Checking XPath Expressions for Synchronization, Access Control and Reuse of Query Results on Mobile Clients

Stefan Böttcher , Adelhard Türling
University of Paderborn
Faculty of Computer Science, Electrical Engineering and Mathematics
Fürstenallee 11 , D-33102 Paderborn , Germany
email : stb@uni-paderborn.de , Adelhard.Tuerling@uni-paderborn.de

Abstract
The evaluation of XPath expressions plays a central role in accessing XML documents and therefore may be used in XML database systems for different components. We demonstrate that different applications ranging from access control to transaction synchronization to the reuse of query results have very similar requirements to the evaluation of XPath expressions, which can be solved by the same two steps. Firstly, we compute from each XPath expression a regular expression of the selected node paths and right-shuffle predicate filters to the selected nodes. Secondly, we describe the treatment of predicate filters which may be used in XPath expressions for queries, access control, and synchronization, and present a fast predicate evaluator for these predicates. Finally, we introduce the concept of "fall-back decisions", which allow us to use an incomplete but efficient theorem prover, which solves most cases in practice and guarantees correct fall-back behavior for the other cases.

Keywords: XPath predicate evaluation, XPath overlap test, subset test.

1. Introduction

1.1 Motivation and problem description

XML plays a central role as data exchange format in client-server applications with mobile partners. Whenever XML data is stored in a database, then both, access control and synchronization of the clients' access to XML data are important topics [1],[5],[15]. Furthermore, when query results have to be shipped from a server-side XML database to mobile clients over small bandwidth connections, then reusing previous query results which are already stored at the client-side in order to answer a new query may considerably reduce the data transfer from server to client.
XPath [18] is a standard which is used as data access language for XML data and which additionally plays a central role in other XML standards like XSL(T) and query languages like XQuery. XPath expressions can be used to characterize fragments of an XML document which are read by a query or are modified, as well as fragments which have been loaded into a local memory by a previous query, and fragments to which an access right is granted. Our work concentrates on XPath expressions that are used to exchange, access, or control the access of data. We present two tests, a subset test and an overlap test for XPath expressions, both of which contribute to solve three different problems in an XML database system with mobile clients: access control, the reuse of

previous query results, and synchronization. Our approach works on the qualified node names of the DTD (*not* on the physical node set) for elements or attributes which must be checked (or locked) for a given query or update operation.
We follow approaches like [5,14] for the definition of access rights, i.e., we define which fragment of an XML document may be read and which fragment may be written. Access control has to check that no user attempts to access an XML fragment within an access mode (read or write) beyond his access rights granted in this access mode and thereby violates his access rights. Because we use XPath expressions in order to describe queries and access rights, the node set of any given XML document which can be accessed by an XPath query must be a subset of the node set which would be selected by an XPath expression which describes the access right exists. It is an essential requirement of our applications to the access control component that access violations are detected and reported, because users with different access rights must cooperate, but their individual decisions depend on their knowledge, i.e. on the data which they read. Therefore, our applications *can not accept* that different users get different answers to the same query. That is why we cannot implement an access right by using an additional filter which restricts the query results, as suggested in other contributions, e.g. [5,16].
For example, one XPath expression

```
    XP1 = / root / customer [ @lastname="Meier" ]
```

can be used to query for `customer` elements which are directly located under a `root` element and have an attribute `lastname` with the value `"Meier"`. Such an XPath expression can either be used directly by an XPath query engine or indirectly by a component like XQuery or XSLT which use XPath in order to query for certain subsets of an XML document. Similarly, an *access right* for a user can be defined by another XPath expression

```
    XP2 = / root / customer
```

which allows the user to access all `customer` elements under the `root` element. Our subset tester proves without access to the database that the query `XP1` selects a subset of the nodes described by the access right `XP2` in all possible database states, i.e., an access which uses the query `XP1` can be granted to a user with the access right `XP2`.
The reuse of query results is interesting in an application environment where mobile clients with small bandwidth connections query XML data which is provided by a server-side database. In order to be sure that a result of a previous query which is stored at the client-side can be reused, the client can prove that a new query asks for only a subset of the nodes that are stored as a previous query result. Reusing the previous query result to answer the new query, will significantly save communication costs between the server and the client compared to submitting the new query to the server and transferring the results back to the client. For example, let us assume that the query result of XP1 is still available in the main memory of a mobile device and a new query

```
    XP3 = /root/customer[ @firstname="Tom" and @lastname="Meier" ]
```

is submitted by the client application (say, because this data is needed in an XSLT stylesheet), then again our subset tester is used. However, this time it proves that `XP3` selects a subset of `XP1`, i.e. it concludes, that the previous query result of `XP1` can be reused. That is, instead of submitting a query for `XP3` to the server and sending the answer back to the mobile client, the previous query result of `XP1` can be used to compute the answer to `XP3` without data exchange between client and server.

Finally, transaction synchronization has to prevent two concurrent transactions from accessing the same fragment of an XML database concurrently with at least one of them requiring an exclusive access right. For example, let us assume that a concurrent transaction wants to have an exclusive access to all data which are described by an XPath expression

```
    XP4 = / root / * [ @destination="Berlin" ]
```

and the DTD of our XML document allows to substitute `customer` for the node test `*` in `XP4`, then our overlap tester finds out that the XPath expressions XP3 and XP4 overlap, i.e., there is a possible database state in which both XPath expressions select the same set of data (in this case customers "Tom Meyer" who want to fly to the destination "Berlin"). In such a situation the scheduler at the server-side decides that both transactions have to run serial, because they may access overlapping data.

As mentioned in the previous example, we need the DTD information in order to expand * correctly, i.e., to find out that we can choose the element `customer` for *.

To summarize: The input of our tester is a DTD and two XPath expressions (XP1 and XP2) which are used for queries, access rights, locks, or stored previous query results. Our goal is to prove without any physical access to the actual state of the XML database that XP1 and XP2 select disjointed node sets or that the node set selected by XP1 is a subset of the node set selected by XP2.

1.2. Our focus in comparison to related work

Other contributions have investigated the reuse of previous query results, transaction synchronization and the control of access rights independently of each other, whereas our predicative view of the problems allows us to use the same approach to solving key aspects of all three problems for XML database systems.

Within the area of access control on XML databases there have been contributions to a wide variety of different aspects ranging from policies to user groups to document location in the web, to access control for fragments of XML documents [10,2,6,1]. Within the approaches to access control on fragments of XML documents, there is a trend towards fine-grained access rights, which include and go beyond access rights at the level of attributes or elements [16,5]. We follow and extend this direction and define access rights not only on the level of nodes, but also include fine-grained access rights specified by XPath expressions with predicate filters. In comparison to other approaches which use an additional filter that operates on physically given data in order to avoid access violations, our application environment requires our approach to give complete answers to all users and to reject each access that violates access rights.

In comparison to approaches which delegate transaction synchronization to a database system that is used to store, query and modify the information contained in XML documents [8], our approach operates on the XPath formulas alone and does not rely on the existence of a database system. Therefore, it can be used as an add-on to an existing XML database system, but it can also be used to control concurrent access to large XML documents which are stored in a file system.

Within the area of reusing previous query results, we follow approaches for semantic caching, which was introduced in [13] in contrast to tuple-based and page-based caching. In contrast to the first contributions to semantic caching, we follow [7, 11, 12, 17] and decide on basis of the XPath expressions themselves and based on the DTD whether or not a new XPath query can reuse a previous XPath query result. Previous contributions [7, 11, 12, 17] report on decidability results or give upper and lower bounds for the complexity of the containment test for certain subclasses of XPath expressions. However, in contrast to this, we focus on a fast decision as to whether or not an XPath query result can be reused and we allow our containment test to be incomplete.

Finally, our approach integrates ideas from predicate logic and provides a uniform predicate tester for access control, the reuse of query results, and synchronization. In contrast to contributions from the database area, we use a subset of XPath in order to access, lock and query document fragments. The work presented here is based on our previous work [3], however it adds the following contributions. We consider an additional problem, i.e. the ability to reuse previous query results for the evaluation of new XPath queries. We introduce a concept, called "fall-back behavior", which guarantees correct application behavior, and thereby allows us to use a fast but incomplete predicate tester. That is, depending on the application, we treat incomplete answers of our predicate tester in such a way that they do not harm access control, the reuse of query results, or transaction synchronization. Furthermore, we consider a more general DTD definition, i.e. we allow recursive definitions in DTDs, which leads us to a completely different evaluation strategy for XPath expressions. Finally, we allow a larger set of predicate filters and present an extended theorem prover.

2. Normalization of XPath expressions

2.1. An overview of the complete approach

Within Section 2, we define the considered subset of XPath expressions and we show how to transform each of these XPath expressions into an equivalent set of *normalized XPath expressions*. Each normalized XPath expression selects only nodes with the same qualified node name, whereas the union of all normalized XPath expressions represents the same node set as the originally given XPath expression does. Each normalized XPath expression consists of two parts, a node path expression which describes the paths from the root node to the normalized XPath expression's selected node, and a predicate filter attached to the selected node name. This allows us (in Section 3) to reduce subset tests and overlap tests for arbitrary XPath expression to subset tests and overlap tests on normalized XPath expressions so that they can be performed individually for each identified node. Because most applications prefer a tester to be rather fast than complete, we have developed an incomplete tester, and we introduce (within Section 4) the concept of fall-back decisions to handle the tester's incompleteness.

2.2. The considered subset of allowed XPath expressions and DTD definitions

In general, an XPath expression is defined as being a sequence of location steps `/<LocationStep`$_1$`>/…/<LocationStep`$_N$`>`, where `<LocationStep`$_I$`>` is defined as `axis-specifier`$_I$`::node-test`$_I$`[predicate filter`$_I$`]`.

As the expressiveness of XPath is richer than required by our applications and may lead to complex test conditions, we restrict the considered XPath expressions for queries, previous query results, access rights, write operations and locks to a subset called **allowed XPath expressions** by the following rules (most of which are taken from [3]) for predicate filters, node tests and axes:

1. Allowed predicate filters

Predicate filters [F] of allowed XPath expressions may only contain the following filter expressions F:

- Every allowed XPath expression which does not contain a filter expression itself 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 .
- Every comparison is an allowed filter expression, where the comparison operands are
  1. constants as e.g. ‘ ”77” ’ or
  2. allowed XPath expressions which do not contain a filter expression themselves

  and the comparison operators are ‘=’ or ‘!=’.

If F1 and F2 are allowed filter expressions, then ‘F1 and F2’ , ‘F1 or F2’ and ‘not (F1)’ are allowed filter expressions.

2. Allowed node tests

All *node name tests* including the wildcard ‘*’ and name-spaces are allowed. However, we forbid *node type tests* like node(), processing-instruction(), and comment(). For example, when A is an attribute and E is an element, then @A , ./E , ../E, and //*/E are allowed XPath expressions.

3. Allowed axis-specifiers

- Our tester allows *absolute* or *relative location paths* with the following *axis specifiers* in their *location steps*: self, parent, ancestor, ancestor-or-self, child, descendant, descendant-or-self, namespace and attribute, but we forbid the following (-sibling) and preceding (-sibling) axes.

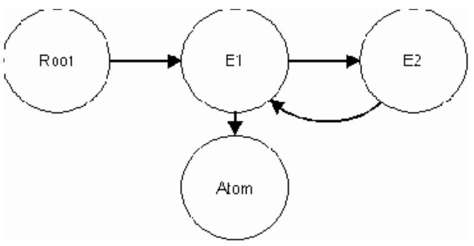
This restriction of XPath to *allowed XPath expression*, which excludes position predicates like ‘[2]’ and the use of the sibling axes, appears to be appropriate to data centric XML documents for the following reason. When nodes are added to or deleted from the document, XPath expressions containing position predicates or any of the sibling axes may have a different result set afterwards. Within the following sections, we consider only the allowed subset of XPath, i.e., when we use the terms *location path, location step* and *filter predicates*, we refer to allowed XPath expressions only.

2.3. A preparation step: constructing a DTD graph

Before the normalization of XPath expressions starts, the DTD is arranged in a so called directed *DTD graph* which contains all the elements and attributes of the DTD as nodes. The *DTD graph* contains a node for each element or attribute defined in the DTD and contains a directed edge for each parent-child relation or attribute-axis relation found in the DTD. Whenever the DTD is recursive, the DTD graph contains a cycle. For example, let the DTD contain the following element definitions:

```
<!ELEMENT Root ( E1? ) >
<!ELEMENT E1 ( E2 | Atom )>
<!ELEMENT E2 (E1* )>
```

```
  <!ELEMENT Atom (#PCDATA)>
```
The corresponding DTD graph is:



The DTD graph is used to compute so called node paths when a location step is expanded. A *node path* is a sequence of element names[1], which starts at the root (or at a node reached by a previous location step) and follows the child-axis or attribute-axis to the element of interest (i.e. to the element which is reached by a location step). Each node reached with the last expansion step is called *selected node of its (normalized) XPath expression*.

The DTD graph describes a superset of the paths allowed by the DTD. Missing information can be added to the DTD graph by attaching additional filters to the DTD graph nodes [4]. For example, a filter [./E2 xor ./Atom] attached to the node E1 expresses that each node E1 has either a child E2 or a child Atom. Similarly, a filter [unique(./E2)] attached to the node E1 expresses the constraint that for each element E1 there exists at most one child element E2. Such filters can be used to improve the completeness of a tester. However when the applications require or accept a fast but incomplete tester (as outlined in Section 4), it is also correct to ignore the filters for the following reason. If the intersection of the path sets selected by two XPath queries (XP1 and XP2) is empty under the superset of paths described by the DTD graph, we are also sure that this intersection is empty under subset of paths for XP1 and XP2 allowed under the given DTD, because the given DTD is at least as restrictive as the DTD graph. The same holds for the subset test, i.e., if an XPath query XP1 selects a subset of the paths selected by another query XP2 within the superset of paths described by the DTD graph, we can be sure that XP1 also selects a subset of XP2 under the more restrictive DTD.

In general we accept well-formed DTDs. We exclude the element type ANY for efficiency reasons, because ANY would result in a lot of cycles in the DTD graph and thereby in more complex further computations.

2.4 The first normalization step: computing node path expressions

The goal of the first normalization step is to perform equivalence transformations on a given XPath expressions such that the only remaining element location steps follow the child-axis (and the attribute-axis respectively), i.e. that each selected node can be described by a node path from the root node to this node. We compute a node path expression which describes these node paths, i.e. the node path expression describes the

[1] The last node in a node path may also be an attribute, however, we avoid mentioning this explicitly in what follows in order to keep the presentation simple.

set of all node paths to a node selected by the given XPath expression. This transformation into a node path expression is applied to the XPath expression location step by location step and uses the given DTD graph. Predicate filters are ignored within this first normalization step and are considered in the second step.
In order to give an extended example of how the first normalization step computes a node path expression from a given XPath expression, we use the DTD graph of the previous section and the following XPath expression

```
        (XP1)       //E1//E2/../*//E1    .
```

The first location step, `//E1`, includes arbitrary long node paths from the root to any element `E1` (i.e. `/Root/E1` , `/Root/E1/E2/E1` , etc.). That is why we use a node path expression

$$\texttt{/Root(/E1/E2)}^{\texttt{N}}\texttt{/E1} \quad , \texttt{N} \geq \texttt{0}$$

in order to describe all node paths from the root to the element `E1`. This node path expression contains a *node path loop* , i.e. $\texttt{(/E1/E2)}^{\texttt{N}}$ (N≥0), which summarizes all node paths containing zero or more sequences of `/E1/E2`. Detecting loops is part of this normalization step, and avoids us finding an infinite number of node paths to the node name `E1`.
The second location step in `XP1` , `//E2` , can be described by an additional node path expression for node paths form the previously selected node names (here only `E1`) to the currently selected node names (here `E2`) , in this case by the node path expression $\texttt{/E2(/E1/E2)}^{\texttt{N2}}$ (N2≥0). Thereafter, the node path expression which describes all node paths for the first two location steps of the given XPath expression `XP1`, i.e. for the XPath expression `//E1//E2` , can be computed by appending the node path expression computed for the second location step (from `E1` to `E2`) to the previously computed node path expression which describes node paths from the root to `E1`, i.e., the result is

$$\texttt{/Root(/E1/E2)}^{\texttt{N}}\texttt{/E1/E2(/E1/E2)}^{\texttt{N2}} \ (\texttt{N} \geq 0,\ \texttt{N2} \geq 0)\ .$$

This can be simplified to

$$\texttt{/Root(/E1/E2)}^{\texttt{N}} \ (\texttt{N} \geq 1)$$

where $\texttt{(/E1/E2)}^{\texttt{N}}$ (N≥1) describes a loop of one or more sequences of `/E1/E2` .
As the third location step of `XP1`, i.e. the parent-axis location step `/..`, requires removing the last node from each computed node path, the node path expression for the first three location steps is

$$\texttt{/Root(/E1/E2)}^{\texttt{N}}\texttt{/E1[./E2]} \ (\texttt{N} \geq 0)$$ [2].

The next location step of `XP1` , `/*` , is applied to the node name `E1` and selects its two successor node names, `Atom` and `E2`. For each successor node name, a node path expression is computed by appending the node name to the previously computed node path expression, i.e., we get the node path expression

$$\texttt{/Root(/E1/E2)}^{\texttt{N}}\texttt{/E1[./E2]/Atom} \ (\texttt{N} \geq 0),$$

which describes all node paths of `//E1//E2/../*` to `Atom` nodes, and similarly, we get the node path expression

$$\texttt{/Root(/E1/E2)}^{\texttt{N}}\texttt{/E1[./E2]/E2} \ (\texttt{N} \geq 0)$$

which describes all node paths to `E2` nodes. The latter node path expression can be

[2] The predicate filter [./E2] states that only elements E1 which have a successor E2 are selected.

simplified to

    /Root(/E1/E2)^N/E1/E2 (N≥0) .

The XPath expression XP1 requires to apply the last location step `//E1` to both of the previously selected node names, `Atom` and `E2`, but an application of this location step to `Atom` is prohibited by the DTD graph, because `Atom` has no successor nodes. Therefore, the last location step can only be applied to the node name `E2`. All node paths from `E2` to `E1` are described by the node path expression (/E1/E2)^{N2}/E1 (N2≥0) . This node path expression of the node paths from E2 to E1 is appended to the previously computed node path expression, in order to get the final result

    /Root(/E1/E2)^N/E1/E2(/E1/E2)^{N2}/E1 (N≥0, N2≥0) ,

which can be simplified to

    /Root(/E1/E2)^N/E1 (N≥1) .

This finally computed node path expression describes all node paths from the root node to those nodes which are selected by the XPath expression XP1.

2.5. General expansion of an XPath expression using the DTD graph

Let us generalize the example given. The node path expression which corresponds to an XPath expression is computed location step by location step by evaluating the DTD graph, where location steps in successor direction (child-axis, descendent-axis, and descendent-or-self-axis) are treated differently from location steps in ancestor direction (parent-axis, ancestor-axis, and ancestor-or-self-axis). Each location step is applied to each node in a set of selected nodes, where the first location step is applied to the root node only.

Whenever the location step follows the successor direction (child-axis, descendent-axis, or descendent-or-self-axis), then for each (previously) selected node[3], a node path expression which describes the node paths from the previously selected node to the node selected by this location step is computed. This node path expression is appended to that node path expression which describes all node paths from the root node to the previously selected node. After the expansion of this location step, we again have a set of selected nodes and for each node name a node path expression which describes all node paths to that selected node.

Whenever the location step follows the ancestor direction (parent-axis, ancestor-axis, or ancestor-or-self-axis), then for each selected node and the given node path expression which describes the node paths to that selected node, the location step in ancestor direction selects one or more nodes on the node path from the root to the currently selected node name. That is why a location step in ancestor direction splits the given node path into a prefix which becomes the new node path to the selected node and a suffix which is used as a predicate filter.

For example, let XP2 be the result of appending a location step `/ancestor::E1` to the XPath expression `XP1` of the previous subsection, i.e.,

    XP2 = //E1//E2/../*//E1/ancestor::E1 .

[3] In the above example, the last location step `//E1` has to be considered for two previously selected node names, `Atom` and `E2`, because both were selected by the previous location step, `/*`.

After the location steps `//E1//E2/../*//E1` have been evaluated as before, which has given us the node path expression `/Root(/E1/E2)`^{N}`/E1 (N≥1)`, the location step `/ancestor::E1` requires going back to any occurrence of `E1` in the node path except the last one. Hence, for `XP2`, we get the node path expression

`/Root(/E1/E2)`^{N1}`/E1 [ .(/E2/E1)`^{N2}` ]  (N1≥0) (N2≥1).`

Here,

`/Root(/E1/E2)`^{N1}`/E1 (N≥0)`

is the prefix which describes all node paths from the root node to selected nodes `E1` and

`[ .(/E2/E1)`^{N2}` ] (N2≥1)`

is a filter expression which is used to further restrict the selected nodes.
To summarize, a location step in ancestor direction applied to a given node path expression selects one or more nodes. For each selected node, the location step splits the given node path expression into parts: the first identifies the selected node and the second is used as a predicate filter which has to be applied to the selected node.
In order to simplify of the following presentation, we let *normalized XPath expressions* use a syntactic extension of XPath expressions which allows node path expressions for node paths to occur everywhere, where XPath allows an element to occur, and which also allows loops of child-axis location steps and loops of parent-axis location steps to occur in predicate filters.

2.6. The second normalization step: shuffling predicate filters to the right

While predicate filters have been ignored in the previous computations of node path expressions, within the following second normalization step, predicate filters are shuffled to the right-most node names in the node paths, i.e., to the selected node names. For example,

`/Root/E1[./@a=”5”]/E2`

is transformed into

`/Root/E1/E2[../@a=”5”] .`

In general, each *predicate filter* which does not belong to the last location step is right-shuffled along the child-axis (or attribute-axis) until it becomes a filter for the *selected node name*. This is done by adding parent-axis steps to the location path of each filter comparison which is not a constant for each right-shuffle.
Similarly, when a predicate filter is right shuffled along a node path loop to descendent nodes, this adds a corresponding node path loop along the ancestor-axis to the predicate filter. For example, let [FE] be a predicate filter then

`/Root/E1 [FE]/(/E2/E1)`^{N}`   (N≥0)`

is transformed into

`/Root/E1/(/E2/E1)`^{N}` [(../../)`^{N}` FE]   (N≥0) .`

Here, the *parent-axis loop* `(../../)`^{N}` (N≥0)` in the predicate filter `[(../../)`^{N}` FE]` is a shortcut notation for an even positive number of location-steps along the parent-axis.

Finally, all predicate filters [P1], ..., [Pn] which belong to the selected 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 a selected node, we add a predicate filter [true( )] to that selected node.
To summarize, our normalization uses the DTD graph to transform an arbitrarily allowed XPath expression into an equivalent set of normalized XPath expressions each of which is of the form

```
        rex [predicate filter] ,
```

where `rex` is a regular node path expression which describes all node paths of the given XPath expression to one selected node and `predicate filter` is a predicate filter expression which may or may not contain node path expressions and parent-axis loops. Notice that `rex` is a regular expression which is built only along the child-axis and which contains no wildcards (”*”) as a node test, but may contain node path loops. Furthermore, a normalized XPath expression defines all the restrictions of the selected node through predicate filters in the location step of the selected node itself, and the previous location steps of the location path do not contain any predicate filter any more. This will be the input for the predicate evaluator discussed in Section 3.

3. Problem reduction to predicate tests on XPath expressions

3.1. Reducing the locking of fragments to an overlap test for node path expressions

A new lock request can be granted, if the XML fragment to be locked does not overlap with any other locked XML fragment for which a lock in an incompatible lock mode has already been granted to a different transaction. In order to check this, the overlap test is applied to each node of the DTD graph which is selected by both lock requests (or by both XPath expressions associated with the lock requests respectively). When given two normalized XPath expressions, `rex1[X1]` and `rex2[X2]` which select the same DTD graph node, the overlap test returns false (i.e. that the normalized XPath expressions are disjointed), if `rex1` and `rex2` select no common node path or `‘( X1 and X2 )’` is unsatisfiable.
A tester for regular expressions [9] checks whether or not `rex1` and `rex2` select a common node path, and our tester for filter predicates (as described in Sections 3.3 to 3.6) checks whether or not `‘(X1 and X2)’` is satisfiable.

3.2. Reducing access control and the reuse of query results to subset tests on node path expressions

Access is granted (or an old query result can be reused respectively), if an XPath expression `XP1` which is being used for the current operation selects a subset of the nodes which are selected by an XPath expression `XP2` which is given for an access right (or the previous query result respectively) – independently of the current content of an XML document. The subset test is performed on the normalized XPath expressions computed from `XP1` and `XP2`. Our subset test is successful, if for each DTD graph node which is selected by a normalized XPath expression

```
        rex1 [ filter1 ]
```

which is computed from XP1, a normalized XPath expression

```
       rex2 [ filter2 ]
```

which is computed from XP2 exists, such that the following holds:

1. The element paths described by the regular expression rex1 are a subset of the element paths described by the regular expression rex2.
2. filter1 $\Rightarrow$ filter2 .

A subset test for regular expressions as needed in order to check the first condition is described in [9].The second condition is equivalent to:

(filter1 and (not filter2)) is unsatisfiable,

i.e., we can use the same predicate evaluator as we can use for the overlap test. Access is granted (or an old query result can be reused), if and only if this formula is unsatisfiable for each DTD graph node. If access cannot be granted, an access violation exception is thrown. If it is not possible to reuse an old query result, the new query has to be submitted to the server and the results have to be shipped back to the client.

3.3. Filters with elements for which the DTD allows multiple occurrences

The tester for filter expressions has to distinguish two cases for each element (say 'E2') which is a child element of another element (say 'E1'). If the DTD allows at most one child element E2 for a given element E1, then the expression './E2/@a="3" and ./E2/@a="4"' is unsatisfiable. Therefore, a predicate filter [./E2/@a="3"] used in the context of elements E1 must select a node set which is disjointed from the node set selected by the predicate filter [./E2/@a="4"]. In the case, the test is forwarded to our predicate tester without any further modification. The same is done for all attributes, because for every given context node, there is (at most) one occurrence of this attribute.
However, if the DTD allows a single element E1 to have multiple child elements 'E2', then for a given context node E1 there may be one child element 'E2' with a value "3" for the attribute 'a' and another child element 'E2' with a value "4" for the attribute 'a'. Therefore, the expression

'./E2[@a="3"] and ./E2[@a="4"]' (*)

is satisfiable, when the DTD allows multiple child elements E2. Therefore, an XPath expression //E1[./E2/@a="3"] , which only requires the existence of a child 'E2' with an attribute 'a' with a value of "3", may overlap with an XPath expression //E1[./E2/@a="4"] , which requires the existence of a (possibly different) child 'E2' with an attribute 'a' with a value of "4".
That is why in the second case, i.e., when multiple child element 'E2' are allowed for the same element 'E1′, we rename the element names 'E2' to 'E2(1)', 'E2(2)' , … etc., where 'E2(1)' and 'E2(2)' represent possibly different occurrences of 'E2'. Then we rewrite the formula (*) as

./E2(1)[@a=″3″] and ./E2(2)[@a=″4″] (**) .

This formula is forwarded to our predicate tester which now finds out that this conjunction is satisfiable. From a logic point of view, './E2(1)' requires the existence of an element '/E2' which is a child element of the current element, and /E2(2) requires the existence of a possibly but not necessarily different element '/E2' which is also a child element of the current element.[4] Only a predicate '`not( ./E2[@a])`' will be rewritten as '`not( ./E2(X)[@a])`', with a variable `X` which means that whatever value `X` has (for `X=1`, `X=2`, …etc.) no attribute `a` of the element `E2` exists.[5] This is useful for finding out e.g. that the rewritten formula

```
   ./E2(1)[@a="3"]  and  not( ./E2(X)[@a])
```

is unsatisfiable or in other words that

   '`./E2[@a="3"]` implies `./E2[@a]`'

which is needed for the subset test.

3.4. Rewrite rules for predicate filters with loops

Before we start rewriting formulas, we rename all loop variables `N` which belong to independent loops in such a way that we use different variables `N1`, `N2`, … for different loops. Thereafter, we apply the following rewrite rules to the formulas which are received from the subset test or from the overlap test. Let `L` be a loop of parent-axis location steps $(../)^{N}$ with $(N\geq 0)$, and let F1 and F2 be filter expressions. Then we move disjunctions and conjunctions outside of loops, i.e.

```
  "L (F1 or F2)"       ➔   "L(F1) or L(F2)".
  "L (F1 and F2)"      ➔   "L(F1) and L(F2)".
```

For example, "$(../)^{N}$`[@a and @b]`" is replaced with "$(../)^{N}$`[@a] and` $(../)^{N}$`[@b]`". Note that the value for `N` must be identical in both parts of the conjunction.
Thereafter, we apply rewrite rules for filter formulas without loops as follows.

3.5. Rewrite rules for the evaluation of filter formulas without loops

The next step is to apply the following equivalence transformation rules which transform a formula towards a disjunctive normal form (DNF) as close as possible:
Let `A` and `B` be location path expressions (without a loop) which require the existence of a location path `A` or `B` relative to the current node, and let `C` be a constant. We use `eq(A,B)` to describe the equality of `A` and `B` and `neq(A,B)` to describe the inequality of `A` and `B`, where `eq(A,B)` and `neq(A,B)` both assume that both `A` and `B` exist relative to the current node.
We separate path tests from constraints regarding equality and inequality of given attribute values:

```
  " A = B             ➔  " A and B and eq(A,B)"
  " A != B  "         ➔  " A and B and neq(A,B) "
  " not ( A = B ) "   ➔  " not (A) or not (B) or neq(A,B)"
  " not ( A != B ) "  ➔  " not (A) or not (B) or eq(A,B) "
  " A = C  "          ➔  " A and eq(A,C)"
  " A != C  "         ➔  " A and neq(A,C) "
```

[4] In terms of predicate logic, we could write the formula (**) as
 `(∃ ./E2(1)) (∃ ./E2(2)) (./E2(1)[@a="3"]  and  ./E2(2)[@a="4"]  .`
[5] In terms of predicate logic, this would be '`(∀X) not( ./E2(X)[@a])`'

```
  ″ not ( A = C ) ″      ➔  ″ not (A) or neq(A,C)″
  ″ not ( A != C ) ″     ➔  ″ not (A) or eq(A,C) ″ .
```

Let additionally F, F1 and F2 be filter expressions.
We move negations inside location path expressions by applying the following rule[6]:

```
  ″ not ( A [F] ) ″      ➔  ″ not (A) or A[not(F)] ″
```

We move negations inside the formula as far as possible by applying the following rules:

```
  ″ not ( F1 and F2 ) ″➔  ″ not ( F1 ) or not (F2) ″
  ″ not ( F1 or F2 ) ″ ➔   ″ not ( F1 ) and not (F2) ″
  ″ not ( not(F) ) ″   ➔   ″ F ″
```

We move disjunctions outside of conjunctions and location path expressions, i.e.

```
  ″ (F or F1) and F2 ″ ➔   ″ F and F2 or F1 and F2 ″
  ″ A [ F1 or F2 ]″    ➔   ″ A [F1] or A [F2]″.
```

And we move conjunctions outside of location paths, i.e.

```
  ″ A [F1 and F2]″     ➔   ″ A [F1] and A [F2]″.
```

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 using the following algorithm.

3.6. Test algorithm for a single conjunction

We use the following algorithm in order to check whether or not a single conjunction of conditions is satisfiable which extends an algorithm given in [3].

```
(0)If the conjunction contains a location path A and
    it contains an expression not(A) [7]
      return  “conjunction is unsatisfiable” ;
(1)For each constant occurring in the conjunction
      introduce an own equivalence class Ci (1<=i<=m);
(2)For each node path which does not contain a loop
      introduce an own equivalence class Ci (m+1<=i<=n);
(3)For each comparison eq(A,B) found in the conjunction
   with $C_A$ and $C_B$ being the equivalence classes containing
   A and B:
      introduce a new equivalence class $C_{A+B}$
      which contains the union of all elements of $C_A$ and $C_B$
      and thereafter delete $C_A$ and $C_B$. ;
(4)For each equivalence class Ci
      if Ci contains two different constants
            return  “conjunction is unsatisfiable” ;
(5)If the conjunction contains a comparison neq(A,B) where
   A and B are found in the same equivalence class
```

[6] Unfortunately, there is no similar rule which allows us to move a negation inside (or outside) of a loop, because a loop summarizes many alternative places of a filter. This is one source where our tester is incomplete; however, an approach to overcome this incompleteness for the subset test is presented in a forthcoming paper [4].

[7] As mentioned in Section 3.3. not(A) may be of the form not E2(X) and A may be E2(1).

```
      return  “conjunction is unsatisfiable” ;
(6)If the conjunction contains “not(../)$^N$(A)” where A is a
   node path and the conjunction contains a condition
   “Path/A” where “Path/” is unifiable with “(../)$^N$”
      return  “conjunction is unsatisfiable” ;
(7)If the conjunction contains “not (../)$^N$(A!=B)” and
   the conjunction contains a condition neq(Path/A,Path/B)
   where “Path/” is unifiable with (../)$^N$
      return  “conjunction is unsatisfiable” ;
(8)If the conjunction contains “not (../)$^N$(A=B)” and
   the conjunction contains a condition eq(Path/A,Path/B)
   where “Path/” is unifiable with (../)$^N$
      return  “conjunction is unsatisfiable” ;
(9)return  “conjunction is assumed to be satisfiable” ;
```

At first (step (0)), we check for each occurrence of ‘not(A)’ in a conjunction whether or not the same conjunction also contains the location path ‘A’ alone (stating that ‘A’ exists). Remember that not(A) claims that the node set selected by this node path relative to the current node is empty. Therefore, if the conjunction contains also a location path ‘A’, it is unsatisfiable. For example, the condition ‘not(E(X)/@a1)’ states that in the current context no element ‘E’ has an attribute ‘a1’, and the condition ‘E(2)/@a1’ states that an element ‘E’ which has an attribute ‘a1’ exists. In order to find out that the conjunction of both conditions is unsatisfiable, we substitute the variable ‘X’ with the constant ‘2’.
Secondly (steps (1) and (2)), the algorithm introduces an equivalence class for each constant and for each node path which does not contain a loop, where different node paths are considered as being different, i.e. ./E1/@a , ./E1/@b , and ./E1/E1/@a are considered as belonging to three different equivalence classes.
Thirdly (step (3)), for each equality constraint eq(A, B) which occurs in the conjunction, we combine the equivalence class which contains ‘A’ and the equivalence class which contains ‘B’ into a single equivalence class which contains the union of all attribute paths and constants found in at least one of both equivalence classes.
Fourthly, whenever there is an equivalence class which contains 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 unsatisfiable.
At fifth, we consider all conditions of the form neq(A, B) occurring in the conjunction. If we find out, that ‘A’ and ‘B’ belong to the same equivalence class (i.e. that eq(A,B) holds) for at least one condition of the form neq(A, B), then this conjunction is unsatisfiable.
At sixth, if the conjunction contains a term “not (../)^N(A)” where A is a node path, this forbids any ancestor element which can be reached by (../)^N to have a node path A (to an element or attribute). If now the conjunction additionally contains a

condition `Path/A` where “`Path/`” is unifiable[8] with `(../)`^{N}, this would require an ancestor to have such an attribute `A`. That is why the algorithm returns “conjunction is unsatisfiable” .

At seventh, if the condition contains a term “`not (../)`^{N}`(A=B)`” and the conjunction contains a condition `eq(Path/A,Path/B)` where “`Path/`” is unifiable with `(../)`^{N} , then again the loop forbids such a condition `eq(Path/A,Path/B)` to become `true`. Therefore, the algorithm returns “conjunction is unsatisfiable” .

Similarly, if the condition contains a term “`not (../)`^{N}`(A!=B)`” and the conjunction contains a condition `neq(Path/A,Path/B)` where “`Path/`” is unifiable with `(../)`^{N} , then again the loop forbids such a condition `neq(Path/A,Path/B)` to become `true`. Again, the algorithm returns “conjunction is unsatisfiable” .

Otherwise (step (9)) the conjunction is treated as satisfiable. Since our tester is fast but incomplete, step (9) includes unsatisfiable predicates. Note however that treating some unsatisfiable conjunctions (most of which will never occur in practice) as satisfiable allows us to significantly speed up our predicate evaluator and still guarantees correct fallback decisions for all three applications.

4. Fall-back decisions for time-out situations

Our basic idea is to test whether or not access may be granted, a query result may be reused, or transactions may be executed concurrently without access to the XML database. The time consumed for such a test and whether or not it is acceptable depends on the application and may be too short to run a complete tester in rarely occurring cases (i.e. for some pairs of ‘complex’ XPath expressions). Therefore, for each application, we need a fall-back behavior that is used, whenever the time which is needed for such a test exceeds the given time limits.

If the combination of two XPath expressions to be locked for concurrent transactions is too complex for a tester, to decide in time whether or not the XPath expressions overlap, we would not run the transactions in parallel. Since this pessimistic fall-back decision has the same consequences as if the tester finds out, that the XPath expressions overlap, it is sufficient to use an incomplete tester, which returns the pessimistic fall-back decision in such complex situations.

As regards the reuse of query results which are already locally stored (say on a mobile device), if a tester cannot decide in time whether or not a new XPath query expression selects a subset of a previous query result which is already stored on that mobile client, we would not reuse that query result. Since the consequence of this decision is the same as if the tester finds out, that the new XPath expression does not select a subset of a previous one, it is again sufficient to use an incomplete tester. In complex situations, this tester returns a fall-back decision which behaves as if the new XPath expression does not select a subset of the old one.

[8] We say that “`Path/`” and`(../)`^{N} are unifiable, if and only if we can substitute a value for N such that they are identical, e.g. `(../)`^{N} and `../../` are identical, if we are allowed to insert 2 for `N`.

For access rights, there are two alternative approaches. One is to forbid the use of complex access rights, so that no fall-back decision is needed, in which a tester can not decide about the access right in time (i.e. whether or not an XPath expression which is used to access a fragment selects a subset of the nodes for which an access right exists). Complex test situations for our tester can be avoided, if e.g. an XPath expression which is used for an access right does not contain a predicate filter at all or does not contain a comparison of two access expressions in a predicate filter. The other alternative is to allow arbitrary complex access right predicates and to reject a user's access in situations, where a user query would exceed the time limit given for access control. These situations occur very rarely, because access rights are less complex than usual queries, and the test is usually much simpler than for the reuse of previous query results. If in such a case a user's access is rejected, the user query has to be reformulated to solve the problem.
Altogether, the fall-back decision for access control is the same as if the tester finds out, that the accessing XPath expression does not select a subset of nodes selected by the XPath expression for the access right: in complex situations, the tester returns the fall-back decision, i.e., that the accessing XPath expression does not select a subset of the access right XPath expression.
Finally, we can adapt our tester to the specific needs of the application. This means that one application can grant more time in order to achieve more completeness while another decides to reduce completeness in order to raise efficiency. For example, transaction synchronization will use the tester only for those cases which allow an efficient decision (with the effect that it tends to lock larger parts of the document instead of checking locks for a long time).

5. Summary and conclusions

Our contribution to access control, the reuse of previous query results, and locking in XML databases reduces central problems in these application areas to subset tests and overlap tests on pairs of XPath expressions. The basic idea is to use a DTD graph in order to transform XPath expressions into equivalent regular expressions which describe paths of child-axis location steps to accessed nodes, and to right-shuffle predicate filters along these paths to the accessed nodes. Then we reduce the subset test (and the overlap test respectively) for XPath expressions to a subset test (and an overlap test respectively) on the regular node path expressions and a subset test (and an overlap test respectively) on the filter expressions. Our tester is efficiently applicable to a wide variety of practical cases, but incomplete. Therefore, we introduce the concept of fall-back decisions in each application, in order to treat the cases where the tester is incomplete in an appropriate way.
One direction for further research is to reduce the gap where the tester tends to incompleteness, e.g. by covering more cases where a formula contains loops. While a step towards this direction has been contributed in [4] for the subset test, we consider a similar extension to the overlap test to be a promising direction for further research.

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, A. Türling: Access Control and Synchronization in XML Documents. In Proceedings: XML Technologie für das Semantic Web, Berlin 2002, Springer, LNI P-14, 2002.
[4] S. Böttcher, R. Steinmetz: Testing Containment of XPath Expressions in order to Reduce the Data Transfer to Mobile Clients. 7th East European Conference on Advances in Databases and Information Systems, Dresden, September, 2003.
[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. Proc. of the 7 th Int. Conf. on Extending Database Technology (EDBT), Konstanz, March, 2000. Springer, LNCS, Volume 1777.
[7] Alin Deutsch, Val Tannen: Containment and Integrity Constraints for XPath. KRDB, 2001.
[8] Fiebig, T., Helmer, S., Kanne, C.-C., Moerkotte, G., Neumann, J., Schiele, R. Westmann, T.: Natix: A Technology Overview. In A.B. Chaudri et al. (Eds): Web Databases and Web Services 2002, LNCS 2593, pp12-33, 2003.
[9] Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading, MA, 1979.
[10] Michiharu Kudo , Satoshi Hada: XML document security based on provisional authorization, Proceedings of the 7th ACM conference on Computer and communications security, 2000.
[11] Gerome Miklau, Dan Suciu: Containment and Equivalence for an XPath Fragment. PODS 2002: 65-76.
[12] Neven, F., Schwentick, T.: XPath Containment in the Presence of Disjunction, DTDs, and Variables. ICDT 2003: 315-329.
[13] Qun Ren, Margaret H. Dunham: Semantic Caching and Query Processing, SMU Technical Report 98-CSE-04.
[14] Schöning, H., Wäsch, J.: Tamino - An Internet Database System. Proc. of the 7th Int. Conf. on Extending Database Technology (EDBT), Konstanz, March, 2000. Springer, LNCS, Vol. 1777.
[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] Peter T. Wood: Containment for XPath Fragments under DTD Constraints. ICDT 2003: 300-314.
[18] XPath: XML Path Language (XPath) Version 1.0. W3C Recommendation, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.