to use standard SQL queries to implement the SeaView multi-level secure database. It is simple to define a relational view via an SQL query, and derive the relational schema of the view. In contrast, the hierarchical structure and the dependency (e.g., ancestor and descendants) of XML data as well as the presence of disjunction and recursion in DTDs make it impossible to define a security view via a single query, and moreover, they introduce new challenges in how to generate a view that conforms to a view DTD. The challenges were observed in [4], which showed that even for XML views of relational data, it is highly nontrivial to ensure that the views typecheck. This observation has been further confirmed in [2] which showed that type checking of XML views of relational data, even for simple DTDs, is computationally intractable: co-NEXPTIME for extremely restricted view definitions, and undecidable for realistic views.

The insight to construct XML security views was found in [42] where it was demonstrated how to compute a full database labelling from a partial one implied by security views. From the XML viewpoint, a partial assignment of security labels to XML document nodes can be also extended to a full assignment. From the latter, it is easy to compute an XML analogue of relational view by means of “sanitization” operation which hides (e.g., deletes or encrypts) nodes with negative authorizations, but reveals (e.g., moves up to a permitted ancestor in the case of deletion of a forbidden parent) their permitted children [7], [17], [24], [33]. In the following sections, we will give an overview of these and other security models of access control to XML documents.

11.1 Runtime policy evaluation

The general scenario of the current category of proposals is the following: The systems defines a set of access control rules of the form $\langle subject, object, action, sign\rangle$ where *subject* is self-explaining, *object* is an XML element/attribute expressed by XPath, *action* is typically `read/write`, $sign \in \{+, -\}$. Different conflict resolution rules and default policy are established as well. With respect to user's request $\langle req\_subject, req\_object, req\_action\rangle$, access control rules applicable to *req_subject* is selected, their *sign*s for *req_action* are propagated to *req_object*. Hence, permission is granted to the user if the resulting sign on *req_object* is +; otherwise, the access is denied.

In particular, [33] introduces the notion of *provisional authorization*, when some provisional action (e.g., logging, transcoding) is performed according to the user's request.

The proposal of [10] considers the case when the access control is moved to clients, e.g, secure tokens and smart cards that are used as trust components in different mobile devices (e.g., PC, PDA, cellular phone) participating in applications dealing with sensitive information (e.g., certification, authentication, electronic voting, e-payment, health care, digital right management, etc.).

Several papers consider the case of evolving access control policies expressed in XQuery [26] and by means of RDF [3], [29]. Such policies can be used for a derivation of new access control rules including content-based constraints of requested and other documents, environmental information like time and place of request initiator, information about possessed privileges. As a result, hardcoding rules for each document and its parts is no more necessary. This feature is especially valuable for systems with dynamic XML documents and requesters' population.

Run-time policy evaluation can be accelerated by efficient lookup of compressed accessibility map where compression is performed on objects [52], on objects and actions [31], on objects, subjects and actions [53].

Another direction for improving runtime policy evaluation concerns static analysis of user queries [40], integration of policy into user query [37], matching user query against efficient policy representation as a tree [41]. In the case when mandatory access control is considered, recursive checks can be reduced by adding special predicates to node tests in the user query [14].

An association between XML nodes can be hidden either at the stage of policy definition [29] or after detecting the possibility of information leakage in security view [50].

An optimization for efficient twig query evaluation in the presence of mandatory access control security annotation is proposed in [14].

Finally, access control for XML documents can be strengthened with a role-based concepts [48].

11.2 Security views for XML

The scenario for this group of proposals is to extract an authorized XML view which includes data relevant to the user's clearance: access control rules applicable to the user are used to partially type the XML tree with Y (+) and N (-) labels. After that, partial annotation is extended to full one. Finally, fully annotated label is sanitized, i.e., N labelled elements are either deleted completely [7], [24] or modified [17].

A different approach to XML security views was shown in [46] where there was an attempt to define a security view by a single XPath expression. In a nutshell, XML elements matching non-asterisk location steps of XPath are added to a view, while asterisks represent forbidden parts of the XML.

11.3 Cryptographically enforced access control

Cryptographically enforced access control is exploited in so-called push architecture, when all users have the same document but each user extracts own view according to a set of possessed keys.

XML encryption pool is presented in [25]. Namely, the method of [19] is used to define permitted and forbidden nodes and the approach of [8] to associate keys with nodes. The nodes with a restricted access are then excluded from the view and are attached to the document in encrypted form as a pool. The pool also stores the information about the original location of a hidden node and a set of associated keys.

The approach in [9] is based on [7]. It avoids generation of multiple physical views for each use by means of different keys for encrypting different portions of the same document. One and only one key is responsible for encryption of each portion of the source XML document. To minimize the number of encryption keys, the portions of the document protected with the same set of policies are encrypted with the same keys. The consequent scenario is key distribution and periodical broadcast of the encrypted document.

Miklau and Suciu extend the Bertino's idea of secure and selective dissemination of XML documents with the notion of *conditional access control rules* [38], which generalizes the term "subject", i.e. authorization is based not on network identifier or user name, but on knowledge presented by the user.

The ideas of [9] and [38] are refined and extended with RBAC in [16].

Nowadays, push architecture is investigated from the view point of secure outsourcing when not only users but also document holder is untrusted [13], [30], [43], [47], [51].

11.4 Schema-based security views

Stoica and Farkas [45] proposed to produce single-level views of XML when conforming DTD is annotated by labels of different confidentiality levels. The key idea lies in analyzing semantic correlation between element types, modification of initial structure of DTD and using cover stories. Altered DTD then undergoes "filtering" when only element types of the confidentiality lever no higher that the requester's one are extracted. However, the proposal requires expert's analysis of semantic meaning of production rules, and this can be unacceptable if database contains a large amount of schemas which are changed occasionally.

Another view-based approach is proposed by Fan et al in [23]. In this paper, we underlined the similarities and the differences with our approach. Here we summarize the proposal of [23]. The process of an access control policies enforcement can be described as follows: (1) define access specification for each class of users, (2) derive a sound and complete DTD security view for a particular access specification, (3) supply the user with a corresponding security DTD view, (4) user issues a query in terms of kept DTD view, (5) a query over the view schema is rewritten to a query over the initial schema and optimized, (6) the optimized query is evaluated over the XML and the result is returned to the requester.

The latest approach to schema-based security views was presented in [39]. The solution allows a complete restructuring of a DTD and relies on a command-like specification language. However, it was mentioned in [39] that many operations are not commutative and have restrictions that means a possibility of errors while designing access control policies.

11.5 Discussion

Two main drawbacks can be easily observed in the first group of run-time policy evaluation scenarios. The first one is that for every user request, accessibility of an XML node is calculated via a propagation of security annotations which may involve the whole tree. The second drawback is that the user is supplied with neither a schema of available data nor an XML view.

The second group of proposals slightly alleviates the problems mentioned above: an XML view is calculated only one and may be queried by the user locally. However, a meaningful DTD schema is still missing. The same is related to crypto-proposals, the difference is that the user may extract the view locally as well if he has a set of corresponding keys.

The group of schema-based view proposals seems to provide the most promising solutions. Namely, the proposal by Stoica and Farkas [45] can construct views that preserve semantic meaning among element types of the original DTD which is missing in more recent proposals [23], [34]. However, this proposal still needs to be evolved in the sense that it is not clear how the constructed view can be used for XML querying. Some intuiting on a possible evolution can be seen in [23] (query rewriting) and [34] (view materialization). Finally, the latest proposal in schema-based views was published in [39]. Their querying language is based on XQuery rather than on XPath. However, we have some doubts about a practical applicability of the method due to its error-prone nature while policy designing and modification.

12 Conclusions

This paper elaborates on certain issues left open in [23]. In particular, we studied access control and security specifications defined over general DTDs in terms of regular expressions rather than normalized DTDs of [23]. Furthermore, we developed a new algorithm for deriving a security view definition from more intuitive access control specification (w.r.t. a non-recursive DTD) without introducing dummy element types, and thus preventing inference of sensitive information from the XML structure revealed by dummies. In addition, we provided an extension of a view derivation algorithm in a bottom-up direction. Other our contributions include the study of different access control policies and experimental evaluation of DTD view construction for Y|N labels and view-based vs. naïve materialization. The former shows that our method is scalable and has an unessential degradation tending to become constant with the growth of the number of annotations. The latter demonstrates advantages of view-based approach over a naïve one from the view point of both performance and space consumption.

Several extensions to the security model are targeted for the future work. First, we plan to extend the definitions of security views and authorization specifications by supporting more complex XML Schema [22] instead of DTDs. Second, we are also studying extensions of our algorithm for deriving security-view definitions with respect to recursive DTDs/schemas. In addition, our next step toward enforcing inference control will be to investigate reasoning techniques in the presence of integrity constraints and ID/IDREF attributes. The observation concerning leakage of information in the case where the security specifications are not closed under intersection is left for the future work as well. Finally, we are studying a query evaluation in the presence of a query rewriting use case.

References

1. XMark – An XML Benchmark Project. http://monetdb.cwi.nl/xml/index.html.
2. Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. Typechecking xml views of relational databases. *ACM Trans. Comput. Logic*, 4(3):315–354, 2003.
3. Chutiporn Anutariya, Somchai Chatvichienchai, Mizuho Iwaihara, Vilas Wuwongse, and Yahiko Kambayashi. A rule-based XML access control model. In *RuleML*, pages 35–48, 2003.
4. M. Benedikt, C. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. DTD-directed publishing with attribute translation grammars. In *Proceedings of the 28th Conference on Very Large Data Bases (VLDB'02)*, 2002.

5. M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. In *Proceedings of the 13th International Conference on Database Theory (ICDT'03)*, 2003.
6. E. Bertino, S. Jajodia, and P. Samarati. A flexible authorization mechanism for relational data management systems. *ACM Transactions on Information Systems (TOIS)*, 17(2):101–140, 1999.
7. Elisa Bertino, M. Braun, Silvana Castano, Elena Ferrari, and Marco Mesiti. Author-X: A Java-based system for XML data protection. In *Proceedings of the IFIP TC11/ WG11.3 Fourteenth Annual Working Conference on Database Security*, pages 15–26. Kluwer, B.V., 2001.
8. Elisa Bertino, Barbara Carminati, Elena Ferrari, Bhavani Thuraisingham, and Amar Gupta. Selective and authentic third-party distribution of XML documents. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(10):1263–1278, oct 2004.
9. Elisa Bertino and Elena Ferrari. Secure and selective dissemination of XML documents. *ACM Transactions on Information and System Security (TISSEC)*, 5(3):290–331, 2002.
10. Luc Bouganim, François Dang Ngoc, and Philippe Pucheral. Client-based access control management for xml documents. In *Proceedings of the 30th Conference on Very Large Data Bases (VLDB'04)*, pages 84–95, 2004.
11. N. Boulahia-Cuppens, F. Cuppens, A. Gabillon, and K. Yazdanian. Multiview model for object-oriented database. In *Proceedings of the Annual Computer Security Applications Conference*, pages 222–231, 1993.
12. T. Bray, J. Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0*. W3C, February 1998.
13. Barbara Carminati, Elena Ferrari, and Elisa Bertino. Securing XML data in third-party distribution systems. In *Proceedings of the fourteenth international conference on Information and knowledge management (CIKM'05)*, pages 99–106, Bremen, Germany, 2005. ACM Press.
14. SungRan Cho, Sihem Amer-Yahia, Laks V.S. Lakshmanan, and Divesh Srivastava. Optimizing the secure evaluation of twig queries. In *Proceedings of the 28th Conference on Very Large Data Bases (VLDB'02)*, pages 490–501, 2002.
15. J. Clark and S. DeRose. XML path language (XPath) version 1.0. w3c recommendation, november 1999. http://www.w3.org/TR/xpath.
16. Jason Crampton. Applying hierarchical and role-based access control to XML documents. In *Proceedings of ACM Workshop on Secure Web Services (SWS'04)*, Fairfax, VA, USA, 2004. ACM Press.
17. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. *ACM Transactions on Information and System Security (TISSEC)*, 5(2):169–202, 2002.
18. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. *ACM Transactions on Information and System Security (TISSEC)*, 5(2):169–202, 2002.

19. Ernesto Damiani, Sabrina de Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Design and implementation of an access control processor for XML documents. In *Proceedings of the 9th International Conference on World Wide Web (WWW'00)*, pages 59–75, Amsterdam, The Netherlands, 2000. North-Holland Publishing Co.

20. Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Fine grained access control for SOAP e-services. In *Proceedings of the 10th International Conference on World Wide Web (WWW'01)*, pages 504–513. ACM Press, 2001.

21. Sabrina De Capitani di Vimercati and Pirangela Samarati. Access control: Policies, models, and mechanism. In R. Focardi and F. Gorrieri, editors, *Foundations of Security Analysis and Design - Tutorial Lectures*, volume 2171 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

22. David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer Second Edition. W3C Recommendation. http://www.w3.org/TR/xmlschema-0/, 2004.

23. Wenfei Fan, Chee-Yong Chan, and Minos Garofalakis. Secure XML querying with security views. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 587–598. ACM Press, 2004.

24. Alban Gabillon and Emmanuel Bruno. Regulating access to XML documents. In *Proceedings of the IFIP TC11/WG11.3 fifteenth annual working conference on Database and application security*, pages 299–314, Niagara, Ontario, Canada, 2001. Kluwer Academic Publishers.

25. Christian Geuer-Pollmann. XML pool encryption. In *Proceedings of the 1st ACM Workshop On XML Security (XMLSEC'02)*, pages 1–9, Fairfax, VA, 2002. ACM Press.

26. Siddhartha K. Goel, Chris Clifton, and Arnon Rosenthal. Derived access control specification for XML. In *Proceedings of the 2nd ACM Workshop On XML Security (XMLSEC'03)*, pages 1–14. ACM Press, 2003.

27. G. Gottlob, C. Koch, and R. Pichler. Efficient algorithm for processing XPath queries. In *Proceedings of the 28th Conference on Very Large Data Bases (VLDB'02)*, 2002.

28. Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

29. Vaibhav Gowadia and Csilla Farkas. RDF metadata for XML access control. In *Proceedings of the 2nd ACM Workshop On XML Security (XMLSEC'03)*, pages 39–48, Fairfax, Virginia, 2003. ACM Press.

30. Ravi Chandra Jammalamadaka and Sharad Mehrotra. Querying encrypted XML documents. In *Proceedings of the 10th International Database Engineering and Applications Symposium (IDEAS'06)*, pages 129–136, Washington, DC, USA, 2006. IEEE Computer Society.

31. Mingfei Jiang and Ada Wai-Chee Fu. Integration and efficient lookup of compressed XML accessibility maps. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(7):939–953, July 2005.

32. Michiharu Kudo and Satoshi Hada. XML access control language: Provisional authorization for XML documents. http://www.trl.ibm.com/projects/xml/xacl/xacl-spec.html, 2000.

33. Michiharu Kudo and Satoshi Hada. XML document security based on provisional authorization. In *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS'00)*, pages 87–96, New York, NY, USA, 2000. ACM Press.

34. Gabriel Kuper, Fabio Massacci, and Nataliya Rassadko. Generalized XML security views. In *Proceedings of the tenth ACM symposium on Access control models and technologies (SACMAT'05)*, pages 77–84. ACM Press, June 2005.

35. T. F. Lunt, R. R. Schell, W. R. Shockley, M. Heckman, and D. Warren. A near-term design for the SeaView multilevel database system. In *Proceedings of IEEE Symposium on Security and Privacy (SSP-88)*, pages 234–244. IEEE Computer Society Press, 1988.

36. Teresa F. Lunt, Dorothy E. Denning, Roger R. Schell, Heckman, Mark, and William R. Shockley. The SeaView security model. *IEEE Transactions on Software Engineering (TOSE)*, 16(6):593–607, 1990.

37. Bo Luo, Dongwon Lee, Wang-Chien Lee, and Peng Liu. QFilter: Fine-grained run-time XML access control via NFA-based query rewriting. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management (CIKM'04)*, pages 543–552, New York, NY, USA, 2004. ACM Press.

38. Gerome Miklau and Dan Suciu. Controlling access to published data using cryptography. In *Proceedings of the 29th Conference on Very Large Data Bases (VLDB'03)*, pages 898–909, September 2003.

39. Sriram Mohan, Arijit Sengupta, Yuqing Wu, and Jonathan Klinginsmith. Access control for XML - a dynamic query rewriting approach. In *Proceedings of the 32th Conference on Very Large Data Bases (VLDB'06)*, pages 1–12, Seoul, Korea, 2006. VLDB Endowment.

40. Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. XML access control using static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS'03)*, pages 73–84. ACM Press, 2003.

41. Naizhen Qi and Michiharu Kudo. XML access control with policy matching tree. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS'05)*, volume 3679 of *Lecture Notes in Computer Science*, pages 3–23. Springer-Verlag, 2005.

42. Xiaolei Qian. View-based access control with high assurance. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy (SSP'96)*, page 85, Washington, DC, USA, 1996. IEEE Computer Society.

43. Michael Schrefl, Katharina Grun, and Jurgen Dorn. Semcrypt - ensuring privacy of electronic documents through semantic-based encrypted query processing. In *Proceedings of the 21st International Conference on Data Engineering Workshops (ICDEW'05)*, page 1191, Washington, DC, USA, 2005. IEEE Computer Society.

44. P. D. Stachour and B. Thuraisingham. Design of LDV: A multilevel secure relational database management system. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2(2):190–209, 1990.

45. Andrei G. Stoica and Csilla Farkas. Secure XML views. In *Proceedings of the 16th International Conference on Data and Applications Security (IFIP'02)*, volume 256 of *IFIP Conference Proceedings*, pages 133–146. Kluwer, July 2002.

46. Roel Vercammen, Jan Hidders, and Jan Paredaens. Query translation for XPath-based security views. 2006.

47. Hui (Wendy) Wang and Laks V.S. Lakshmanan. Efficient secure query evaluation over encrypted XML databases. In *Proceedings of the 32th Conference on Very Large Data Bases (VLDB'06)*, pages 127–138, Seoul, Korea, 2006. VLDB Endowment.

48. Jingzhu Wang and Sylvia L. Osborn. A role-based approach to access control for XML databases. In *Proceedings of the 9th ACM symposium on Access control models and technologies (SACMAT'04)*, pages 70–77. ACM Press, 2004.

49. J. Wilson. Views as the security objects in a multilevel secure relational database management system. pages 70–84. IEEE Computer Society Press, 1988.

50. Xiaochun Yang and Chen Li. Secure XML publishing without information leakage in the presence of data inference. In *Proceedings of the 30th Conference on Very Large Data Bases (VLDB'04)*, pages 96–107, 2004.

51. Yin Yang, Wilfred Ng, Ho Lam Lau, and James Cheng. An efficient approach to support querying secure outsourced XML information. volume 4001/2006 of *Lecture Notes in Computer Science*, pages 157–171. Springer Berlin/Heidelberg, 2006.

52. Ting Yu, Divesh Srivastava, Laks V. S. Lakshmanan, and H. V. Jagadish. A compressed accessibility map for XML. *ACM Transactions on Database Systems (TODS)*, 29(2):363–402, 2004.

53. Huaxin Zhang, Ning Zhang, Kenneth Salem, and Donghui Zhuo. Compact access control labeling for efficient secure XML query evaluation. In *Proceedings of the 21st International Conference on Data Engineering Workshops (ICDEW'05)*, page 1275, 2005.