

# Generalized XML Security Views

Gabriel Kuper

kuper@acm.org

Fabio Massacci

Fabio.Massacci@unitn.it

Nataliya Rassadko

Nataliya.Rassadko@dit.unitn.it

## Abstract

We investigate a generalization of the notion of XML security view introduced by Stoica and Farkas [22] and later refined by Fan et al. [12]. The model consists of access control policies specified over DTDs with XPath expression for data-dependent access control policies. We provide the notion of *security views* for characterizing information accessible to authorized users. This is a transformed (sanitized) DTD schema that can be used by users for query formulation and optimization. Then we show an algorithm to materialize “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.

## 1 Introduction

XML [4] 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 in XML and the selective exposure of information to different classes of users based on their access privileges.

To address this issue we need simple, powerful, fine grained authorization mechanisms that

1. can control access to both content and structure;
2. can be enforced without annotating the entire document;
3. still provide a “sanitized” schema information to users.

While specifications and enforcement of access control are well understood for traditional databases [10, 17, 20, 21], the study of security for XML is less established. Although a number of security models have been proposed for XML [3, 6, 9, 16, 18, 19], these models do not meet criterion 3 above and, to a lesser extent, criterion 2. More specifically, these proposed models enforce security constraints at the document level by fully annotating the entire XML document/database [6, 3, 9]; these require expensive view materialization, and complicate the consistency and integrity maintenance. To overcome this limitations, the notion of XML security views was initially proposed by Stoica and Farkas [22] and later refined by Fan et al. [12].

The most important limitation of the mainstream models is the lack of support for authorized users to query the data: they either do not provide in advance any schema information of the accessible data, or expose the entire original document DTD (or its loosened variant). If no schema is provided, or cannot be derived from the chosen access control model, the solution is hardly practical for large and complex documents. If we want to query an hospital databases just to know who is the nurse on duty in ward 13, there is little sense in sanitizing the entire databases (clinical records and all) which is largely irrelevant for us.

Furthermore, 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 will actually look like.

On the other side, revelation of excessive schema information might lead to security breaches: an unauthorized user can deduct or infer confidential information via multiple queries (essentially if the authorization specifications are not closed under intersection) and analysis of the schema even if just accessible nodes are queried.

## 1.1 Our Contribution.

We generalize the notion of XML security views to arbitrary DAG DTDs and to conditional constraints expressed in a very expressive XPath fragment. For each view, a security specification is a simple extension of the document DTD  $D$  with security annotations and security policies exploited to obtain full annotation from partial one. This specification has the advantage that can be easily implemented with little or no modification to state-of-the-art DTD parsers and offer security officers an intuitive feeling of the actual look of sanitized document.

From the specification, we derive a security view  $V$  consisting of a *view DTD*  $D_v$  and a function  $\sigma$  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  $\sigma$  is withheld from the users, and is used to extract accessible data from the actual XML documents to populate a structure conforming to  $D_v$ .

Query optimization can then be performed by users (using security view) and then by the system (by expanding and optimizing the selection function). Thus, it is no longer necessary to process an entire document and only relevant data is retrieved. Moreover, the users can only access data via  $D_v$ , and no information beyond the view can be inferred from (multiple) queries posed on  $D_v$ .

Thus the users can only access data via  $D_v$ , and no information beyond the view can be inferred from (multiple) queries targeted at  $D_v$ .

More specifically, the main contributions of the paper include:

- A refined version of access policies over XML documents using conditional annotations at DTD level;
- A notion of security view that enforces the security constraints at the schema level and provides a view DTD characterizing them;
- An efficient algorithm for materializing security views, which ensures that views conform to view DTDs;
- An algorithm for deriving a security view from a specification of security annotations;
- An algorithm for deriving a security view from a specification of security policies as XPath expressions.

## 1.2 Plan of the paper

The rest of the paper is organized as follows. First we present preliminary notions on XML and XPath in Sec. 2. In Sec. 3 we provide a motivating example. Next we introduce the notion of security specification (Sec. 4) and the notion of view (Sec. 5). We show how to materialize a view and that using views is equivalent to annotating directly the document (Sec. 6). In Sec. 7 we describe classification of security policies with respect to consistency and completeness properties. Some extensions of our model are outlined in Sec. 8. Sec. 9 shows how to derive security views if policy is defined as an XPath expression. Next we present implementation issues (Sec. 10). Finally we conclude the paper in Sec. 11.

## 2 A Primer On XML and XPath

We first review DTDs (Document Type Definitions [4]) and XPath [7] queries.

**Definition 2.1:** A DTD  $D$  is a triple  $(Ele, P, \text{root})$ , where  $Ele$  is a finite set of *element types*;  $\text{root}$  is a distinguished type in  $Ele$ , and  $P$  is a function defining element types such that for each  $A$  in  $Ele$ ,  $P(A)$  is a regular expression over  $Ele \cup \{\text{str}\}$ , where  $\text{str}$  is a special type denoting PCDATA. We use  $\epsilon$  to denote the empty word, and “+”, “;”, and “\*” to denote disjunction, concatenation, and the Kleene star, respectively. We refer to  $A \rightarrow P(A)$  as the *production* of  $A$ . For all element types  $B$  occurring in  $P(A)$ , we refer to  $B$  as a *subelement type* (or a *child type*) of  $A$  and to  $A$  as a *generator* (or a *parent type*) of  $B$ .  $\square$

We assume that DTD is non-recursive, i.e., that the graph has no cycles. Sec. 8 discusses this limitation.

**Definition 2.2:** An XML tree  $T$  conforms to a DTD  $D$  iff

1. the root of  $T$  is the unique node labelled with  $\text{root}$ ;
2. each node in  $T$  is labelled either with an  $Ele$  type  $A$ , called an  $A$  *element*, or with  $\text{str}$ , 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  $\text{str}$  value and is a leaf of the tree.

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

**Example 2.1:** Consider a DTD describing database of applications for PhD program. The DTD  $D$  is defined to be  $(Ele, P, db)$ , where

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

and the function  $P$  is defined as follows (we omit the definition of elements whose type is  $\text{str}$ ):

```
P(applications)      = (application*)
P(application)      = (student-data, recomm-letter*, unreliable*)
P(student-data)     = (department, degree, waver, name)
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 tree conforming to  $D$  consists of a list of *applications* for PhD/MS program. Each application is initiated by a student described via *student-data* with an attribute *id* uniquely identifying student and representing 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” and means that student does (does not) waive his/her right to inspect the content of recommendation letters. Application is supported by several letters of recommendation (*recomm-letter*), some of them

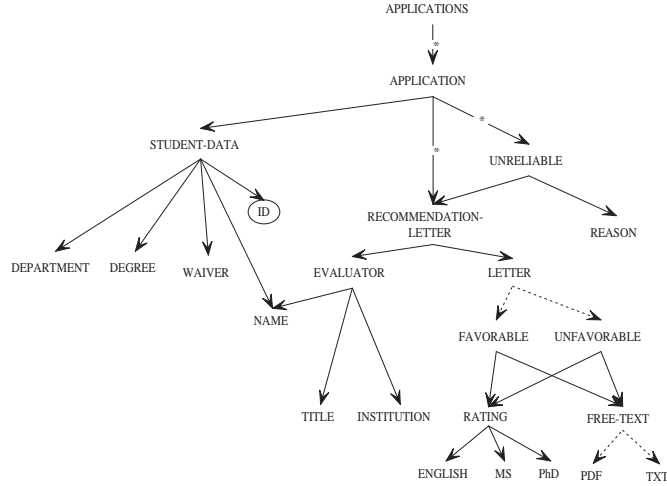


Figure 1: The graph representation of the document DTD  $D$

can be classified as *unreliable* under some *reason*. Each letter has *letter* body and is provided by a separate *evaluator* having *name*, *title* and *institution* attributes. Evaluator places comments on applicant's skills in *free-text* field, which is either *PDF* or *TXT* file, and rates applicant's *English* proficiency, achievements during *MS* program and possible contribution in *PhD* program. Letters of recommendation are reviewed by admission committee and are assigned to a category *favorable* or *unfavorable* depending on the context.

The corresponding DTD is depicted on Fig. 1. □

**Remark 2.1** *Regular expressions in a DTD are 1-unambiguous as required by the XML Standard [4]. In contrast to [12], we consider DTDs defined with general (1-unambiguous) regular expressions.*

We consider a class of XPath queries, which corresponds to the CoreXPath of Gottlob et al. [15] augmented with the union operator and atomic tests and which is denoted by Benedict et al. [12] as  $\mathcal{X}$ .

The XPath axes we consider as primitive are *child*, *parent*, *ancestor-or-self*, *descendant-or-self*, *self*. Gottlob, Koch and Pichler [15] show how the semantics of such axes can be computed in polynomial time. In the sequel we denote by  $\theta$  one of those primitive axes and by  $\theta^{-1}$  its inverse. Notice that each primitive axis has its inverse within the same set of primitives. For instance  $\text{descendant-or-self}^{-1} = \text{ancestor-or-self}$ .

**Definition 2.3:** An XPath expression in  $\mathcal{X}$  is defined by the following grammar:

$$\begin{aligned}
 \langle xpath \rangle &::= \langle path \rangle \mid \langle '/' \rangle \langle path \rangle \\
 \langle path \rangle &::= \langle step \rangle (\langle '/' \rangle \langle step \rangle)^* \\
 \langle step \rangle &::= \theta \mid \theta[\langle qual \rangle] \mid \langle path \rangle \langle '/' \rangle \langle path \rangle \\
 \langle qual \rangle &::= A \mid '*' \mid op\ c \mid \langle xpath \rangle \mid \\
 &\quad \langle qual \rangle \text{ and } \langle qual \rangle \mid \langle qual \rangle \text{ or } \langle qual \rangle \mid \\
 &\quad \text{not } \langle qual \rangle \mid \langle '(' \rangle \langle qual \rangle \langle ')' \rangle
 \end{aligned}$$

where  $\theta$  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* production is called *qualifier* and is denoted by  $q$ . We denote by  $\mathcal{X}_{NoTest}$  the fragment build without the *op c* test. □

For sake of readability, we ignore the difference between *xpath* and *path* we denote both with  $p$ . We also abbreviate

$$\begin{aligned}
\mathcal{S}_{\rightarrow} [[/p]] (N) &= \mathcal{S}_{\rightarrow} [[/p]] (\{\text{root}\}) \\
\mathcal{S}_{\rightarrow} [[\theta[q]]] (N) &= \theta(N) \cap \mathcal{E} [[q]] \\
\mathcal{S}_{\rightarrow} [[\theta[q]/p]] (N) &= \theta(\mathcal{S}_{\rightarrow} [[/p]] (N)) \cap \mathcal{E} [[q]] \\
\\
\mathcal{S}_{\rightarrow} [[p_1 \cup p_2]] (N) &= \mathcal{S}_{\rightarrow} [[/p_1]] (N) \cup \mathcal{S}_{\rightarrow} [[/p_2]] (N) \\
\mathcal{S}_{\rightarrow} [[(p_1 \cup p_2)/p]] (N) &= \mathcal{S}_{\rightarrow} [[/p_1/p]] (N) \cup \mathcal{S}_{\rightarrow} [[/p_2/p]] (N) \\
\mathcal{S}_{\leftarrow} [[/p]] &= \begin{cases} \{n \text{ occurs in } T\} & \text{if } \text{root} \in \mathcal{S}_{\leftarrow} [[/p]] \\ \emptyset & \text{otherwise} \end{cases} \\
\mathcal{S}_{\leftarrow} [[\theta[q]]] N &= \theta^{-1}(N \cap \mathcal{E} [[q]]) \\
\mathcal{S}_{\leftarrow} [[\theta[q]/p]] N &= \theta^{-1}(\mathcal{S}_{\leftarrow} [[/p]] \cap \mathcal{E} [[q]]) \\
\\
\mathcal{S}_{\leftarrow} [[p_1 \cup p_2]] &= \mathcal{S}_{\leftarrow} [[/p_1]] \cup \mathcal{S}_{\leftarrow} [[/p_2]] \\
\mathcal{S}_{\leftarrow} [[(p_1 \cup p_2)/p]] &= \mathcal{S}_{\leftarrow} [[/p_1/p]] \cup \mathcal{S}_{\leftarrow} [[/p_2/p]] \\
\mathcal{E} [A] &= \mathcal{T} (A) \\
\mathcal{E} [q_1 \text{ and } q_2] &= \mathcal{E} [q_1] \cap \mathcal{E} [q_2] \\
\mathcal{E} [q_1 \text{ or } q_2] &= \mathcal{E} [q_1] \cup \mathcal{E} [q_2] \\
\mathcal{E} [\text{not } q] &= \{n \text{ occurs in } T\} \setminus \mathcal{E} [q] \\
\mathcal{E} [p] &= \mathcal{S}_{\leftarrow} [[/p]]
\end{aligned}$$

Figure 2: The semantics of operators

self with  $\epsilon$ , child[A]/p with A/p, descendant-or-self[A]/p with //A/p, q[op c] with q op c and p = p<sub>1</sub>/p<sub>2</sub>, where p<sub>2</sub> is //p'<sub>2</sub>, is written p as p<sub>1</sub>//p'<sub>2</sub>. The ancestor 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. [15] and is identical to proposal of Benedickt et al. [2]. 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 to queried node. The  $\mathcal{E} [[q]]$  function evaluates qualifiers and returns all nodes that satisfy  $q$ .

For sake of readability we overload the  $\theta$ -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 that  $\theta(N) = \{m \mid n \theta m \text{ for } n \in N\}$ . In other words,  $\theta(N)$  returns the nodes that are reachable according 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.

The semantics of the other operators is shown in Fig. 2.

### 3 A Motivating Example

The need to provide users with a schema-level security view is illustrated by the access control requirements in Example 3.1.

**Example 3.1:** The applicant can access only his/her own data located under field `student-data`. Access to fields `favorable` and `unfavorable` is forbidden, while visibility of `rating` and `free-text` is established according to the accessibility to field `letter`. The latter is accessible if the `waiver` is `true` (data-dependent access). Moreover,

the applicant should not be aware of reliability of the recommendation letters as the leakage of this information to recommenders might lead to diplomatic incidents.  $\square$

How can such constraints be enforced? Cho et al. [6] and Bertino et al. [3] enforce these constraints directly on the XML document. Damiani et al. [9] express their security specifications as sets of XPath expressions. However they also transform their XPath specifications into an annotation of the entire document. So we have systems that do specify how to restrict access at the *data level*.

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. [9], 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, since highly confidential information, such as “unreliable” letters, can be deduced anyway.

The information might be inferred via multiple queries by exploiting the document DTD. 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 those of unreliable category), while  $Q_2$  returns only `evaluators` of reliable `recomm-letters`. Although most of the unreliable data is hidden, a look at the document DTD allows one to infer which letters are considered as unreliable: the `evaluators` in  $Q_1$  that are not returned by  $Q_2$ ; thus a security breach.

All evaluators are visible, but by different ways. The trick is to make requestor unable to distinguish those ways.

In traditional relational databases users access a *View* of the data and permissions are assigned to views [17, 20]. A user may be denied the knowledge of the existence of an attribute of a relational schema. What we need here is a view of the document (at the schema level) that the user can use for queries, but that hides not only data but also structural information.

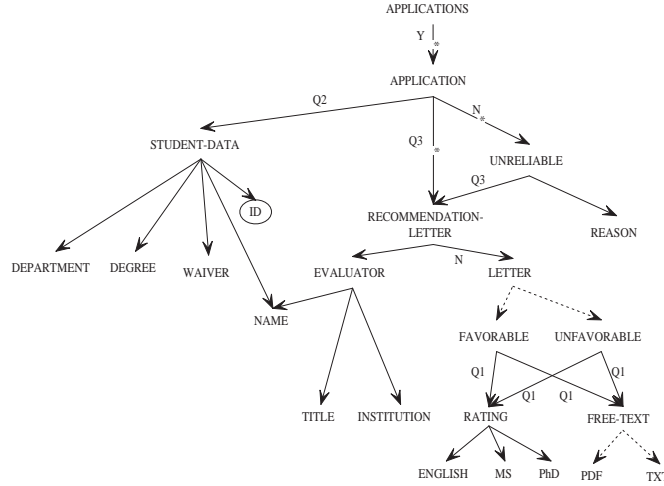
We borrow from Stoica and Farkas [22] the notion of access control model for XML that specifies and enforces security constraints at the *schema* level. For the actual notation we refine and generalize the proposal from Fan et al. [12]: authorizations are defined on a document DTD by annotating element types with Y/N or XPath qualifiers, indicating their accessibility.

From such a specification we can then infer a *view DTD*  $D_v$  and a *selection function*  $\sigma$  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. This means that the users can only access data via  $D_v$ , and no information beyond the view can be inferred from (multiple) queries targeted at  $D_v$ .

The function  $\sigma$  is withheld to the users, and is used to extract accessible data from the actual XML documents with XPath queries to populate a document structure conforming to  $D_v$ .

## 4 Security Specifications

In this section we present our access-control specification language. An *access specification*  $S$  is an extension of a document DTD  $D$  associating security annotations with productions of  $D$ .



(a) Security annotation defined at DTD level

$q_1 \doteq ancestor::application[./student-data[@id = $login]/waiver/text() = "true"];$   
 $q_2 \doteq ./student-data[./@id = $login]$   
 $q_3 \doteq ancestor::application[./student-data[@id = $login];$   
 (b) Meaning of security annotation qualifiers

Figure 3: Security annotation for competing student

**Definition 4.1:** A *authorization specification*  $S$  is a pair  $(D, \text{ann})$ , where  $D$  is a (document) DTD,  $\text{ann}$  is a partial mapping such that, for each production  $A \rightarrow P(A)$  and each child element type  $B$  in  $P(A)$ ,  $\text{ann}(A, B)$ , if explicitly defined, is an annotation of the form:

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

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

Intuitively, annotating production rule  $P(A)$  of the DTD with an unconditional annotation is a security constraint expressed at the schema level:  $Y$  or  $N$  indicates that the corresponding  $B$  children of  $A$  elements in an XML document conforming to the DTD will always be accessible ( $Y$ ) or always inaccessible ( $N$ ), no matter what the actual values of these elements in the document are. If  $\text{ann}(A, B)$  is not explicitly defined, then  $B$  inherits the accessibility of  $A$ . On the other hand, if  $\text{ann}(A, B)$  is explicitly defined it may *override* the accessibility of  $B$  obtained via propagation.

At the data level, the intuition is the following: given an XML document  $T$ , the document is typed with respect to the DTD, and the annotations of the DTD are attached to the corresponding nodes of the document, resulting in a *partially annotated* XML document. Intuitively, given an XML tree  $T$  conforming to  $D$ , the specification  $S$  uniquely defines the accessibility of the elements of  $T$ . Since  $T$  is an instance of  $D$  and the regular expressions in  $D$  are 1-unambiguous, this implies that each  $B$  element  $v$  of  $T$  has a unique parent  $A$  element and a unique production that “parses” the  $A$  subtree. Then we convert the document  $T$  to a *fully annotated* one by labelling all of the unlabelled nodes with  $Y$  or  $N$ . This is done by evaluating the qualifiers and replacing them by  $Y$  or  $N$  annotations, and then by a suitable policy for completing the annotation of the yet labelled nodes of the tree. When everything is labelled we remove all  $N$ -labelled nodes from  $T$ .

We should emphasize that semantics of qualifiers presented in this paper is *different* from that of in [12]. According to [12] a false evaluation of the qualifier is considered as “no label” and requires the inheritance of an access from ancestors, while we assume that once evaluated on the document, a qualifier is mapped to either  $Y$  or  $N$ . This greatly

```

<!ELEMENT applications (application*)>
<!ELEMENT application (student-data, recommendation-letter*, unreliable*)>
<!ELEMENT unreliable (recommendation-letter, reason)>
<!ELEMENT student-data (department, degree, name, waiver)>
<!ELEMENT recommendation-letter (evaluator, letter)>
<!ELEMENT evaluator (title, institution, name)>
<!ELEMENT letter (favorable|unfavorable)>
<!ELEMENT favorable (rating, free-text)>
<!ELEMENT unfavorable (rating, free-text)>
<!ELEMENT rating (MS, PhD, English)>
<!ELEMENT free-text (TXT|PDF)>
<!ELEMENT reason (#PCDATA)>
<!ELEMENT waiver (#PCDATA)>
<!ELEMENT department (#PCDATA)>
<!ELEMENT degree (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT institution (#PCDATA)>
<!ELEMENT MS (#PCDATA)>
<!ELEMENT PhD (#PCDATA)>
<!ELEMENT English (#PCDATA)>
<!ELEMENT TXT (#PCDATA)>
<!ELEMENT PDF (#PCDATA)>

<!ATTLIST student-data id CDATA #IMPLIED>

<!ATTLIST applications
  hierarchy_security_policy CDATA #FIXED "topDown"
  local_security_policy CDATA #FIXED "closed"
  hierarchy_conflict_security_policy CDATA #FIXED "hierarchyFirst"
  value_conflict_security_policy CDATA #FIXED "denialFirst"
  security_annotation_data CDATA #FIXED "Y">

<!ATTLIST application security_annotation_data CDATA #FIXED "Q"
  security_annotation_xpath CDATA #FIXED "./student-data[@id=$login]">

<!ATTLIST rating security_annotation_data CDATA #FIXED "Q"
  security_annotation_xpath CDATA #FIXED
  "ancestor::*[self::application[./student-data[@id=$login]/waiver/text()='true']]">

<!ATTLIST free-text security_annotation_data CDATA #FIXED "Q"
  security_annotation_xpath CDATA #FIXED
  "ancestor::*[self::application[./student-data[@id=$login]/waiver/text()='true']]">

<!-- recommendation-letter tag should be visible under unreliable tag
if ancestor "application" is visible-->
<!ATTLIST recommendation-letter security_annotation_data CDATA #FIXED "Q"
  security_annotation_xpath CDATA #FIXED
  "ancestor::*[self::application[./student-data[@id=$login]]]">

<!ATTLIST letter security_annotation_data CDATA #FIXED "N">
<!ATTLIST unreliable security_annotation_data CDATA #FIXED "N">

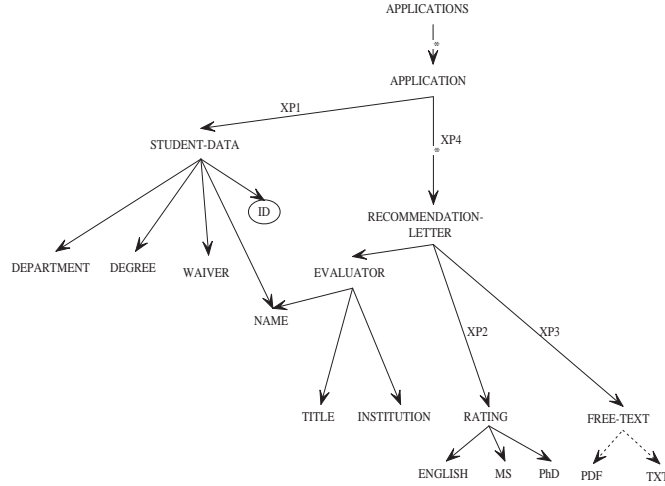
```

Figure 4: Partially annotated DTD

simplifies the intuition of the annotation for a security administrator.

**Example 4.1:** In Fig. 3(a) we show an example of security specification: 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. 3(b)).  $\$login$  is a dynamic variable that is assigned at run time and depends on the student's login name. Fig. 4 shows an input DTD file that can be encoded according to the conditions of Example 3.1. Note that `application` and `recomm-letter` have accessibility condition  $q_1$  and  $q_3$  respectively, while accessibility of elements `rating`, `free-text` is described by qualifier  $q_2$ .  $\square$

**Example 4.2:** The partial annotation generated by the policy in Figure 3(a) is extended to a full annotation by labelling the element types `favorable` and `unfavorable` under `letter` and `reason` under `unreliable` with



(a) Security DTD view

$xp_1 = ./student-data[@id = \$login]$   
 $xp_2 = ./letter/(favorable \cup unfavorable)/rating[q]$   
 $xp_3 = ./letter/(favorable \cup unfavorable)/free-text[q]$   
 $xp_4 = ./(\epsilon \cup unreliable)/recomm-letter$   
 where  $q = applications/application/student-data/waiver = "true"$   
 (b) Meaning of XPath expressions

Figure 5: Security view for competing student

N irrespectively of their position. On the other hand, `evaluator` under `recomm-letter` is labelled as Y. All the other element types labelling depends on the query evaluation.  $\square$

More sophisticated ways of annotation are presented in [14, 23]. In particular, [14] uses XQuery to define derivation access control rules from the existing ones that are organized as XACL privilege triples  $\langle object, subject, access-right \rangle$  [19]. The proposal of [23] is based on the conception of Role Graph Model merged with the conception of RBAC for object-oriented databases.

The construction of a fully annotated document depends heavily on the overall security policy that is chosen to get completeness [10]. The top-down procedure that we describe next is the result of *most-specific-takes-precedence* policy which simply says that an unlabelled node takes the security label of its first labelled ancestor. Damiani et al. [9] use a *closed* policy as default: if a node is not labeled then label it as N. We return to this issue in Sec. 7, where we extend our model to allow alternate propagation techniques.

**Definition 4.2:** Let  $(D, ann)$  be a authorization specification and  $T$  a XML document conforming to  $D$ . The *authorized version*  $T_A$  of  $T$  according the authorization specification is obtained from  $T$  as follows:

1. Type  $T$  with respect to  $D$  and label nodes with  $ann$  values;
2. Evaluate qualifiers top down starting from the root and replace annotations by Y or N depending on the result;
3. For each unlabelled node, label it with the annotation of its nearest labelled ancestor;
4. 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  $T$ .  $\square$

**Example 4.3:** Fig. 5(a) shows the security view generated from the security specification in Fig. 3(a). It hides confidential information. Fig. 5(b) lists some of the XPath annotations that are used to populate the appropriate element types from the original document DTD.  $\square$

Since  $T$  is a tree (a node has only one ancestor) it is not possible to have a conflict on labelling. There are different policy to extend the labelling that may lead to conflicts. We discuss this later in Sec. 7.

The pruning algorithm is more severe than that used by Damiani et al. [9] who delete only subtrees that are entirely labelled N, and delete only the data from nodes labelled N with some descendant labelled Y. As a consequence, the authorized view  $T_A$  no longer conforms to the original DTD  $D$ , not even to its loosened variant.

**Example 4.4:** In example 3.1 since `unreliable` is forbidden, the user should not even know that it exists. So he receives documents without it.  $\square$

## 5 Security Views

We now turn to the enforcement of an access specification. To this end, we introduce the notion of *security view* which consists of two parts. The first part is a schema that is seen by the user, while the second part is a function that is hidden from the user, which describes how the data in the new schema should be derived from the original data. The intuition behind our approach is similar to that of security views for relational databases in multi-level security [17] and the notation is borrowed from [12].

### 5.1 The Definition of Security Views

We first present the syntactic definition of security views.

**Definition 5.1:** Let  $D$  be a DTD. A *security view* for  $D$  is a pair  $(D_v, \sigma)$  where  $D_v$  is a DTD and  $\sigma$  is a function from pairs of element types such that for each element type  $A$  in  $D_v$  and element type  $B$  occurring in  $P(A)$ ,  $\sigma(A, B)$  is an expression in  $\mathcal{X}$ .  $\square$

**Definition 5.2:** 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  $B_1, \dots, B_m$  be the list of element types that occur in  $P(A)$ . Then the children of  $n$  in  $T_{\mathcal{S}}$  are

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

These nodes should be ordered according to the document order in the original document.

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

**Definition 5.3:** 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.  $\square$

Not all views are valid: wrong typing, violated cardinality constraints, and other problems could be all causes of of a view to be invalid. However, the views that we construct from an annotated DTD are valid.

**Example 5.1:** The view with the only production  $\text{root} \rightarrow AA^*$  and  $\sigma(\text{root} \rightarrow A, A) := (A = \text{“alice”})$ , is not defined on the document having the string “alice” as the only  $A$ -child of  $\text{root}$ .  $\square$

**Example 5.2:** The view with the productions  $\text{root} \rightarrow A$  and  $A \rightarrow B$ , where  $\sigma(\text{root} \rightarrow A, A) := A$  and  $\sigma(A \rightarrow B, B) = \text{parent/parent}$  is invalid on any documents because the resulted materialized document cannot be a tree.  $\square$

Security specification and views are related as follows.

**Definition 5.4:** Let  $(D, \text{ann})$  be a authorization specification, and let  $\mathcal{S} = (D_v, \sigma)$  be a security view for  $D$ . We say that  $\mathcal{S}$  is *data equivalent* to  $(D, \text{ann})$  iff for every document  $T$ , conforming to  $D$ , the materialized version  $T_{\mathcal{S}}$  coincides with the authorized version  $T_A$ .  $\square$

Two weaker characterizations are based on the notion of *data secrecy* and *data availability*<sup>1</sup>.

**Definition 5.5:** Let  $(D, \text{ann})$  be a 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 authorization version  $T_A$ .
2.  $\mathcal{S}$  guarantees *data availability* iff for every  $T$  conforming to  $D$ , and every node  $n$  of  $T$ , if  $n$  occurs in authorized tree  $T_A$  then  $n$  occurs in materialized version  $T_{\mathcal{S}}$ .

$\square$

Intuitively, a secrecy-preserving view assure us that no forbidden node is leaked whereas a availability-preserving view is a guarantee that no permitted node is held from legitimate principals. Obviously if a view is data equivalent, then it also guarantees secrecy and availability but the converse does not hold. Indeed a data equivalent view also “preserves the structure” of the original document. We leave such concept of structure preservation informal at this stage, though one may think to subsumption of XML schemas as a possible way to classify views.

## 5.2 View Materialization

Given a security view  $\mathcal{S} = (D_v, \sigma)$  and document  $T$  conforming to a DTD  $D$ , we show how to construct  $T_{\mathcal{S}}$  in Fig. 6.

The following is immediate:

**Proposition 5.1:** *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  $T$ .*  $\square$

A classical question for relational database research, namely whether a view produced by the MATERIALIZE algorithm is actually populated by some instances, has a trivial yes answer. Since the root of the document is always labelled  $Y$ ,

<sup>1</sup>Sometimes these notions are also termed consistency and completeness in the literature [10] but that terminology can be misleading in our context.

**Algorithm:** MATERIALIZE

**Input:** a document  $T$  conforming to DTD  $D$ , a DTD View  $(D_v, \sigma)$

**Output:** a materialized view  $T_S$  of  $T$  or  $\perp$  (there is no such view)

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

Figure 6: Algorithm MATERIALIZE

the materialized view has always one node. We can show that for the XPath fragment we can be as efficient as we can hope for. Indeed, Gottlob, Koch and Pichler [15] have shown that for CoreXPath (i.e.  $\mathcal{X}$  without union and test) it is  $f(|\sigma|, |T|) = |\sigma| \times |T|$ . We extend their result to  $\mathcal{X}$  without test without penalties in complexity and with a  $T$  factor to the full  $\mathcal{X}$  fragment.

We now study the complexity of the algorithm. Let  $f(n, d)$  be the complexity of evaluating an XPath expression of size  $n$  on a document of size  $d$ . Gottlob et al. [15] have shown that for CoreXPath (i.e.  $\mathcal{X}$  without union and test) it is  $f(|\sigma|, |T|) = |\sigma| \times |T|$ . 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  $\sigma$ . Then:

**Theorem 5.2:** *Algorithm MATERIALIZE computes a materialized view in time  $O(f(|\sigma|, |T|) \times |T|)$ .* □

**Lemma 5.3:** *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 [15] 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  $\mathcal{S}_\rightarrow [p_1(p_2 \cup p_3)](N) = \mathcal{S}_\rightarrow [p_1/p_2](N) \cup \mathcal{S}_\rightarrow [p_1/p_3](N)$  the processing of  $p_1$  is duplicated.

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 of  $(\lambda X. \mathcal{S}_\rightarrow [p_2](X) \cup \mathcal{S}_\rightarrow [p_3](X)) \mathcal{S}_\rightarrow [p_1](N)$ .

For the evaluation of the  $\mathcal{S}_\leftarrow$  function we use a single target DAG for the construction of the query DAG.

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.  $\square$

**Lemma 5.4:** *Every XPath query  $p \in \mathcal{X}$  over a document  $T$  can be evaluated in time  $O(|p| \times |T|^2)$ .*  $\square$

The addition of the test operation increases slightly the complexity because the computation of the  $\mathcal{O}(c)$  operator requires the comparison of the `str` value  $c$  with the `str` value at every node of the tree. This yields a quadratic increase in data complexity. Once the  $\mathcal{O}(c)$  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.

**Corollary 5.5:** *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.*  $\square$

**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 5.3, resp. Lemma 5.4, 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.  $\square$

**Remark 5.1** *We cannot obtain a linear bound in the size of  $T$  because of the ancestor and descendant axis in the XPath fragment under consideration. The materialization of each node of  $T_S$  require the evaluation of a query over  $T$  which may involve the entire original document.*

## 6 From Authorization Specifications to Views

Our main result is to show how to construct a security view, given a document DTD and an authorization specification on it. The idea behind our algorithm is to eliminate qualifiers by expanding each qualifier into a union of two element types: one is the original element type, which is annotated Y, and the other is a new type, essentially a copy of the original 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 of these new type names are ultimately deleted in the final security view.

The next step expands the annotation to a “full annotation”. The notion of a full annotation was defined on annotated documents, and we showed that every document has a unique full annotation. 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 to the way we handle qualifiers, i.e., we introduce new element types, and label the original one Y and the “copy” N. Finally, we delete all the element types that are labelled N, modifying the regular expressions and the  $\sigma$  functions correspondingly.

We show the algorithm ANNOTATE VIEW in Fig. 7 and algorithm BUILD VIEW in Fig. 8.

**Definition 6.1:** Let  $S = (D, \text{ann})$  be an authorization specification. The DTD constructed by ANNOTATE VIEW algorithm is the *fully annotated* DTD corresponding to  $(D, \text{ann})$ .  $\square$

**Theorem 6.1:** *Let  $(D, \text{ann})$  be a security specification where  $D$  is non-recursive. Algorithms ANNOTATE VIEW and BUILD VIEW terminate and produce a valid security view.*  $\square$

**Proof:** We have loops in the algorithm ANNOTATE VIEW (steps 6 and 18) and in the algorithm BUILD VIEW (step 1).

Step 6 in ANNOTATE VIEW eliminates qualifiers from the authorization specification.

Step 18 in ANNOTATE VIEW is to extend the annotation to a “full” annotation, i.e., one where `ann` is defined as either Y or N for every element type. We do this by a “top-down” traversal of the DTD, starting from the root. The fact that

**Algorithm:** ANNOTATE VIEW

**Input:** A authorization specification  $(D, \text{ann})$

**Output:** Fully annotated DTD  $D$

```

1: Initialize  $D_v := D$  where  $\text{ann}$  is defined on  $D_v$  as on  $D$ ;
2: for all production rules  $A \rightarrow P(A)$  in  $D_v$  do
3:   for all element types  $B$  occurring in  $P(A)$  do
4:     initialize  $\sigma(A \rightarrow P(A), B) := B[\epsilon]$ 
5:   //Below we will eliminate qualifier annotation
6:   for all element types  $B$  with  $\text{ann}(B) = \mathbf{Q}[q]$  do
7:     add to  $D_v$  a new element type  $B'$  and a production rule  $B' \rightarrow P(B')$ 
8:     set  $P(B') := P(B)$ 
9:     for all element types  $C$  occurring in  $P(B')$  do
10:       $\sigma(B' \rightarrow P(B'), C) := \sigma(B \rightarrow P(B), C)$ 
11:     set  $\text{ann}(B) = \mathbf{Y}$  and  $\text{ann}(B') = \mathbf{N}$ 
12:     for all production rules  $A \rightarrow P(A)$  do
13:       if  $B$  occurs in  $P(A)$  then
14:          $\sigma(A \rightarrow P(A), B) := B[q]$ ;
15:          $\sigma(A \rightarrow P(A), B') := B[\neg q]$ ;
16:         replace  $B$  by  $B + B'$  in  $P(A)$ 
17:   //Below we will get fully annotated DTD  $D$ 
18: while  $\text{ann}(B)$  of some element types  $B$  is undefined do
19:   if all generators  $A$  of  $B$  have defined  $\text{ann}(A)$  then
20:     if all  $\text{ann}(A) = \mathbf{Y}$  then
21:       set  $\text{ann}(B) := \mathbf{Y}$ ;
22:     else if all  $\text{ann}(A) = \mathbf{N}$  then
23:       set  $\text{ann}(B) := \mathbf{N}$ ;
24:     else
25:       add to  $D_v$  a new element type  $B'$  and a production rule  $B' \rightarrow P(B')$ 
26:       set  $P(B') := P(B)$ 
27:       for all element types  $C$  occurring in  $P(B')$  do
28:          $\sigma(B' \rightarrow P(B'), C) := \sigma(B \rightarrow P(B), C)$ 
29:       set  $\text{ann}(B) = \mathbf{Y}$ ,  $\text{ann}(B') = \mathbf{N}$ ,
30:       for all generators  $A$  of  $B$  do
31:         if  $\text{ann}(A) = \mathbf{N}$  then
32:           replace  $B$  with  $B'$  in  $P(A)$ 

```

Figure 7: Algorithm ANNOTATE VIEW

**Algorithm:** BUILD VIEW

**Input:** Fully annotated DTD  $D$

**Output:** A security view  $(D_v, \sigma)$

```

1: for all element types  $B$  with  $\text{ann}(B) = \mathbf{N}$  do
2:   for all production rules  $A \rightarrow P(A)$  do
3:     if  $B$  occurs in  $P(A)$  then
4:       for all  $C$  that occurs in  $P(B)$  do
5:         set  $\sigma(A \rightarrow P(A), C) := \sigma(A \rightarrow P(A), B) / \sigma(B \rightarrow P(B), C) \cup \sigma(A \rightarrow P(A), C)$ 
6:         replace  $B$  by  $P(B)$  in  $P(A)$  if  $B \rightarrow P(B)$  exists and by  $\epsilon$  otherwise
7:  $D_v$  consists of all the element types  $A$  for which  $\text{ann}(A) = \mathbf{Y}$ , with the  $\sigma$  function restricted to these types.

```

Figure 8: Algorithm BUILD VIEW

```

<!ELEMENT applications (application* ,#application*)>
<!ELEMENT application (student-data,recommendation-letter* ,#recommendation-letter* ,unreliable*)>
<!ELEMENT #application (#student-data,recommendation-letter* ,#recommendation-letter* ,unreliable*)>
<!ELEMENT unreliable (recommendation-letter ,#recommendation-letter ,reason)>
<!ELEMENT student-data (department,degree,name,waiver)>
<!ELEMENT #student-data (#department ,#degree ,#name ,#waiver)>
<!ELEMENT recommendation-letter (evaluator,letter)>
<!ELEMENT #recommendation-letter (#evaluator,letter)>
<!ELEMENT evaluator (title,institution,name)>
<!ELEMENT #evaluator (#title ,#institution ,#name)>
<!ELEMENT letter (favorable | unfavorable)>
<!ELEMENT favorable (rating ,#rating ,free-text ,#free-text)>
<!ELEMENT unfavorable (rating ,#rating ,free-text ,#free-text)>
<!ELEMENT rating (MS,PhD,English)><!ELEMENT #rating (#MS ,#PhD ,#English)>
<!ELEMENT free-text (TXT | PDF)><!ELEMENT #free-text (#TXT | #PDF)>
<!ELEMENT reason (#PCDATA)>
<!ELEMENT waiver (#PCDATA)><!ELEMENT #waiver (#PCDATA)>
<!ELEMENT department (#PCDATA)><!ELEMENT #department (#PCDATA)>
<!ELEMENT degree (#PCDATA)><!ELEMENT #degree (#PCDATA)>
<!ELEMENT name (#PCDATA)><!ELEMENT #name (#PCDATA)>
<!ELEMENT title (#PCDATA)><!ELEMENT #title (#PCDATA)>
<!ELEMENT institution (#PCDATA)><!ELEMENT #institution (#PCDATA)>
<!ELEMENT MS (#PCDATA)><!ELEMENT #MS (#PCDATA)>
<!ELEMENT PhD (#PCDATA)><!ELEMENT #PhD (#PCDATA)>
<!ELEMENT PDF (#PCDATA)><!ELEMENT #PDF (#PCDATA)>
<!ELEMENT English (#PCDATA)><!ELEMENT #English (#PCDATA)>
<!ELEMENT TXT (#PCDATA)>
<!ELEMENT #TXT (#PCDATA)>

```

Figure 9: Fully annotated DTD: element part

DTD is non-recursive implies that whenever there remains at least one element type  $B$  with  $\text{ann}(B)$  undefined, there must be one such  $B$  such that whenever  $B$  occurs in  $P(A)$ ,  $\text{ann}(A)$  has already been defined. For one such  $B$ , do the following, and repeat until all element types are annotated. Thus 18 in ANNOTATE VIEW always terminates, whereas step 1 in BUILD VIEW will terminate as it always reduces the number of element types in the DTD by one.

We next show that  $D_v$  is a DTD.  $D_v$  would fail to be a DTD only if, for some element type  $A$  in  $D_v$ ,  $P(A)$  includes an element type  $B$  that is deleted in step 7 of BUILD VIEW. Choose such an  $A$  and  $B$  such that  $B$  has no successor in the DTD tree (we make use again of the non-recursive nature of  $D$ ). Since  $B$  is deleted,  $\text{ann}(B)$  must be equal to  $\mathbb{N}$ , and therefore  $B$  is replaced by  $P(B)$  in step 1 of BUILD VIEW, 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  $D$ .

This shows that we get a security view. To prove it is valid, we must show that  $T_S$  conforms to  $D_v$ . To do this, we first examine  $T'$  the fully annotated version of  $T$  (Definition 4.2) and  $D'$ , the fully annotated DTD defined above. As this point, we would like to show that  $T'$  conforms to  $D'$ , but there is a problem, namely that some of the nodes in  $T'$  should be typed by new element types that were introduced in  $D'$ , which is impossible. To get around this problem, modify the definition of “conforms”, to allow each new element types  $B'$  introduced by the algorithm to type the same nodes that were typed by  $B$ .

With this modified definition of “conforms”, an examination of steps 6 and 18 of the algorithm ANNOTATE VIEW, comparing them to the corresponding steps in the definition of  $T'$ , shows that  $T'$  conforms to  $D'$ . Furthermore, a node in  $T'$  that is annotated  $\mathbb{N}$  (resp.  $\mathbb{Y}$ ) will be typed by a type in  $D'$  that is annotated  $\mathbb{N}$  (resp.  $\mathbb{Y}$ ).

It follows immediately from the definitions, that if we take  $D'$  with the  $\sigma$  function defined in algorithm ANNOTATE VIEW,  $T'$  “conforms” to  $D'$ . As we delete nodes in step 5, we can show that this property is preserved, so that  $T_S$  “conforms” to  $D_v$ . Since all the new nodes have been deleted at this point, the new definition of “conforms” reduces to the standard definition, completing the proof.  $\square$

Note that our assumption that regular expressions in DTDs may be 1-ambiguous is essential, as the following example

```

<!ATTLIST applications
  hierarchy_security_policy CDATA #FIXED "topDown"
  value_conflict_security_policy CDATA #FIXED "denialFirst"
  local_security_policy CDATA #FIXED "closed"
  hierarchy_conflict_security_policy CDATA #FIXED "hierarchyFirst"
  security_annotation_data CDATA #FIXED "Y">
<!ATTLIST application security_annotation_data CDATA #FIXED "Y">
<!ATTLIST #application security_annotation_data CDATA #FIXED "N">
<!ATTLIST unreliable security_annotation_data CDATA #FIXED "N">
<!ATTLIST student-data security_annotation_data #FIXED "N" id CDATA #IMPLIED>
<!ATTLIST #student-data security_annotation_data #FIXED "N" id CDATA #IMPLIED>
<!ATTLIST recommendation-letter security_annotation_data CDATA #FIXED "Y">
<!ATTLIST #recommendation-letter security_annotation_data CDATA #FIXED "N">
<!ATTLIST letter security_annotation_data CDATA #FIXED "N">
<!ATTLIST favorable security_annotation_data #FIXED "N">
<!ATTLIST evaluator security_annotation_data #FIXED "N">
<!ATTLIST #evaluator security_annotation_data #FIXED "N">
<!ATTLIST unfavorable security_annotation_data #FIXED "N">
<!ATTLIST rating security_annotation_data CDATA #FIXED "Y">
<!ATTLIST #rating security_annotation_data CDATA #FIXED "N">
<!ATTLIST free-text security_annotation_data CDATA #FIXED "Y">
<!ATTLIST #free-text security_annotation_data CDATA #FIXED "N">
<!ATTLIST reason security_annotation_data #FIXED "N">
<!ATTLIST waiver security_annotation_data #FIXED "Y">
<!ATTLIST #waiver security_annotation_data #FIXED "N">
<!ATTLIST department security_annotation_data #FIXED "Y">
<!ATTLIST #department security_annotation_data #FIXED "N">
<!ATTLIST degree security_annotation_data #FIXED "Y">
<!ATTLIST #degree security_annotation_data #FIXED "N">
<!ATTLIST name security_annotation_data #FIXED "Y">
<!ATTLIST #name security_annotation_data #FIXED "N">
<!ATTLIST title security_annotation_data #FIXED "Y">
<!ATTLIST #title security_annotation_data #FIXED "N">
<!ATTLIST institution security_annotation_data #FIXED "Y">
<!ATTLIST #institution security_annotation_data #FIXED "N">
<!ATTLIST MS security_annotation_data #FIXED "Y">
<!ATTLIST #MS security_annotation_data #FIXED "N">
<!ATTLIST PhD security_annotation_data #FIXED "Y">
<!ATTLIST #PhD security_annotation_data #FIXED "N">
<!ATTLIST English security_annotation_data #FIXED "Y">
<!ATTLIST #English security_annotation_data #FIXED "N">
<!ATTLIST TXT security_annotation_data #FIXED "Y">
<!ATTLIST #TXT security_annotation_data #FIXED "N">
<!ATTLIST PDF security_annotation_data #FIXED "Y">
<!ATTLIST #PDF security_annotation_data #FIXED "N">

```

Figure 10: Fully annotated DTD: attribute part

shows.

**Example 6.1:** Consider the DTD with element types  $A, B, C, D$ , where  $D \rightarrow CA(A + B)$  and  $D \rightarrow (A + B)^*$ . If  $\text{ann}(A) = \text{ann}(B) = \text{ann}(C) = Y$ , and  $\text{ann}(D) = N$ , then the security DTD computed by Algorithm ANNOTATE VIEW will have the production  $D \rightarrow (A + B)^* A(A + B)$ , which is not equivalent to any 1-unambiguous regular expression [5].  $\square$

In practice, if we really need to use 1-unambiguous regular expressions, one could approximate the expressions generated by the algorithm with 1-ambiguous expressions that capture a larger language ([1] describes one method to do this).

**Example 6.2:** Fully annotated DTD depicted on Fig. 9 is the result of application of the algorithm ANNOTATE VIEW to partially annotated DTD of Fig. 4. Elements marked with # symbol are introduced artificially during the process of qualifier elimination (step 6) and inheritance from differently annotated parents (step 18). Fig. 10 represents attribute part of DTD. Note that artificial elements have security annotation N while initial elements are marked by Y.  $\square$

```

<!ELEMENT applications (application*,recommendation-letter*,
  ((rating,free-text)|(rating,free-text))* ,recommendation-letter*,
  ((rating,free-text)|(rating,free-text))*>
<!ELEMENT application (student-data,recommendation-letter*,
  ((rating,free-text)|(rating,free-text))* ,recommendation-letter*,
  ((rating,free-text)|(rating,free-text))*>
<!ELEMENT recommendation-letter (evaluator,
  ((rating,free-text)|(rating,free-text))*>
<!ELEMENT student-data (department,degree,name,waiver)>
<!ATTLIST student-data id CDATA #IMPLIED>
<!ELEMENT recommendation-letter (evaluator,
  ((rating,free-text)|(rating,free-text))*>
<!ELEMENT evaluator (title,institution,name)>
<!ELEMENT rating (MS,PhD,English)>
<!ELEMENT rating (MS,PhD,English)>
<!ELEMENT degree (#PCDATA)>
<!ELEMENT department (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT waiver (#PCDATA)>
<!ELEMENT institution (#PCDATA)>
<!ELEMENT PhD (#PCDATA)>
<!ELEMENT MS (#PCDATA)>
<!ELEMENT English (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT TXT (#PCDATA)>

```

Figure 11: DTD view  $D_v$

**Example 6.3:** The result of application of BUILD VIEW to DTD of Fig. 10 is depicted on Fig. 11. Note that all security related attributes (compared with input DTD annotation depicted on Fig. 4) are eliminated. Corresponding  $\sigma$ -function is represented on Fig. 12.  $\square$

Note, that  $\sigma$ -function has rules with contradictory conditions  $Q$  and  $not(Q)$  (e.g. 3–11, 14–22), therefore corresponding  $\sigma$  will always return empty set. The same is true also for rule 2 because it contains both condition  $not(P)$  and  $R$  with subcondition  $P$ . These rules can be eliminated on the process of optimization which is an open issue and is leaved for future work.

We now need a technical lemma.

**Lemma 6.2:** *Let  $(D, ann)$  be a security specification where  $D$  is a not-recursive DTD and  $(D_v, \sigma)$  be the security view that is constructed by Algorithms ANNOTATE VIEW and BUILD VIEW, for any sequence of element types  $B_0 \dots B_n$  in the full annotated  $D$  such that (i)  $B_{i+1}$  is a child type of  $B_i$  for  $i = 0 \dots n - 1$ , (ii) each  $B_i$  for  $i = 1 \dots n - 1$  is annotated N, there exists an XPath expression  $p$  and  $q_1 \dots q_n$  XPath qualifiers such that the following equation holds for all set of nodes  $N$ :*

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

$\square$

**Proof:** The proof is by a nested induction on  $n$  and the number of iteration of step 1 of algorithm BUILD VIEW.

For the base case,  $n = 1$ , then  $B_1$  is a child of  $B_0$ . Then, before step 1 of BUILD VIEW is executed, algorithm ANNOTATE VIEW would set  $\sigma(B_0 \rightarrow P(B_0), B_1) = B_1[q_1]$  for a suitable qualifier  $q_1$ . Therefore, up to that point of the execution of the algorithm, the theorem holds by setting  $p = \emptyset$ . During step 1 of algorithm BUILD VIEW it is possible that the elimination of some N-children of  $B_0$  would modify the selection function for  $B_1$ . By evaluating the  $\sigma(B_0 \rightarrow P(B_0), B_1)$  expression constructed by step 4 of BUILD VIEW with the  $\mathcal{S}_{\rightarrow}$  operator and by induction

```

applications -> P(applications):
 1 sigma(applications, application)= application[P]
 2 sigma(applications, recommendation-letter) = application[not(P)]/recommendation-letter[R]
 3 sigma(applications, rating) =
   application[ not(P)]/recommendation-letter[ not (Q)]/letter/favorable/rating[Q]
 4 sigma(applications, free-text) =
   application[ not(P)]/recommendation-letter[ not(Q)]/letter/favorable/free-text[Q]
 5 sigma(applications, rating) =
   application[ not(P)]/recommendation-letter[ not(Q)]/letter/unfavorable/rating[Q]
 6 sigma(applications, free-text) =
   application[ not(P)]/recommendation-letter[ not(Q)]/letter/unfavorable/free-text[Q]
 7 sigma(applications, recommendation-letter) =
   application[ not(P)]/unreliable/recommendation-letter[R]
 8 sigma(applications, rating) =
   application[ not(P)]/unreliable/recommendation-letter[not(Q)]/letter/favorable/rating[Q]
 9 sigma(applications, free-text) =
   application[ not(P)]/unreliable/recommendation-letter[not(Q)]/letter/favorable/free-text[Q]
10 sigma(applications, rating) =
   application[ not(P)]/unreliable/recommendation-letter[not(Q)]/letter/unfavorable/rating[Q]
11 sigma(applications, free-text) =
   application[ not(P)]/unreliable/recommendation-letter[not(Q)]/letter/unfavorable/free-text[Q]

application -> P(application):
12 sigma(application, student-data) = student-data
13 sigma(application, recommendation-letter) = recommendation-letter[R]
14 sigma(application, rating) = recommendation-letter[ not(Q)]/letter/favorable/rating[Q]
15 sigma(application, free-text) = recommendation-letter[not(Q)]/letter/favorable/free-text[Q]
16 sigma(application, rating) = recommendation-letter[not(Q)]/letter/unfavorable/rating[Q]
17 sigma(application, free-text) = recommendation-letter[not(Q)]/letter/unfavorable/free-text[Q]
18 sigma(application, recommendation-letter) = unreliable/recommendation-letter[R]
19 sigma(application, rating) = unreliable/recommendation-letter[not(Q)]/letter/favorable/rating[Q]
20 sigma(application, free-text) = unreliable/recommendation-letter[not(Q)]/letter/favorable/free-text[Q]
21 sigma(application, rating) = unreliable/recommendation-letter[not(Q)]/letter/unfavorable/rating[Q]
22 sigma(application, free-text) = unreliable/recommendation-letter[not(Q)]/letter/unfavorable/free-text[Q]

student-data -> P(student-data):
23 sigma(student-data, department) = department
24 sigma(student-data,degree) = degree
25 sigma(student-data, name) = name
26 sigma(student-data, waiver) = waiver

recommendation-letter -> P(recommendation-letter):
27 sigma(recommendation-letter, evaluator) = evaluator
28 sigma(recommendation-letter, rating) = letter/favorable/rating[Q]
29 sigma(recommendation-letter, free-text) = letter/favorable/free-text[Q]
30 sigma(recommendation-letter, rating) = letter/unfavorable/rating[Q]
31 sigma(recommendation-letter, free-text) = letter/unfavorable/free-text[Q]

evaluator -> P(evaluator):
32 sigma(evaluator, title) = title
33 sigma(evaluator, institution) = institution
34 sigma(evaluator, name)= name

rating -> P(rating):
35 sigma(rating, MS) = MS
36 sigma(rating, PhD) = PhD
37 sigma(rating, English) = English

free-text -> P(free-text):
38 sigma(free-text, TXT) = TXT
39 sigma(free-text, PDF)= PDF

where P = ./student-data[id=$login]
      Q = ancestor::*[self::application[./student-data[id=$login]/waiver/text()='true']]
      R = ancestor::*[self::application[./student-data[id=$login]]]

```

Figure 12:  $\sigma$ -function

hypothesis we get

$$\begin{aligned}
& \mathcal{S}_{\rightarrow} [\sigma(B_0 \rightarrow P(B_0), B_1)](N) &= \\
& \mathcal{S}_{\rightarrow} [\sigma(B_0 \rightarrow P(B_0), C)/\sigma(C \rightarrow P(C), B_1) \cup \sigma(B_0 \rightarrow P(B_0), B_1)](N) &= \\
& \mathcal{S}_{\rightarrow} [\sigma(B_0 \rightarrow P(B_0), C)/\sigma(C \rightarrow P(C), B_1)](N) \cup \mathcal{S}_{\rightarrow} [\sigma(B_0 \rightarrow P(B_0), B_1)](N) &= \\
& \mathcal{S}_{\rightarrow} [\sigma(B_0 \rightarrow P(B_0), C)/\sigma(C \rightarrow P(C), B_1)](N) \cup \mathcal{S}_{\rightarrow} [p_0](N) \cup \mathcal{S}_{\rightarrow} [B_1[q_1]](N) &= \\
& \mathcal{S}_{\rightarrow} [p_1](N) \cup \mathcal{S}_{\rightarrow} [B_1[q_1]](N) & \quad 18
\end{aligned}$$

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  $B_0 \dots B_n$  be the sequence of nodes and let  $B_i$  for  $i \in \{1 \dots n-1\}$  be the last node that is eliminated by step 1 of the algorithm BUILD VIEW. Since the DTD is not recursive neither  $\sigma(B_0 \rightarrow P(B_0), B_i)$ , nor  $\sigma(B_i \rightarrow P(B_i), B_n)$  can be changed by this step. By evaluating the  $\mathcal{S}_\rightarrow$  operator and by induction hypothesis we get:

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

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

**Remark 6.1** *In the statement of the lemma we have 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 6.3:** *Let  $(D, \text{ann})$  be a authorization specification,  $D$  is non-recursive, let  $(D_v, \sigma)$  the security view constructed by Algorithms ANNOTATE VIEW and BUILD VIEW. Let  $T$  be a document,  $T_A$  the authorized version of  $T$  and  $T_S$  the materialized version of  $T$  with respect to  $(D_v, \sigma)$ . Then  $T_A$  is isomorphic to  $T_S$ .*  $\square$

**Proof:** The proof is done by a top-down induction on  $T$ . The root of  $T$  is clearly in both  $T_A$  and  $T_S$ .

By induction, assume that  $n$  is of element type  $A$ , and is in both  $T_A$  and  $T_S$ . We must show that each child  $n$  in  $T_A$  is also a child of  $n$  in  $T_S$ , and vice versa. The result will then follow, as the order of the children of  $n$  is the same in both documents. Note, that for this to work it is essential that nodes in  $A$  should be ordered with the old order.

Let, therefore,  $m$  be a child of  $n$  in  $T_A$ , of type  $B$ . Assume, first, that  $m$  is a child of  $n$  in the original document  $T$ . Consider the fully annotated DTD  $(D_F, \text{ann}')$ . Since  $n$  is in  $T_S$ ,  $\text{ann}'(A) = Y$ . Since  $m$  is in  $T_A$ , it follows that  $\text{ann}(B)$  cannot be equal to  $N$ , and hence  $\text{ann}'(B) = Y$ , and so element type  $B$  is in  $D_v$ . Furthermore, if  $\text{ann}(B) = Q[q]$ , then  $q$  must hold at  $m$ .

We must show that  $m$  is in  $\mathcal{S}_\rightarrow [\sigma(A \rightarrow P(A), B)](\{n\})$ . Let  $p = \sigma(A, B)$ . The algorithm ANNOTATE VIEW initially sets  $p = B$  (step 2), may replace  $p$  by  $B[q]$  in step 12, and may add additional disjuncts in step 2 of algorithm BUILD VIEW. 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  $n, n_1, \dots, n_k, m$  ( $k \geq 1$ ) be the sequence of nodes in  $T$  from  $n$  to  $m$ , of element types  $B_1, \dots, B_k$ . Since these nodes are not present in  $T_A$ , each  $\text{ann}(B_i)$  ( $1 \leq i \leq k$ ) must be either undefined,  $N$  or  $Q[q_i]$ , with the qualifier in the latter case evaluating to false at  $n_i$ . Furthermore,  $\text{ann}(B)$  must be either  $Y$  or a qualifier  $Q[q]$  that evaluates to true at  $m$ , which implies that  $B$  is in  $D_v$ .

To show that  $m$  is in  $\mathcal{S}_\rightarrow[[\sigma(A, B)]](\{n\})$ , observe first that  $D_F$  contains element types  $B'_j$  whenever  $\text{ann}(B_j)$  is undefined or is a qualifier. For this part of the proof, we shall write  $B'_j$  as a synonym for  $B_j$  in the remaining case, when  $\text{ann}(B_j) = \text{N}$ . Whenever  $\text{ann}(B_i)$  is  $\text{Q}[q_i]$ , step 12 of the algorithm ANNOTATE VIEW initially sets  $\sigma(B'_{i-1}, B'_i)$  to  $B_i[\neg q]$  (writing  $B'_0 = A$ , for convenience); when  $\text{ann}(B_i) = \text{N}$  or is undefined,  $\sigma(B'_{i-1}, B'_i)$  is initially set equal to  $B'_i$  in step 2 of ANNOTATE VIEW. Finally, step 1 of BUILD VIEW deletes element types  $B'_1, \dots, B'_k$ , replacing  $\sigma(A, B)$  by a disjunction of paths, and by lemma 6.2 we get:

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

with some of the  $q_i$ 's absent, when  $\text{ann}(B_i)$  is  $\text{N}$  or undefined. It follows that  $m \in \mathcal{S}_\rightarrow[[\sigma(A, B)]](\{n\})$ , as desired.

For the converse, let  $m$  be a child of  $n$  in  $T_S$ . We must show that  $m$  is a child of  $n$  in  $T_A$ .

From the definition of  $T_S$ ,  $m$  must be in the result of evaluating  $\sigma(A, B)$  at  $n$ . Let  $n = n_0, n_1, \dots, n_k, m = n_{k+1}$  ( $k \geq 0$ ) be the shortest path from  $n$  to  $m$  that is used in the evaluation of the  $\sigma$  function, and let  $\sigma'$  be the value of the  $\sigma$  function after application of the algorithm ANNOTATE VIEW. We claim that  $n_{i+1} \in \mathcal{S}_\rightarrow[[\sigma(B_i, B_{i+1})]](\{n_i\})$  ( $0 \leq i \leq k, B_0 = A, B_{k+1} = B$ ). We show this by induction on the last  $B_i$  eliminated in step 1 of BUILD VIEW: this step replaces  $\sigma(B_{i-1}, B_{i+1})$  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  $n_{i+1}$  was in the second disjunct above, we would have a contradiction with assumption that our path was the shortest. Therefore we have  $n_{i+1} \in \mathcal{S}_\rightarrow[[\sigma(B_i, B_{i+1})]](\{n_i\})$  and  $n_i \in \mathcal{S}_\rightarrow[[\sigma(B_{i-1}, B_i)]](\{n_{i-1}\})$ , proving our claim. We therefore know that  $\sigma(B_{i-1}, B_i)$  is

1.  $B_i$  when  $\text{ann}(B_i)$  is either  $\text{N}$  or undefined. The case  $\text{ann}(B_i) = \text{Y}$  is impossible except when  $i = k + 1$ , as the element type in question is deleted in step 1 of BUILD VIEW.
2.  $B_i[\neg q]$  when  $\text{ann}(B_i)$  is  $B_i[\neg q_i]$ .

In both case, it follows that  $n, m_1, \dots, m_k, m$  is a path in  $T$ . It remains to show that  $m_1, \dots, m_k$  are deleted in  $T_A$ . For nodes annotated with a qualifier, this is immediate; for 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.  $\square$

The complexity of the algorithm is as follows:

**Theorem 6.4:** *Let  $(D, \text{ann})$  be a authorization specification for a non-recursive DTD, let  $P$  be size of the largest production rule in  $D$ . Let  $n_Y$  be the number of element types annotated with  $\text{Y}$ , and let  $n_{\text{other}}$  the number of element types otherwise annotated or not annotated. Then the size of the select function  $\sigma$  generated by the algorithm is bounded by  $O(n_{\text{other}} \times |\text{ann}|)$  and the size of the View DTD  $D_v$  is bounded by  $O(n_Y \times P^{n_{\text{other}}+1})$ .*  $\square$

**Proof:** For the first bound observe that the introduction of the symbols  $/$  and  $\cup$  in the definition of  $\sigma$  only happens when eliminating an element type labelled with  $\text{N}$  in the fully annotated DTD, and there are at most  $n_{\text{other}}$  element types of this sort. All qualifiers appearing the  $\sigma$  are the same as qualifiers that were in the original authorization specification, or their negations, and therefore their size is bounded by  $|\text{ann}|$ .

For the second step observe that we only replace occurrences of  $\text{N}$ -element types in a regular expression of a  $\text{Y}$ -labelled element type with another regular expression, and that each of those replacement eliminates a  $\text{N}$ -labelled node.  $\square$

The above upper bound is tight as the following example shows:

**Example 6.4:** Consider DTD with the production  $\text{root} \rightarrow A_0$  and  $A_i \rightarrow A_{i+1}A_{i+1}$  for  $i = 0 \dots n - 1$  and where  $\text{ann}(A_0) = \text{N}$ ,  $\text{ann}(A_n) = \text{Y}$ . Then the DTD View  $D_v$  has only one rule

$$\text{root} \rightarrow \overbrace{A_n \dots A_n}^{2^n \text{ times}},$$

and the select function is  $\sigma(\text{root}, A_n) = A_0 / \dots / A_n$ . □

## 7 Other security policies

Our 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* policy [10]. 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* (De Capitani di Vimercati and Samarati [10]). 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 7.1:** A policy is *complete* and *consistent* if every partially annotated tree can be extended to a fully annotated tree. □

We list here several possible policies. These are variations of classical security policies that are used in other settings ([10]).

We have identified a number of policies for value propagation and conflict resolution:

**Local Propagation Policy:** “open”, “closed”, or “none”;

**Hierarchy Propagation Policy:** “topDown”, “bottomUp”, or “none”;

**Structural Conflict Resolution:** “localFirst”, “hierarchyFirst”, or “none”;

**Value Conflict Resolution:** “denialTakesPrecedence”, “permissionTakesPrecedence”, or “none”.

The Local Propagation Policy 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.

The Hierarchy Propagation Policy specifies node annotation inheritance in the tree. In the case of “topDown”, an unlabelled node with a labelled parent inherits the label of its parent. In the case of “bottomUp” an unlabelled node inherits the label from a labelled children. Note that the “bottomUp” case can result in conflicts, and they should be addressed by the Value Conflict Resolution Policy.

The Structural Conflict Resolution Policy specifies whether the local or hierarchy rule takes precedence (“localFirst” or “hierarchyFirst” respectively); while “none” means that the choice depends on the values and on the Value Conflict Resolution Policy. 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 (“denialTakesPrecedence”); Y always has precedence over N (“permissionTakesPrecedence”), and no choice (“noneTakesPrecedence”).

In the sequel we show some sufficient conditions for complete and consistent policy combinations. We start with some policies that we can term *topDown*.

**Proposition 7.1:** All policies that satisfy one of the conditions of table 7.1 are sound and complete. □

**Proof:** Assume that  $T$  is a partially annotated tree. We show that the annotation can be extended to a full tree.

Consider condition 1 of *topDown* security policy of table 7.1.

hierarchy	local	structural conflict	value conflict	condition
topDown	≠none	hierarchyFirst	*	*
topDown	none	*	*	root is annotated

Table 1: topDown policy conditions

hierarchy	local	structural conflict	value conflict	condition
bottomUp	≠none	hierarchyFirst	≠none	*
bottomUp	none	*	≠none	leaves are annotated

Table 2: bottomUp policy conditions

Base case: if the root is annotated then we are done. If it is not annotated then according to the definition it can obtain its annotation from *local* security policy: Y/N if *local=open/closed* respectively. Thus root annotation is defined.

Inductive case: consider an arbitrary node  $n$  with annotated parent  $p$ . If  $n$  is annotated we are done. Otherwise,  $n$  obtains its annotation from the parent since *structural conflict=hierarchyFirst*. Thus annotation of any node is defined.

Consider condition 2 of *topDown* security policy of table 7.1.

Base case: the root is annotated.

Inductive case: consider an arbitrary node  $n$  with annotated parent  $p$ . If it is annotated we are done. Otherwise it obtains its annotation from the parent. Thus annotation of any node is defined  $\square$

Next we have some policies that we can term *bottomUp*.

**Proposition 7.2:** *All policies that satisfy one of the conditions of table 7.2 are sound and complete.*  $\square$

**Proof:** . Assume that  $T$  is a partially annotated tree. We show that the annotation can be extended to full tree.

Consider condition 1 of *bottomUp* security policy of table 7.2.

Base case: if the children are annotated then we are done. If some of them are not annotated then according to the condition they can obtain their annotation from *local* security policy: Y/N if *local=open/closed* respectively. Thus annotation of all leaves is defined.

Inductive case: consider an arbitrary node  $n$  with all annotated children. If  $n$  is annotated we are done. Otherwise,  $n$  obtains its annotation from the children since *structural conflict=hierarchyFirst*. However, different children can have different annotation. On the other hand, *value conflict≠nothingTakesPrecedence* can be used to define “winning” label. Thus annotation of any node is defined.

Consider condition 2 of *bottomUp* security policy of table 7.2.

Base case: all leaves are annotated.

Inductive case: consider an arbitrary node  $n$  with all annotated children. If it is annotated, we are done. Otherwise it obtains its annotation from the children. However, different children can have different annotation. On the other hand, *value conflict≠nothingTakesPrecedence* can be used to define “winning” label. Thus annotation of any node is defined.  $\square$

Now we consider some policies that we can term *local*.

hierarchy	local	structural conflict	value conflict
*	≠none	localFirst	*
none	≠none	*	*

Table 3: local policy conditions

hierarchy	local	structural conflict	value conflict
≠none	≠none	noneFirst	≠none

Table 4: multilabel policy conditions

**Proposition 7.3:** *All policies that satisfy one of the conditions of table 7.3 are sound and complete.* □

**Proof:** Assume that  $T$  is a partially annotated tree. We show that the annotation can be extended to full tree.

Consider condition 1 of *local* security policy of table 7.3. Since *structural conflict* = localFirst, *local* is enforced in the first turn.

Consider case 2 of *local* security policy of table 7.3. Since *hierarchy* security policy is not defined, *local* is enforced.

Thus, for each not annotated node  $n$ , we enforce *local* security policy that assigns either a label Y or N depending on *local* policy definition. □

In some cases both *hierarchy* security policy and *local* security policy are defined, but *structural conflict* security policy is “noneFirst”. In these cases we apply both *hierarchy* and *local* security policy thus obtaining for each node a set of more than one security annotation. So it is not really clear from the user specification what is really wanted. The “winning” label is defined by means of *value conflict* security policy which should not be equal to “noneTakesPrecedence”. We call such policies “resolvable multilabel” security policies.

**Proposition 7.4:** *All policies that satisfy one of the conditions of table 7.4 are sound and complete.* □

**Proof:** Assume that  $T$  is a partially annotated tree. We show that the annotation can be extended to full tree.

Since *structural conflict* is not defined but both *hierarchy* and *local* are not “none”, we enforce both of them independently (in cases when it is possible, e.g. if *hierarchy* = topDown but root is not annotated, we cannot enforce *hierarchy* from the root; however, we can start enforcement of *hierarchy* policy from any annotated node, because explicitly defined label overrides propagated one). As the result, for each node we will receive a set of labels. Since *value conflict* is defined, it can be used for defining the “winning” label.

Since each node is assigned at least one label (considering *local* policy), partial annotation can be extended to full annotation. □

**Remark 7.1** *All the other policies are classified as unresolvable: considering the table 7.1, policies following condition in line 1 are incomplete, policies in lines 2 and 3 of are inconsistent, policy in line 4 may be either inconsistent or incomplete.*

Extending the security view approach to other policies, such as these, requires modifying the construction of the security view so that it propagates annotations in a way that corresponds to annotation propagation in the security policy. We leave this to the future work.

hierarchy	local	structural conflict	value conflict
none	none	*	*
≠none	≠none	none	none
bottomUp	*	hierarchyFirst	none
bottomUp	none	≠hierarchyFirst	none

Table 5: unresolvable policy conditions

## 8 Extending to Recursive DTDs

The restriction in our current proposal is the requirement of nonrecursive nature of DTDs. For authorization specification of recursive DTD it is possible to derive a fully annotated DTD by modifying step 18 of the algorithm ANNOTATE VIEW, but one cannot construct a select function in XPath that guarantees both secrecy and availability by modifying step 1 of the algorithm BUILD VIEW.

The reason for this is that to handle with recursive DTDs correctly, one should repeat step 18 of ANNOTATE VIEW until a fix point is reached. Then, if there are still unlabelled nodes they are part of a cycle of completely unlabelled nodes. We could then consider all entry points of the cycle, and apply step 18 of ANNOTATE VIEW to all entry points at the same time: if all generators of entry points outside the cycle are Y-nodes then all nodes of the cycle can be labeled Y. The case for N is similar. In the case of conflicts, apply step 24 of ANNOTATE VIEW to all entry points of the cycle at the same time. This process breaks progressively more cycles until all cycles get labelled.

The problem, however, is that XPath lacks the full Kleene-star operator. Thus we cannot select exactly the nodes in which an element must be reached just after a particular loop is traversed an arbitrary number of times. It may be possible to extend the security view with “dummy nodes” that map to epsilon rules, and obtain the desired result, but such a solution would not be acceptable as the schema would be meaningless to the user. Using the present algorithm, we can obtain an approximate solution: *by stopping the modified ANNOTATE VIEW-algorithm after a finite number of iterations of step 1 of BUILD VIEW-algorithm we have a secrecy preserving view.*

The problem, however, is that XPath lacks the full Kleene-star operator. XPath language. Thus we cannot select exactly the nodes in which an element must be reached just after a particular loop is traversed an arbitrary number of times. It may be possible to extend the security view with “dummy nodes” that map to epsilon rules, and obtain the desired result, but such a solution would not be acceptable as the schema would be meaningless to the user. Using the present algorithm, we can obtain an approximate solution: *by stopping the modified ANNOTATE VIEW-algorithm after a finite number of iterations of step 1 of BUILD VIEW-algorithm we have a secrecy preserving view.*

## 9 From XPath Policies to Views

In the XML access control literature we often find policies over XML documents expressed as XPath queries [8, 9] tagged to principals<sup>2</sup>. If the principal is matched to the actual user making the request then the XPath query is used to select the subset of nodes that will be labelled with some security attribute.

Formally, we can say that a policy is a collection of tuples of the form  $\langle p, q, l, u \rangle$  where  $p$  and  $q$  are XPath expressions,  $l$  is some security label, and  $u$  is an expressions denoting a set of principals.

Typically, the security label  $l$  describes some hierarchical access property such as “forbidden at schema level”, “permitted at document level”, “recursively forbidden” etc. This labelling is accompanied by an algebra that specify what

<sup>2</sup>We prefer the notion of principals as it subsumes notions such as roles, groups, users, ip addresses etc. leaving the word users for actual final users.

final Y/N label must be attached to a document node that receives multiple labels. See for instance [8].

As for principals we can have fairly complicated lattices of principals describing how one inherits the permissions (or prohibitions) from other principals and sophisticated matching functions associating users to principals. See for instance [9].

However, in the current XML access control proposals the access control decision is always split in a two tiered process: (i) get all applicable policies to the current requester, (ii) apply the policies to the XML document sending the result to the requester. Since there is no need to recast here step (i) for which there are already a number of fairly sophisticated proposals we focus on the more interesting second step. We assume here only the plain Y, N labels, because a hierarchy of labels can be conceptually treated by labelling the document independently for each label and then use a local function that evaluates the algebra of labels at each node.

The interesting result is to show how a security policy expressed as a set of XPath expression can be casted in our framework. In general it is not possible to cast a security policy to a security annotation of a DTD. However, it is possible to construct the view corresponding to a set of XPath-expressions.

**Definition 9.1:** A *XPath security annotation* for a DTD  $D$  is a set of triples of the form  $\langle p, q, l \rangle$  where  $p$  and  $q$  are XPath expressions defined over  $D$  and  $l$  is either Y or N. □

The intuitive meaning of each triple is arrive at the node reachable through  $p$  and then start labelling all nodes reachable by  $q$  from there. Conceptually the nodes are those reachable by  $p/q$  but this format is more convenient because we might need to have different labelling depending on the policy at hand.

As usual we also assume that a security specification an XPath security annotation couple with a policy to resolve ambiguities.

**Example 9.1:** The XPath security annotation

$$\{\langle \text{/applications/application/student-data}, //, Y \rangle, \langle \text{/applications}, //, N \rangle\}$$

describes a labelling policy such that all nodes reachable from `/applications/application/student-data` should be labelled with Y, and all nodes reachable after `/applications` should be labelled with N. This policy generates a conflict on `/applications/application/student-data` subtree. Depending on the policy that we adopt we can get a number of different results. If *denial takes precedence* apply, the entire tree is labelled N. If *most specific takes precedence* apply the tree is labelled with N, except for the subtree rooted at node reachable by path `/applications/application/student-data` which is labelled Y. □

**Example 9.2:** The XPath security annotation described by

$$\{\langle \text{/applications/application/student-data}, Y, \epsilon \rangle, \langle \text{/applications}, N, // \rangle\}$$

labels the node reachable by `/applications/application/student-data` with Y and all nodes reachable after `/applications` with N. Here the policy also makes a difference: Damiani et al. [9] would have got:

- `/applications/application/student-data` labelled with Y
  - all the rest labelled with N (including also nodes below `/applications/application/student-data`) using the rule most specific takes precedence.
- 

Furthermore we restrict  $p$  to be qualifier-free. We give here an algorithm for non-recursive DTDs that captures at schema level the document-level labelling policy of Damiani et al. [8] which is a variant of the closed, topDown (there

called `mostSpecificTakePrecedence`), `denialTakesPrecedence` security policy combination. For each node  $n$  and XPath expression  $p$  we need the information of the predecessors of  $n$  along  $p$ , i.e. while unrolling  $p$  we need to keep track of the last step of the paths that we actually used to reach  $n$ .

In the following we make two restriction on the XPath expressions. At first, in the literature we only have two possibilities for  $q$ :  $\epsilon$  (label this node) or  $//$  (label recursively all nodes below this node). Therefore we restrict  $q$  accordingly.

**Definition 9.2:** Let  $\mathcal{XP}$  be an XPath security annotation for a DTD  $D$ , and  $T$  an XML document conforming to  $D$ . We denote  $N_p$  a set of nodes reached from root via  $p$ ,  $N_{pq}$  a set of nodes reached from  $N_p$  via  $q$ . The *XPath filtered version*  $T_{xp}$  of  $T$  is obtained from  $T$  as follows:

**Typing.** Evaluate each triple  $\langle p, l, q \rangle$  so that each element in  $N_{pq}$  is labelled  $l$ . If the element is already labelled by a label  $l' \neq l$  then label it with N.

**Propagation.** For each triple  $\langle p, l, q \rangle$  except those of the form  $p, //, l$  construct tuple  $\langle p/q, l, // \rangle$ . Evaluate constructed tuple so that each descendant of each element of  $N_{pq}$  is labelled  $l$ . If the element is already labelled by a typing step skip it, if it is labelled by a propagation and has a label  $l' \neq l$  then label it with N.

**Local policy.** For each unlabelled node, label it with N.

**Filtering.** Delete all of the nodes labelled with N from the result, making all children of a deleted node  $v$  into children of  $v$ 's parent.

□

According to the Def. 9.2 we can construct algorithm of filtering an XML tree  $T$ . Algorithm scans all policies of the form  $\langle p, l, q \rangle$  and applies typing operation to the nodes of the set  $N_{pq}$ , i.e. assigns *most specific* labels that cannot be overwritten via propagation. After typing operation we have partially annotated tree. After that each tuple  $\langle p, l, q \rangle$  is used to construct  $\langle p/q, l, // \rangle$ , i.e. labelling will be applied to all descendants of typed nodes. Thus we apply *top down propagation* of labels obtained at the typing step. Then we apply the closed policy to complete the labelling and prune the results as in the DTD View Algorithm. Algorithm implementing filtering an XML tree according to the Def. 9.2 is depicted on Fig. 13.

Intuitively each \*-node marks a point where we have an XPath expression that says “from here label everything down”. So we have a *most specific* expression assigning a value on the node. The `topDown` policy then automatically takes care of the `mostSpecificTakes` precedence. Note also that we may not have yet labelled all nodes as we have not labelled the root with Y.

**Theorem 9.1:** Let  $\mathcal{P}$  be an XPath security specification and  $(D_v, \sigma)$  the security view that is constructed by Algorithm XPATH 2 VIEW. Then for every document  $T$ , the XPath filtered version  $T_{xp}$  is identical to the materialized version  $T_S$ . □

## 10 Implementation

We have implemented a preliminary version of a Java tool that outputs a “sanitized” XML document, i.e. document that contains only permitted nodes and the DTD view.

Firstly, we use Xerces Java DOM parser<sup>3</sup> and Wutka DTD parser<sup>4</sup>. The latter we modified to distinguish security

<sup>3</sup><http://xml.apache.org/xerces2-j/>

<sup>4</sup><http://www.wutka.com/dtdparser.html>

**Algorithm:** XPATH 2 VIEW

**Input:**  $\mathcal{X}\mathcal{P}$  an XPath security annotation for a DTD  $D$ , and  $T$  an XML document conforming to  $D$

**Output:** XPath filtered version  $T_{xp}$  of  $T$  is obtained from  $T$

```
1: calculate  $N_p$  (set of nodes reachable from root via  $p$ )
2: calculate  $N_{pq}$  (set of nodes reachable from  $N_p$  via  $q$ )
3: for each  $n \in N_{pq}$  do
4:   if  $n$  is not labelled then
5:     label it with  $l$  and *
6:   else if  $n$  is labelled with  $l' \neq l$  then
7:     label it with N and *
8:   for each descendant of  $n$  do
9:     if  $n$  is not labelled then
10:      label it with  $l$ 
11:     else if  $n$  is labelled with  $l' \neq l$  and  $n$  is not labelled with * then
12:      label it with N
```

Figure 13: XPath-view construction

policy attributes located at root element and security annotations over the rest of DTD.

Then partially annotated DTD is extended to a full annotated one according to the algorithm ANNOTATE VIEW. Next we apply BUILD VIEW to produce  $D_v$  which is used to materialize view of XML document  $T_S$  according to the algorithm MATERIALIZE.

**Example 10.1:** Fig. 14 shows an initial XML document corresponding to DTD of Fig. 4. Fig. 15 and Fig. 16 represent XML view for user with login “dkonovalov” and “vromanov” respectively. Both views correspond to DTD view of Fig. 11 and are extracted by means of  $\sigma$ -function of Fig. 12 during application of algorithm MATERIALIZE.

We should note, that each student has an access only to relevant data, i.e. Dmitry Konovalov with login “dkonovalov” is not able to see the data of Vladimir Romanov having login “vromanov” and vice versa. Moreover, Vladimir Romanov is forbidden to see the content recommendation letters except of the names of his evaluators while Dmitry Konovalov has an access to full content of all recommendation letters. This is because the former student didn’t waive his right to inspect the content of recommendation letters ( $waiver=“false”$ ) while the latter did. Furthermore, no one student is permitted to see elements `unreliable`, `reason`, `letter`, `favorable`, `unfavorable`.  $\square$

## 11 Related Work and Conclusions

A number of security models have been proposed for XML (see [13] for a recent survey). Specifying security constraints with XPath on top of document DTDs was discussed in [9]. The semantics of access control to a user is a specific view of the document determined by the XPath access-control rules. A view derivation algorithm is based on tree labelling. Issues like granularity of access, access-control inheritance, overriding, and conflict resolution are studied in [3, 9].

A different approach is explored in [6]. In a nutshell, access annotations are explicitly included in the actual element nodes in XML, whereas DTD nodes specify “coarse” conditions on the existence of security specifications in corresponding XML nodes. Only elements with accessible annotations appear in the result of a query.

Stoica and Farkas [22] proposed to produce single-level views of XML when conforming DTD is annotated by labels of different confidentiality level. 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 than 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.

This paper elaborates on certain issues left open in [12]. In particular, we studied access control and security specifications defined over general DTDs in terms of regular expressions rather than normalized DTDs of [12]. 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.

Several extensions to the security model are targeted for future work. First, we plan to extend the definitions of security views and authorization specifications by supporting more complex XML Schema [11] instead of DTDs. Second, we are also studying extensions of our algorithm for deriving security-view definitions with respect to recursive DTDs/schemas. Third, we intend to investigate the expressive power and complexity of access-control specifications and security views by varying the fragments of XPath used in the definitions. Finally, our next step toward enforcing inference control will be to investigate reasoning techniques in the presence of integrity constraints and ID/IDREF attributes.

**Acknowledgments.** This project has been partially supported by the MIUR-FIBR project ASTRO and the MIUR-COFIN "Web-based management and representation of spatial and geographical data".

## References

- [1] H. Ahonen. Disambiguation of SGML content models. In *Proceedings of PODP*, Lecture Notes in Computer Science, pages 27–37, 1996.
- [2] M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. In *Proceedings of the International Conference on Database Theory*, 2003.
- [3] E. Bertino and E. Ferrari. Secure and selective dissemination of XML documents. *ACM Transactions on Information and System Security*, 5(3):290–331, 2002.
- [4] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0*. W3C, Feb. 1998.
- [5] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, pages 182–206, 1998.
- [6] S. Cho, S. Amer-Yahia, L. Lakshmanan, and D. Srivastava. Optimizing the secure evaluation of twig queries. In *Proceedings of the International Conference on Very Large Data Bases*, 2002.
- [7] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation. <http://www.w3.org/TR/xpath>, November 1999.
- [8] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Securing XML documents. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2000.
- [9] 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*, 5(2):169–202, 2002.
- [10] S. De Capitani di Vimercati and P. 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.

- [11] D. C. Fallside and P. Walmsley. XML Schema Part 0: Primer Second Edition. W3C Recommendation. <http://www.w3.org/TR/xmlschema-0/>, 2004.
- [12] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 587–598. ACM Press, 2004.
- [13] I. Fundulaki and M. Marx. Specifying access control policies for XML documents with XPath. In *Proceedings of the 9th ACM symposium on Access control models and technologies*, pages 61–69. ACM Press, 2004.
- [14] S. K. Goel, C. Clifton, and A. Rosenthal. Derived access control specification for XML. In *Proceedings of the 2nd ACM Workshop On XML Security*, pages 1–14. ACM Press, 2003.
- [15] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithm for processing XPath queries. In *Proceedings of the International Conference on Very Large Data Bases*, 2002.
- [16] S. Hada and M. Kudo. XML Access Control Language: Provisional Authorization for XML Documents. <http://www.trl.ibm.com/projects/xml/xacl/>, 2000.
- [17] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman, and W. R. Shockley. The SeaView security model. *IEEE Transactions on Software Engineering*, 16(6):593–607, 1990.
- [18] G. Miklau and D. Suciu. Controlling access to published data using cryptography. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 898–909, September 2003.
- [19] M. Murata, A. Tozawa, M. Kudo, and S. Hada. XML access control using static analysis. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 73–84. ACM Press, 2003.
- [20] X. Qian. View-based access control with high assurance. In *Proceedings of the 15th IEEE Symposium on Security and Privacy*, pages 85–93. IEEE Computer Society Press, 1996.
- [21] P. D. Stachour and B. Thuraisingham. Design of LDV: A multilevel secure relational database management system. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):190–209, 1990.
- [22] A. Stoica and C. Farkas. Secure XML views. In *Research Directions in Data and Applications Security, IFIP WG 11.3 Sixteenth International Conference on Data and Applications Security*, volume 256, pages 133–146. Kluwer, 2003.
- [23] J. Wang and S. 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*, pages 70–77. ACM Press, 2004.

```

<?xml version='1.0'?>
<!DOCTYPE applications SYSTEM 'input.dtd'>
<applications>
  <application>
    <student-data id='dkonovalov'>
      <department>CS</department>
      <degree>PhD</degree>
      <name>Dmitry Kononov</name>
      <waiver>true</waiver>
    </student-data>
    <recommendation-letter>
      <evaluator>
        <title>Full Professor</title>
        <institution>University of Suncity</institution>
        <name>Albert Wasserman</name>
      </evaluator>
      <letter>
        <unfavorable>
          <rating>
            <MS>average</MS><PhD>not recommended</PhD><English>below average</English>
          </rating>
          <free-text>
            <TXT>link to txt-file goes here</TXT>
          </free-text>
        </unfavorable>
      </letter>
    </recommendation-letter>
    <unreliable>
      <recommendation-letter>
        <evaluator>
          <title>Researcher</title>
          <institution>Magnificent Labs</institution>
          <name>Maria Shaker</name>
        </evaluator>
        <letter>
          <favorable>
            <rating>
              <MS>outstanding</MS>
              <PhD>highly recommended</PhD>
              <English>outstanding</English>
            </rating>
            <free-text>
              <PDF>link to pdf-file goes here</PDF>
            </free-text>
          </favorable>
        </letter>
      </recommendation-letter>
      <reason>The recommender does not exist.</reason>
    </unreliable>
  </application>

  <application>
    <student-data id='vromanov'>
      <department>CS</department>
      <degree>PhD</degree>
      <name>Vladimir Romanov</name>
      <waiver>false</waiver>
    </student-data>
    <unreliable>
      <recommendation-letter>
        <evaluator>
          <title>Researcher</title>
          <institution>Magnificent Labs</institution>
          <name>Maria Shaker</name>
        </evaluator>
        <letter>
          <favorable>
            <rating>
              <MS>outstanding</MS>
              <PhD>highly recommended</PhD>
              <English>outstanding</English>
            </rating>
            <free-text>
              <PDF>link to pdf-file goes here</PDF>
            </free-text>
          </favorable>
        </letter>
      </recommendation-letter>
      <reason>The recommender does not exist.</reason>
    </unreliable>
  </application>
</applications>

```

```

<applications>
  <application>
    <student-data id="dkonovalov">
      <department>CS</department>
      <degree>PhD</degree>
      <name>Dmitry Konovalov</name>
      <waiver>true</waiver>
    </student-data>
    <recommendation-letter>
      <evaluator>
        <title>Full Professor</title>
        <institution>University of Suncity</institution>
        <name>Albert Wasserman</name>
      </evaluator>
      <rating>
        <MS>average</MS>
        <PhD>not recommended</PhD>
        <English>below average</English>
      </rating>
      <free-text>
        <TXT>link to txt-file goes here</TXT>
      </free-text>
    </recommendation-letter>
    <recommendation-letter>
      <evaluator>
        <title>Researcher</title>
        <institution>Magnificent Labs</institution>
        <name>Maria Shaker</name>
      </evaluator>
      <rating>
        <MS>outstanding</MS>
        <PhD>highly recommended</PhD>
        <English>outstanding</English>
      </rating>
      <free-text>
        <PDF>link to pdf-file goes here</PDF>
      </free-text>
    </recommendation-letter>
  </application>
</applications>

```

Figure 15: XML view for student Dmitry Konovalov

```

<applications>
  <application>
    <student-data id="vromanov">
      <department>CS</department>
      <degree>PhD</degree>
      <name>Vladimir Romanov</name>
      <waiver>false</waiver>
    </student-data>
  </application>
</applications>

```

Figure 16: XML view for student Vladimir Romanov