Access Control and Synchronization in XML Documents

Stefan Böttcher , Adelhard Türling
University of Paderborn , FB 17 (Mathematik-Informatik)
Fürstenallee 11 , D-33102 Paderborn , Germany
email : stb@uni-paderborn.de, Adelhard.Tuerling@uni-paderborn.de

Abstract: Whenever multiple users modify XML documents concurrently, access control and synchronization of read and update operations are required. We suggest to define access rights and locks at the level of node names that are defined in the DTD of the XML document. This allows us to normalize (a subset of) XPath expressions, used e.g. in queries, such that access control and synchronization can be performed at the level of node names too. Then we show that access control and locking at the level of node names can be solved by the same technique, i.e. the identification of corresponding node names of XPath expressions. Finally, we extend our approach to predicate filters that may be used in XPath expressions for both, access control and synchronization in order to qualify a selected subset of nodes, and present a fast predicate evaluator for these predicates.

Keywords: XML documents, XPath, access control, synchronization

1. Introduction

1.1. Problem origin and motivation

There is a growing interest in XML as a data representation language and data exchange format for different enterprise applications. Whenever XML data is read and modified by multiple users, then both, access control and synchronization of the clients' access to XML data are important topics [1],[5],[15]. Our work on access control and synchronization is motivated by the development of applications for an internet company, which require that multiple parties read and modify the same huge XML document concurrently. Common to our application programs is that XPath expressions can be used, in order to describe, which fragment of an XML document shall be read or written. Furthermore, we have different user groups, for which we specify access rights that limit the fragments of the XML document that may be read or written. Therefore, it appears natural to define access rights for read or write access on parts of the XML document with (a subset of) XPath expressions.

1.2. Relation to other work and our focus

Access control and synchronization of operations on XML documents have been investigated separately in recent contributions, whereas we present an integrated approach for key aspects of both problems.

Most contributions to access control on XML documents focus on policies [12,2,6,1]. They cover different aspects of access control, covering e.g. user groups, document location in the web, override policies and access control for fragments of XML documents [12,2]. Access on fragments of XML documents can be defined on the context of the document itself or on the structure of the XML document. Within these approaches, there is a trend towards fine-grained access rights, that include and go beyond individual access rights at the level of attributes or elements [16,5]. We follow this direction and allow not only for fine-grained access rights on the level of nodes, but also include predicate filters as can be specified by XPath queries. While the other approaches focus on transparency of the access control and use access control as an additional filter, our approach has the goal to give complete answers to the users and reject each access that violates access rights.
Concurrency control for concurrent operations working on XML documents has e.g. been discussed in [4]. One approach is to store the information contained in the XML document in a database and to use the database system for the purpose of synchronization. This can be achieved by either mapping the XML document and updates of this XML document to a database (see for example Updating XML [15] and Clock [17]) or by using an XML database system such as Lore [10], Natix [11], Strudel [9] or Tamino [14]. Another approach is to use a separate environment for the storage and manipulation of XML documents [7,13,8]. Since our contribution does not rely on the existence of a database system, it is compatible to both approaches.
Finally, our approach integrates ideas from predicate logic and predicative transaction synchronization [3] and provides a uniform approach to access control and synchronization. But in contrast to contributions from the database area, we use a subset of XPath in order to access, lock and query document fragments.

2. Problem description

2.1. Access control and synchronization

There are two problems to be solved: access control and synchronization. For the definition of access rights, we follow approaches of [5,14] and define for each user group in advance, which part of the XML document may be read and which part may be written. Access rights combine an access mode (read or write) with an XPath expression, whereas for a given XML document the XPath expression describes that XML fragment that may be accessed at most. For example, the write access rights for product marketing people could be described by the following XPath expression

‘ // Product / Marketing // * ’

which means that within the XML document they have write access to the whole subtree starting with a node ‘Marketing’ which is directly located under a node ‘Product’.
Access control has to check that no user accesses an XML fragment data beyond his rights, i.e. the node set of any given XML document that is accessed by a user’s XPath expression must be a subset of the node set accessible according to the XPath expression describing his access right. Note that for our application, it is essential that access violations are detected and e.g. produce an access violation. Since users with different

access rights have to cooperate, but decide on the basis of the data that they read, it is *not acceptable* in our applications that different users get different answers to the same query. Therefore, we can not rely on contributions that use an access right as an additional filter to restrict the query results, as e.g. [16,5].
Furthermore, we assume that every access to XML data is described at the application side in terms of XPath expressions, i.e. that the only way, the application tries to query an XML document is by using XPath expressions. Similarly, for each write operation we require the application to provide the XPath expression that describes the XML fragment that is to be modified. These XPath expressions are communicated to our access control and synchronization subsystems and are used within our algorithms for access control and synchronization.
Synchronization has to guarantee that no fragment of the XML document is currently accessed by two users with at least one of them requiring an exclusive access right.

2.2. The subset of allowed XPath expressions

Since the richness of XPath raises the problem of intractable expressions, we limit the considered XPath expressions for access rights, queries, write operations and locks to a subset called **allowed XPath expressions** by the following rules that restrict axes, node tests and predicate filters:

1. Axe specifiers
We allow for *absolute* or *relative location paths* with the following *axes specifiers* in their *location steps*: self, parent, ancestor, ancestor-or-self, child, descendant, descendant-or-self, namespace and attribute, and we forbid the following (-sibling) and preceding (-sibling) axes.

2. Node tests
We allow all *node name tests* including the wildcards * and name-spaces, but we forbid *node type tests* like text(), comment(), processing-instruction() and node().
For example, when A is an attribute and E is an element, then @A , ./E , ../E, //*/E are allowed XPath expressions.

3. Predicate filters
We restrict *predicate filters* [B] of allowed XPath expressions to the following filter expressions B:

- Every allowed XPath expression is an allowed filter expression. For example, when A is an attribute and E is an element, then @A and ../E are allowed filter expressions used to check the existence of the attribute A or the element ../E respectively.
- Every comparison is an allowed filter expression, where the comparison operands are
  1. constants as e.g. ‘ ”52” ’ or
  2. allowed XPath expressions

  and the comparison operators are ‘=’ or ‘!=’. [1]

[1] We discuss the inclusion of the operators ”<”, ”<=”, ”>”, ”>=” in section 4.6.

- If B1 and B2 are allowed filter expressions, then 'B1 and B2' , 'B1 or B2' and 'not (B1)' are allowed filter expressions.

The restriction for predicate filters is irrelevant for the discussion in chapter 3 but is later used in chapter 4. The definition of *allowed XPath expression* shows that position predicates like '[1]' and any function calls in the predicate like 'count( )', etc. are forbidden. This makes sense in the context of access control and synchronization, since we have a "living" document. E.g. if nodes are added to or deleted from the document, XPath expressions containing position predicates may have a different result set afterwards. The definition of *allowed XPath expression* also forbids to use the preceding(-sibling) and following(-sibling) axes because DTDs can not define an order on sibling elements and it is impossible to decide which sibling to select without physical access. Therefore, in the context of access control (and synchronization) we demand an access control list (and a list of locks respectively) to contain only allowed XPath expressions. Since, for an arbitrary XPath expression, it is always possible to compute an allowed XPath expression locking a superset of nodes in the XML document very fast [2], we consider our restricted set of allowed XPath expressions to be sufficient for a wide variety of applications. In the following discussion, we will focus on the allowed subset of XPath and use the terms *location paths*, *location step* and *filter predicates* referring allowed XPath expressions only.

3. Our approach to solve the problem

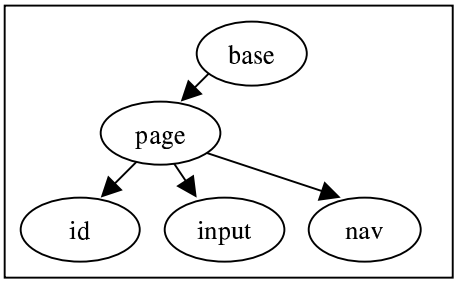
3.1. Access control and synchronization based on XPath expressions

Compared to a physical approach to access control (or locking), which operates on document nodes found in the currently processed XML document, our approach uses the DTD to evaluate XPath expressions given for access rights, queries, write operations, and granted locks. Our approach collects from the XPath expression the accessed qualified node names (*not* the node set) for elements or attributes that have to be checked (or locked) for a given query or update operation. Whenever an XML document contains many document nodes with the same (qualified) node name, and a query or write operation wants to access this node set with a single XPath expression, it may be an advantage to access control and synchronization, if only the XPath expression for this node set has to be checked or locked, instead of locking several nodes having the same node name in the currently given XML document. This is a minor reason for the use of qualified node names of XPath expressions (instead of node sets), besides the major reason, that our applications do no accept to use access rights as filters that modify a query result.

[2] e.g. by ignoring other *predicate filters* and *node type* tests and substituting the sibling axes by a superset fragment built on the child- and descendant-axe

3.2. The DTD and the corresponding DTD tree

```
<!ELEMENT base (page+)>
<!ELEMENT page (input*| nav?)>
<!ATTLIST page id ID #REQUIRED>
<!ELEMENT input (#PCDATA)>
<!ELEMENT nav (#PCDATA)>
```




```
<!ELEMENT base (page+)>
<!ELEMENT page (base*)>
```

A DTD defines a parent child relationship on elements and attributes in an XML document that can be represented as a *DTD tree*. For example, see the figures on the left. We use the defined relationship to compute possible location paths in the normalization algorithm outlined in the following sections.

For this purpose, the corresponding DTD must be as strict as possible. E.g. a definition like <!ELEMENT base ANY> will force the algorithm of section 3.4 to explore all defined elements and their possible children. A definition like this is allowed but slows down the performance. – Furthermore, we allow for recursive definitions in the DTD as in the third figure on the left, but set up a limit for the recursion depth, in order to terminate the normalization algorithm described in the following section. Such a limit of the recursion depth, say k=3, can be chosen e.g. from a given XML document. Then, we substitute the recursive definition in the DTD with a so called corresponding *DTD tree*, that contains the recursively defined nodes down to the chosen recursion depth.

3.3. Examples for the normalization of XPath expressions

Before we outline the normalization algorithm, let us consider an example. Assume, we have a DTD defined as follows.

```
<!ELEMENT base (page)>
<!ELEMENT page (input* | nav?)>
<!ELEMENT input (#PCDATA)>
<!ELEMENT nav (#PCDATA)>
<!ATTLIST page
    id ID #REQUIRED
    style (web | consol) #IMPLIED
    txtid IDREF #IMPLIED
>
<!ATTLIST input  style (headline | plaintext) #IMPLIED >
<!ATTLIST nav
    txtid IDREF #IMPLIED
    style (headline | plaintext | button) #IMPLIED
>
```

Note that the ‘input’ element as defined in the DTD has no attribute ‘txtid’. That is why the node test ‘input’ does not occur in the result of the second step (i.e. the expansion

step) in the following example.

XPath expression given by the application:
./base/page[@id='1']//*[@txtid]/@style

1. Anchor step: transform to absolute path:
/base/page[@id='1']//*[@txtid]/@style

Expand the location step using the desendant-or-self-axe according to the DTD:
/base/page[@id='1'] [@txtid]/@style |
/base/page[@id='1']/nav[@txtid]/@style

2. Transform *predicate filters* to relative filters (in this case nothing to do).

3. Right shuffle all predicate filters
/base/page/@style[../../@id='1'] [../@txtid] |
/base/page/nav/@style[../../@id='1'] [../@txtid]

A normalization example of an XPath expression that contains a *relative location path* using the *child-axe,* the *descendant-axe, node-tests, predicate-filters*, and at the end the *attribute axe*.

Furthermore, we give an example for an expansion step (step 2) that resolves a loop, i.e. a location step using the parent-axe.

Example 2:
/base/page[@id='1']/input[@style='plaintext']/../nav/@txtid

resolved loop:
/base/page[@id='1'][. /input[@style='plaintext']]/nav/@txtid

An example resolving a loop

3.5 Transforming allowed XPath expressions into normalized location paths

In order to obtain a uniform and fast processable representation of the allowed XPath expression (AXE), we normalize it to a set of normalized location paths (NLP), where the result set of AXE is equivalent to the result set of (NLP_1 | NLP_2 | … | NLP_n).
This is done in four steps:

1. **Anchor step1:** If the *allowed XPath expression* is a relative expression, transform it into an absolute expression. Note that relative location paths used by an application to access an XML fragment can always be appended to the location of the current node position in an XML document in order to compute the absolute location path for the XML fragment. (After this step the self-axe is only used in the *filter predicates*.)

2. **Simplify through exploring & expanding step:** This step reduces the set of used axe-specifiers [3] to the child, parent, namespace and attribute axe by exploring the associated DTD and creating a set of normalized location paths.
Therefore, expressions containing the axes: ancestor (-or-self) and descendant (-or-

[3] This step has to be made for all *location paths* in the *predicate filters* as well. In the case of predicate filters, the set of used axes includes the self axe.

self), have to be equivalently replaced by child- and parent-axe notations using the DTD tree (see 3.2.) for substitution. Furthermore, we have to resolve *loops*, that consist of location steps that use the parent-axe and have a previous (i.e. a left neighbored) location step that has a non-parent-axe in the location path (see example 2 in section 3.3).

In order to achieve the reduction of the axes, we explore the corresponding DTD and expand the result set by the following algorithm.

> Formal notation: an **allowed XPath expression** (**AXE**) is a sequence of location steps, /<LocationStep$_1$>/…/<LocationStep$_i$>/…/<LocationStep$_N$> with
> <LocationStep$_i$> = AxisName$_i$::NodeTest$_i$ [Predicate$_i$]

resultSet := {AXE}
for (i=1 to n) // i.e. for every location step 'i' found in AXE
{ **for** each AXE$_k$ ∈ resultSet
// check whether or not LocationStep$_i$ in AXE$_k$ requires to substitute AXE$_k$
if AxisName$_i$ ∈ { ancestor, ancestor-or-self, descendant, descendant-or-self }
{ remove AXE$_k$ from the result set and add the following expressions
for (each node on the axe AxisName$_i$ starting from any possible content node in the DTD tree[4] defined by /…/<LocationStep$_{i-1}$>) **//explore**
{ **//expand**
create an expression by replacing <LocationStep$_i$> with NodeName[predicate$_i$] where NodeName is the name of any node fitting the NodeTest$_i$ corresponding to the associated DTD.
(Note that this includes the exploration of the wildcard *)
} }
else if (AxisName$_i$ = parent and axisName$_{i-1}$!= parent) **//resolve loops**
replace the expression <LocationStep$_{i-2}$>/<LocationStep$_{i-1}$>/<LocationStep$_i$>
with AxisName$_{i-2}$::NodeTest$_{i-2}$ [Predicate$_{i-2}$] [./NodeTest$_{i-1}$ [Predicate$_{i-1}$]]
[./NodeTest$_i$ [Predicate$_i$]]
}

3. **Anchor step2:** Transform all absolute expressions in *predicate filters* into relative expressions. Note that an absolute location path can be rearranged to a relative location path based on any context node using the absolute path to the context node.

4. **Right shuffle of all predicate filters:** This step shuffles the predicate filter of each location step in the location path into the right most node of the expression. We call the right most node in each expression the **accessed node** (**AN**).
Each *predicate filter* not belonging to (i.e. not being in the same location step as) an *accessed node* has to be right shuffled until it is part of the *accessed node*. This is done by adding parent axes steps to the location path of each filter comparison, not being a constant, for each right shuffle.

[4] This loop terminates, because the DTD tree depths is restricted

Finally, all predicate filters [P1], ..., [Pn] belonging to the accessed node are combined into a single predicate filter [P1 and ... and Pn]. This is possible, because we restricted predicate filters to Boolean expressions and none of the filters depends on order (neither document order nor filter application order). Whenever, there is no predicate filter for an accessed node, we add a predicate filter [true( )] to that accessed node.

To summarize, normalization transforms an arbitrary allowed XPath expression into an equivalent set of normalized location paths, that computes the same node set for an arbitrary given XML document.
Notice that a normalized location path is absolute, built only along the child-axe and contains no wildcards ("*") as node test. Furthermore, all restrictions of the accessed node through predicate filters are defined in the location step of the accessed node itself and the previous location steps of the location path do not contain any predicate filter any more. Finally, the predicates use only parent-child relationships containing no loops.
This will be the input for the predicate evaluator discussed in section 4.

3.6. Read access for the expressions used in predicate filters

Which XML fragment is written by an update operation, does not only depend on those nodes for which write access is granted. It depends also on the values read in predicate filters, as the following example shows. If a write lock (or write access) is granted to

/base/page/input[../nav/@style="web"]

this implies a read lock (or read access) to

/base/page/nav/@style

because a change of this value during the write access could maliciously modify the write access right (or lock). Therefore, the write access operation (or write lock) implicitly read accesses (or has to read lock) all nodes, that are accessed by any predicate filter. This can be done by applying the normalization technique to every XPath expression used in a predicate filter as well. The same holds for read locks which use XPath expressions in their filters.
Although we distinguish between read and write access, and use different XPath expressions for read and write access rights, we do not explicitly mention these different access modes in following in order to keep the discussion simple.

3.7. Access control and locking on the basis of corresponding accessed node paths

Within the remainder of section 3, we ignore the predicate filters in the last location step, and we call a normalized location path ignoring the filters of the last location step the **access node path** (**AN-Path**). It identifies a set of siblings in the XML document.
We now can express access rights, queries, updates and locks in terms of (sets of) normalized XPath expressions, that characterize accessed (or locked) parts of an XML document. We say, two (or more) normalized XPath expressions **correspond**, if they have the same AN-Path.
For the remainder of this subsection, let us restrict normalized XPath expressions to AN-

Paths, i.e. ignore predicates. Then, in order to check whether locks conflict or an access right covers an access operation, we only have to check whether or not their AN-Paths correspond.

3.7.1. Access control

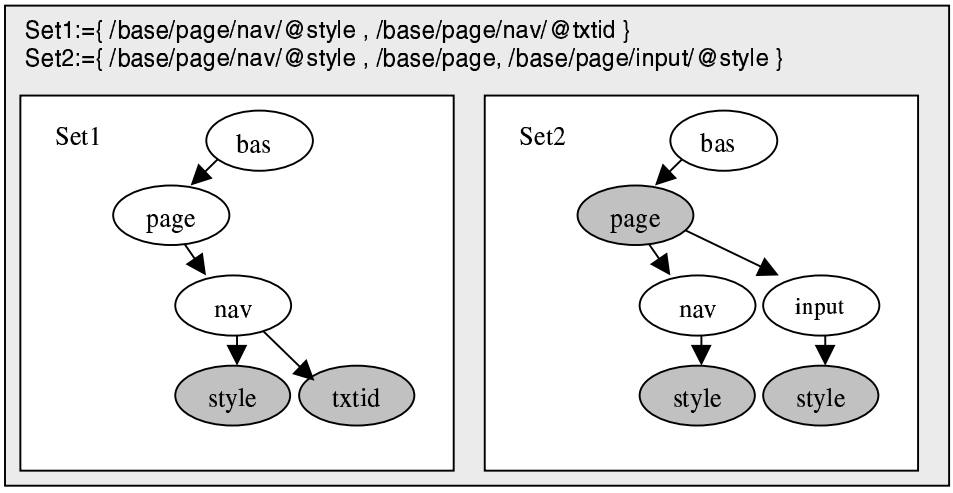
Given access lists containing normalized XPath expressions for both, the read and the write access rights and a query (or update) operation, access control checks whether or not the operation's access list contains a subset of the access right's access list.
For this purpose, we have to check whether or not for every AN-Path of the operation, there is a corresponding AN-Path of the access right. Whenever the operation contains an AN-Path (i.e. requests to access a node), for which no corresponding AN-Path is found in the access rights (i.e. no access right exists), then the access control returns an access violation exception, otherwise access is granted.

3.7.2. Synchronization on the basis of run time access expressions

The document fragment accessed at run time may be considerably smaller than that fragment for which an access right was granted. Therefore, a user program may perform a lock request at run time containing an actually used XPath access expression. Each lock request has to be checked against the locks granted to concurrent user programs.
Concurrent operations do not conflict, if none of the accessed nodes of one operation corresponds to an accessed node of the other operation.

3.7.3. How to compute corresponding accessed node paths (AN-Paths)

Given two sets of AN-Paths (set1 and set2) that describe accessed (or locked) node names, we can compute for a set1 all corresponding AN-Paths of set2 as follows.
For each set, we use a tree for the representation of the result set of the normalization, such that the inner nodes of the tree combine AN paths with common prefixes, i.e. we map each set of identical prefix nodes of different AN paths to a single node of a tree.



This has to be done once per XPath expression and may take a worst time complexity of O(n log n), with n being the number of nodes in the tree.
Now the problem of computing corresponding AN paths is to map two trees, which requires to visit each node of both trees at most once. Therefore, a worst case time complexity of O(n) is sufficient in order to compute for set1 that subset of set2, that represents the set of corresponding nodes, and to identify whether this subset of set2 is empty (i.e. we have no overlap and therefore no lock conflict) or equal to set1 (i.e. set1 is a subset of set2). Note that the size of each set is limited by the size of the DTD tree, which is limited by a finite recursion depth (as described in section 3.2).

4. Predicate tests for corresponding nodes in XPath expressions

Up to now, we did not consider predicates, i.e. we only considered whether or not an AN-Path is accessed (or locked or access is granted for it). In order to allow for finer grained access rights (and to allow for increased user program parallelism), we can include predicates in our XPath expressions as defined in section 2.2..

4.1. Access control using predicates of corresponding AN-Paths

Access control is performed on given normalized XPath expressions for an operation (query or write operation) and for the access rights of the user. Access is granted, if and only if for each normalized XPath expression of the operation, say N_{OP}, there is a corresponding normalized XPath expression in the access right, N_{AR}, with compatible mode (read or write) such that the predicate filter [X1] used by N_{OP} selects a subset of the node set that is selected by the predicate filter [X2] used by N_{AR} . If access can not be granted, an access violation exception is thrown.
Whenever, the predicate filter [X2] is [true( )] , i.e. there is no access restriction for that AN-Path, then we can grant access and we do not need to perform a subset test on the predicate filters.
In order to give a more general example, let a given operation of a user program use a normalized XPath expression with a predicate filter

[X1] = [@a1="6"]

i.e. it selects those accessed nodes for which the attribute 'a1' exists and has a value of "6", and let the corresponding normalized XPath expression for the access right contain a predicate filter, that only requires the existence of attribute 'a1' in the accessed nodes

[X2] = [@a1]

then

$X1 \Rightarrow X2$

holds, and therefore, the operation accesses only a subset of that fragment for which the access right is granted. Since the formula ($X1 \Rightarrow X2$) is equivalent to

'the formula (X1 and not (X2)) is insatisfiable' ,

we can use the same predicate evaluator as used for the overlap test.

4.2. An overlap test for predicates occurring in corresponding AN-Paths

Concurrency control is performed on given normalized XPath expressions for a new (read or write) lock request and for a (read or write) lock already granted to a different user. The new lock request is granted, if the Xpathe expression of the lock request does not overlap with the XPath expression of the lock already granted. Given two normalized XPath expressions, N1 and N2, the overlap test uses the predicate filters [X1] of N1 and [X2] of N2 as input arguments. The overlap test returns true, if '(X1 and X2)' is satisfiable.

For example, assume that three conflicting operations of concurrent programs want to access the same AN-Path of an XML document. Let [X1], [X2] and [X3] be the predicate filters that are used for that AN-Path by the three operations, say [X1] is

[@a1="5" and ./e1[@a2="6"] and @a3="7"]

and [X2] is

[@a1="5" and ./e2 [@a2="6"]]

and [X3] is

[not(./e2) and @a3="8"] .

Both, [X1] and [X2] use the same selection [@a1="5"] in the attribute 'a1' of the current node, and they use restrictions on different successor nodes 'e1' and 'e2' respectively, i.e. the node sets filtered by these predicate filters may overlap.

However, [X1] and [X3] select node sets with different values for the attribute 'a3', i.e. the node sets filtered by [X1] and [X3] do not overlap.

Finally, the node sets filtered by [X2] and [X3] do not overlap, because [X3] selects only those elements, for which a child 'e2' does not exist and [X2] selects those elements which have a child 'e2' that additionally has an attribute 'a2' with a value of "6".

In general, for the test, whether or not two predicate filters [X1] and [X2] occurring in corresponding AN-Paths of normalized XPath expression may address overlapping XML fragments, our predicate evaluator described below checks whether or not the expression '(X1 and X2)' is satisfiable.

4.3. Predicate filters containing elements and lock generalization

Whenever the predicate filters contain element paths with an element, say 'e', then it makes a difference for the overlap test, whether the DTD allows for a given context at most one occurrence of a child element 'e' or multiple child elements 'e'. When the DTD allows for a given context at most one child element 'e', then a predicate filter [./e[@a2="6"]] must select a node set that is disjoint from the node set selected by the predicate filter [./e[@a2="8"]]. However, when the DTD allows for multiple occurrences of a child element 'e' for a given context, then for a given context node there may be one child element 'e' with a value "6" for the attribute 'a2' and another child element 'e' with a value "8" for the attribute 'a2'. Therefore, a predicate filter [./e[@a2="6"]] , which requires only the existence of a child 'e' with an attribute 'a2' having a value of "6", may overlap with a predicate filter [./e[@a2="8"]] , which requires the existence of a (possibly different) child 'e' with an attribute 'a2' having a value of "8".

In the case that the DTD for the given context restricts each node of the selected node set to have at most one child element with a given name, say 'e', the test

'./e[@a2="6"] and ./e[@a2="8"]'

is forwarded to our predicate evaluator[5], whereas in the second case, this conjunction is satisfiable. We treat the second case, by using a filter for a superset, i.e. if the DTD allows for multiple child elements 'e' for a single context node, then we use a more general condition, in this case './e[@a2]' as filter for the lock expression, i.e. in order to treat this case correctly, we lock a superset of what was originally requested.

4.4. The predicate evaluator

In order to check whether or not a formula generated from the overlap test or from the subset test is satisfiable or not, we transform the formula into disjunctive normal form (DNF) by applying the following equivalence transformations:
Let X, Y and Z be Boolean expressions, A and B be comparison operands, and <path> be a location path relative to the current node and <A> and <B> AN-Paths:

Negations are moved inside the formula as far as possible, i.e.
" not (X and Y) " is substituted with " not (X) or not (Y) " and
" not (X or Y) " is substituted with " not (X) and not (Y) " and
" not (not (X)) " is substituted with " X " and
" not (<path>[X]) " is substituted with " not (<path>) or <path>[not (X)] " and
" not (A = B) " is substituted with " not (<A>) or not (<B>) or A != B " [6] and
" not (A != B) " is substituted with " not (<A>) or not (<B>) or A = B " [6] .

Disjunctions are moved outside of conjunctions and location paths, i.e.
" (X or Y) and Z " is substituted with " X and Z or Y and Z " and
" <path>[X or Y] is substituted with " <path>[X] or <path>[Y] " .
Conjunctions are moved outside of location paths, i.e.
" <path>[X and Y] is substituted with " <path>[X] and <path>[Y] " .

A formula in disjunctive normal form (DNF) is satisfiable, if and only if at least one conjunction of the formula in DNF is satisfiable. This is checked as follows.

4.5. The predicate evaluator for a single conjunction

In order to check, whether or not a single conjunction of conditions is satisfiable, we use the following algorithm: We introduce an equivalence class for each constant and for each AN-Path, where different AN-Paths are considered to be different, i.e. @a2 and ./e1/@a2 and ./e1 are considered to belong to three different equivalence classes.
For each comparison [left = right] occurring in the conjunction, we combine the equivalence class containing 'left' and the equivalence class containing 'right' into a

---

[5] The same is done for all attributes, because for every given context node, there is (at most) one occurrence of this attribute.
[6] Here, the condition not <B> can be omitted, if B is a constant.

single equivalence class containing the union of all attributes and constants found in the equivalence class containing 'left' and in the equivalence class containing 'right'.
Whenever, there is an equivalence class containing two different constants, say c1 and c2, this means that the set of conditions given for that location path is equivalent to 'c1=c2' which is insatisfiable.
Furthermore, we consider all conditions of the form [left != right] occurring in the conjunction. If for at least one condition of the form [left != right] we find out, that 'left' and 'right' belong to the same equivalence class (i.e. 'left'='right'), then this conjunction is insatisfiable.
Finally, we check for each occurrence of 'not(A)' in a conjunction whether the same conjunction contains also a comparison 'A = B' or a comparison 'A != B' or A alone (stating that A exists). Remember that after transformation into DNF the 'A' occurring in 'not(A)' is an AN-Path, and not(A) claims that the node set selected by this AN-Path in the currently given XML document is empty. Therefore, if the conjunction additionally contains a comparison 'A = B' or a comparison 'A != B' or 'A', then this conjunction is insatisfiable. Otherwise the conjunction is satisfiable.

4.6. The complexity of the predicate evaluator

Although transformation into disjunctive normal form requires exponential time in general, we usually have few disjunctions in the predicates of corresponding normalized XPath expressions, which means that this transformation usually is fast.
When m is the number of conditions in a conjunction, the predicate evaluator requires a complexity of $O(m^2)$ per conjunction to combine equivalence classes of AN-Paths, to check the conditions containing [A != B] and to check the conditions 'not(A)'. If the DNF contains c conjunctions, the time complexity is $O(c*m^2)$.
If we would also include one or more of the operands "<", "<=" , ">", ">=" , this would require a more complex predicate evaluator for conjunctions, as e.g. the predicate evaluator described in [3], that requires a complexity of $O(m^3)$ with m being the number of conditions in the conjunction.

5. Summary and Conclusions

We have presented a unique approach to access control and synchronization for XML documents, based on XPath expressions. Given XPath expressions for access rights, queries, write operations and locks, we describe how access control and synchronization can be implemented using the DTD and logic. The basic idea is to normalize XPath expressions such that corresponding normalized XPath expressions can be found very efficiently, and to reduce access control and locking to operations on corresponding access node paths (AN-Paths). A simple approach, that we used in section 3, performs access control and synchronization on AN-Paths alone and can be implemented efficiently, i.e. in a time O(n log n). An improved version, that allows for a higher degree of program parallelism and for finer grained access rights, includes the predicate filters used in XPath expressions. We outlined the tests which have to be done for access

control and for synchronization, and we have shown that both tests can be implemented by a single predicate evaluator, for which we presented one possible implementation that (except for the transformation to DNF) needs a time of $O(c*m^2)$ per test. Since our approach to access control and synchronization does not rely on the storage of XML documents in a specific database system, it may be an interesting challenge to combine it with new web services and middleware technology developed around XML.

References:

[1] Elisa Bertino , Silvana Castano , Elena Ferrari, On specifying security policies for web documents with an XML-based language, Proceedings of the Sixth ACM Symposium on Access control models and technologies, May 2001

[2] Elisa Bertino, Silvana Castano, Elena Ferrari, Marco Mesiti: Controlled Access and Dissemination of XML Documents. Workshop on Web Information and Data Management 1999: 22-27

[3] S. Böttcher, M.Jarke, J.W. Schmidt: Adaptive Predicate Managers in Database Systems, VLDB 1986.

[4] S. Böttcher, A. Türling: Transaction Synchronization for XML data in Client Server Web Applications, GI-Jahrestagung, Wien, 2001.

[5] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, Pierangela Samarati: XML Access Control Systems: A Component-Based Approach. DBSec 2000: 39-50

[6] Ernesto Damiani , Sabrina De , Stefano Paraboschi , Pierangela Samarati Securing XML Documents Volume 1777, Issue , pp 121- Lecture Notes in Computer Science

[7] Ernesto Damiani , Sabrina De Capitani di Vimercati , Stefano Paraboschi , Pierangela Samarati, Fine grained access control for SOAP E-services, The tenth international World Wide Web conference on World Wide Web April 2001

[8] S. De Capitani di Vimercati , S. Paraboschi , P. Samarati International Journal of Information Security. Securing SOAP e-services E. Damiani , Dipartimento di Tecnologie dell'Informazione, Universita di Milano, Via Bramante 65, 260 ... ISSN: 1615-5270

[9] Florescu, D., Levy, A., Mendelzon, A.: Database Techniques for the World Wide Web: A Survey. ACM SIGMOD Record, Vol. 27, No. 3, September, 1998

[10] Goldman, R., McHugh, J., Widom, J.: From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. Proc. of the 2nd Int. Workshop on the Web and Databases (WebDB), Philadelphia, June, 1999

[11] Kanne, C.-C., Moerkotte, G.: Efficient Storage of XML Data. Proc. Of the 16 th Int. Conf. On Data Engineering (ICDE), San Diego, March, 2000

[12] Michiharu Kudo , Satoshi Hada, XML document security based on provisional authorization, Proceedings of the 7th ACM conference on Computer and communications security, 2000

[13] Dieter Scheffner, Rainer Conrad: Access Support Tree & Text Array: A Model for Physical Storage of XML Documents. GI Jahrestagung (1) 2001: 406-416

[14] Schöning, H., Wäsch, J.: Tamino -- An Internet Database System. Proc. of the 7 th Int. Conf. on Extending Database Technology (EDBT), Springer, LNCS 1777, Konstanz, March, 2000

[15] Tatarinov, I., Ives, Z.G., Halevy, A.Y., Weld, D.S.: Updating XML, ACM SIGMOD Int. Conf. on Management of Data, 2001

[16] Yue Wang, Kian-Lee Tan: A Scalable XML Access Control System. WWW Posters 2001

[17] Zhang, X., Mitchell, G., Lee, W.C., Rundensteiner, E.A.: Clock: Synchronizing Internal Relational Storage with External XML Documents, RIDE-DM 2001.