An Authorization Model for XML DataBases

Alban Gabillon

Université de Pau et des Pays de l'Adour
IUT de Mont de Marsan
LIUPPA/CSySEC
40000 Mont de Marsan, France
(33)558513712

alban.gabillon@univ-pau.fr

ABSTRACT

In this paper, we define a security model for a native XML database which supports the Xupdate language. Our model is inspired by the SQL security model which is the most famous security model for database. We first define a generic access control model for tree data structures. Then, we apply our model to an XML database which supports the Xupdate language.

Keywords

XML, access controls, XPath, Xupdate, security, privilege, permission.

1. INTRODUCTION

Several discretionary access control models for eXtensible Markup Language (XML) documents have been proposed [Ber00][Dam00][KH00][GB01]. For all these models, the approach is more or less the same: the security administrator writes the security policy in a separate authorization sheet; there is a policy which solves the conflicts between the security rules; there is an algorithm which computes the requester's view on XML documents. These models mainly address the read privilege. Some of them consider the write privilege but they do not clearly indicate in which framework the different update operations for XML are supported. In fact, all these security models have been designed to be implemented as extensions to existing web servers. Moreover, these models suffer from data availability problems or illegal inference channels. These problems were pointed out by Stoica and Farkas [SF02].

In this paper our approach is different. Our objective is to define an access control model for a native XML database [Bou99] which supports Xupdate [LM00]. A native XML database (NXD) stores XML in "native" form, generally as some variant of the Document Object Model (DOM) [Vid98] mapped to an underlying data store. NXDs excel at storing document-oriented data which have a very complex structure with deep nesting, and data which are semi-structured in nature. Xupdate is an XML language and today it is a solution for updating XML documents. Xupdate has been defined by the Xupdate working group from the XML:DB initiative [XDB].

The most famous security model for databases is the security model of Structured Query Language (SQL) for relational databases. The SQL security model is a discretionary security model where each table has an *owner*, and the owner gets to choose the access control policy for that table. Users define and update the security policy for their tables by using the well known GRANT and REVOKE commands.

Our model for NXDs is designed to offer the functionalities of the SQL security model. Throughout this paper, our reference will be the Oracle SQL security model as it is described in [TH98]. Our model also aims to provide solutions for the problems raised by Stoica and Farkas. The contribution of this paper to the existing access control models for XML can briefly be summarized as follows:

- We make the separation between the existence of an XML node and its value. We introduce a new position privilege which allows us to know about the existence of a node but not about its value.
- Nodes which were tagged with a position privilege are shown with a RESTRICTED label. Thus, sensitive values are hidden while the structure of the XML document is preserved.
- We extend the read access control algorithm defined in [GB01] with the new position privilege.
- We introduce various update privileges operating at the node level and we define the associated write control algorithm.
- For each Xupdate operation, we specify the required update privileges.
- We borrow functionalities from the SQL security model like the CREATE ROLE or GRANT/REVOKE commands and we adapt them to the XML case.
- We make a comparison of existing access control models for XML.

In section 2, we review the existing access control models for XML data, we remind the functionalities of the SQL security model and we present an overview of our model. In section 3, we define a

This work was supported by funding from the French ministry for research under "ACI Sécurité Informatique 2003-2006. Projet CASC" and the CNRS under "Langages, Types Sécurité et Intégrité pour des données semi structurées (AS 116 (RTP 9))"

Conference: ACM Workshop on Secure Web Services,
DatePlace: October 29, 2004, Fairfax VA, USA.
Year : 2004
Copyright: Copyright 2004 ACM 1-58113-973-X/04/0010....$5.00.

generic access control model for tree data structures. In section 4, we apply our model to an XML database. Finally, in section 5 we conclude this paper

Note that for the sake of simplification, we shall consider throughout this paper that an XML document has only three types of nodes: element, attribute, text (see [Bra00] for a description of the XML data model).

2. OVERVIEW OF OUR MODEL

2.1 Subjects

The development of an access control system requires the definition of the subjects and objects. There is not much to say about the subjects in existing access control models for XML data. In [Ber00], subjects are simply users. In [KH00], they can be users or roles (see [San98] for a description of roles). In [GB01], they are users who are structured into a group hierarchy. In [Dam00], the authors take into consideration the fact that their access control model is to be implemented as an extension to an existing web server. In their model, subjects are users or IP addresses (or a combination of the two), and they are structured into a group hierarchy. In the model presented in this paper, subjects can be users or roles like in the SQL security model.

2.2 Objects

An object is a granule of information which can be protected. Regarding the objects, there are basically different approaches among existing access control models for XML data:

- In [Dam00][KH00], the smallest object is an element. Authorizations specified for an element are intended to be applicable to the element itself, its content (PCDATA) and its attributes.
- In [Ber00], the smallest object is an element or an attribute. Authorizations specified for an element are intended to be applicable to the element itself and its content.
- In [GB01], an object is a node of an XPath tree (see [CD99] for a description of XPath and figure 1 for an example of an XPath tree). A node of an XPath tree can be an element, the content of an element or an attribute.

We want our security model to have a high expressive power. Therefore, in our model, an object is a node of an XPath tree like in the model in [GB01].

Since the XML model is completely different from the relational model it is difficult to compare the granularity level of our model with the granularity level of SQL. Nevertheless, we can say that the SQL object granularity is rather coarse. Indeed, a typical SQL object is a table. This means that users can define authorizations which address tables but cannot define authorizations which address rows. One solution to enforce row-level security is to create a set of views since views are also SQL objects. Many database application designers have been doing this for years. However, this view-based approach can become quite complex and difficult to make foolproof. Another solution is to deploy Fine Grained Access Control (FGAC) [Feu99] which is included in the latest Oracle releases. FGAC allows application designers to define security policies at the row-level security, but unfortunately deploying FGAC is quite a complex task.

2.3 Security Policy

In all the existing models, the approach is the same: the security administrator writes the security policy in a separate authorization sheet. The security policy consists of a set of authorization rules which can be either positive (grant) or negative (deny). The reason for having both positive and negative authorizations is to have a way to specify exceptions to authorizations which are applicable to sets of subjects or objects. Authorization rules address granules by means of Xpath expressions. Conflicts between the rules are solved by a conflict resolution policy.

- The model in [Dam00] supports the read privilege only. In another paper [Dam02], the authors partially address the write privilege but underline the fact that current XML applications are mostly read-only and that no consensus has emerged up to now on a model for XML updates. An authorization specified on an element can be defined as local. In that case, it is applicable to the element itself, its content and its attributes. It can also be defined as recursive. In that case, the permission/prohibition is propagated to the sub-elements. Authorizations can be specified at the document-level or at the Document Type Definitions (DTDs) level. Authorizations specified at the DTD-level propagate to all XML documents that are instances of that DTD. The conflict resolution policy applies "the most specific object takes precedence" principle. According to this principle instance-level authorizations override DTD-level authorizations and a recursive authorization propagates until overridden by a conflicting authorization on a more specific object. For conflicts which cannot be solved by this principle, the authors suggest to apply other principles like "the most specific subject takes precedence" or "denial takes precedence" …etc.
- The model in [Ber00] supports two kinds of privileges: browsing (read) and authoring (write). Authorizations are specified along with propagation options. Depending on its propagation option, an authorization referring to an element may propagate to all the direct and indirect sub-elements, propagate to all the direct sub-elements only, or not propagate at all. Like in the previous model, authorizations can be specified at the DTD-level or at the instance-level. The conflict resolution policy is similar to the previous one.
- The model in [KH00] supports read and write privileges. The authors define three types of propagation policies: no propagation, propagation up (an authorization referring to an element is propagated to all its parent elements) or propagation down (an authorization referring to an element is propagated to all its sub-elements). The conflict resolution policy is either "denials take precedence" or "permissions take precedence". The main contribution of this paper is to propose a provisional authorization model. A provisional authorization is more than a simple permission/denial. Typically, a provisional authorization specifies that a user has to perform a given action (obligation) before he/she is granted a given privilege
- The model in [GB01] supports the read privilege only. The authors do not define any propagation policy. The conflict resolution policy is based on the priority of the different rules.

- The model in [Lim03] is based on the model defined in [Dam00]. The authors add `write` privileges and suggest a technique for efficiently managing access controls in a web environment which emphasizes the integrity of the documents (i.e. validity with respect to `DTDs`). They consider `XQuery` update operators but these operators are not standardized yet [Sur04][Bru03].

All these models define a view-based access control strategy for handling the `read` privilege. Some of them consider the `write` privilege but,

- they do not clearly indicate in which framework the different update operations for `XML` are supported,
- the access control strategy that they use for handling the `write` privilege is not clearly described.

In fact, all these security models have been designed to be implemented as extensions to existing web servers.

```
<files>
 <record login="mrobert">
  <name>Martin Robert</name>
  <diagnosis>
    Pneumonia
  </diagnosis>
 </record>
 <record …>
   …
 </record>
</files>
```

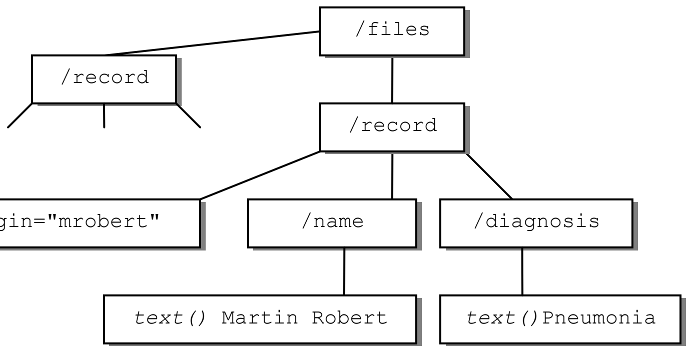

Fig. 1. `XPath` tree representation of an `XML` document

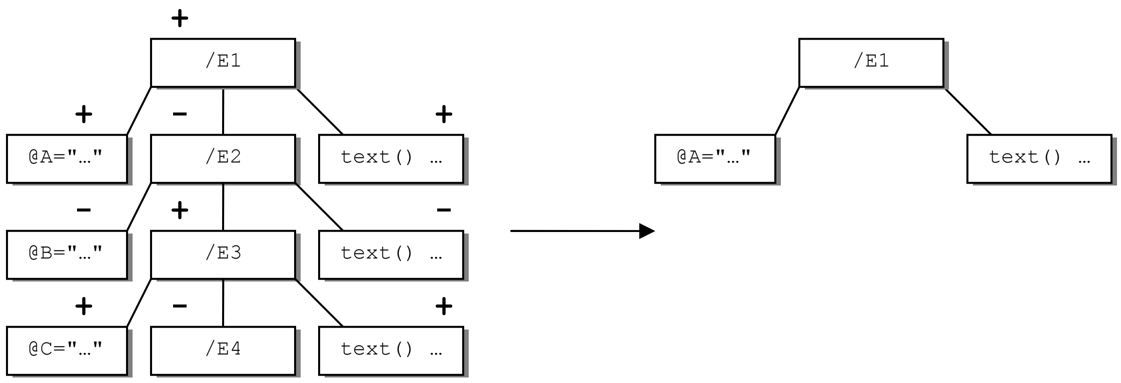

Fig. 2. View access control strategy in model [GB01]

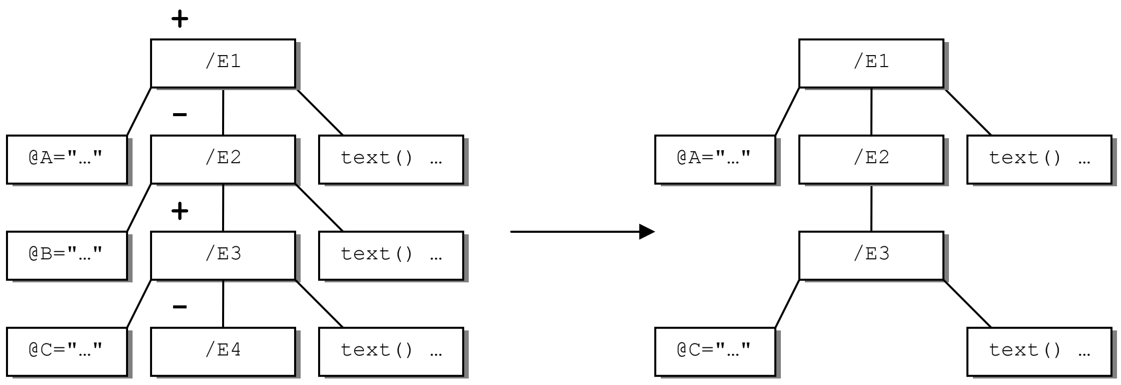

Fig. 3. View access control strategy in model [Dam00]

Moreover, these models suffer from data availability problems or illegal inference channels. These problems were pointed out in [SF02]:

- Regarding the model in [GB01], if access to a node is denied then the user is not allowed to access the entire sub-tree under the node even if access to part of the sub-tree is permitted, therefore limiting the availability of data.
- Regarding the model in [Dam00], in order to preserve the structure of the document, the authors allow start and end tags of elements with negative labels (i.e. access denied) to be released if the element has a descendant with a positive label (access permitted), thus making the semantics of the `read` privilege unclear and allowing inferences about the existence of negative labels and its overall structure.

These two problems are illustrated in figures 2 and 3. Let us assume tag + represents permission and tag – represents denial for a given user `s`.

Figure 2 is related to the model in [GB01]. Recall that this model allows node-level security granularity. The left tree represents a source tree which has been tagged according to the authorizations applying to user `s`. The right tree corresponds to the view user `s` is permitted to see. This view has been computed according to the access control strategy defined in [GB01]. One can see that the sub-tree of which element node `/E2` is the root is not visible at all although user `s` has the permission to see some of the descendant nodes. As it is said in [SF02], this access control strategy reduces the availability of data. One could argue that such a situation comes from a bad security design. Nevertheless, it should be possible to deny access to an internal node while granting access to its descendant nodes.

Figure 3 is related to the model in [Dam00]. Recall that this model allows element-level security granularity. The left tree represents the source tree which has been tagged according to the authorizations applying to user `s`. The right tree corresponds to the view user `s` is permitted to see. This view has been computed according to the access control strategy defined in [Dam00]. One can see that element `/E2` is visible in the view while element `/E4` which has also a negative tag is not visible at all. This has two consequences: it makes the semantics of the `read` privilege unclear and user `s` can infer about the existence of a negative label assigned to element `/E2`.

In the model presented in this paper, we consider that each document has an owner who acts as an administrator for that document. The owner is the user who created the document. The owner has all privileges regarding the document and defines the security policy by issuing some `GRANT` (permissions) and `REVOKE` commands. Like in `SQL`, the default policy is always "denial".

Our model supports three kinds of write privileges (`insert`, `delete` and `update`). The semantics for each of these privileges is given in section 3. Since our model is to be implemented in a native `XML` database supporting `Xupdate`, we specify the privilege that each `Xupdate` operation requires for completion. In section 3, we fully describe our access control strategy for these privileges.

Regarding the problem described in figures 2 and 3, our conclusion is that this problem cannot be solved without introducing a privilege which protects the *existence* of nodes. In section 3, we give a formal definition of a node. Intuitively a node is defined by a position and a value. Therefore, our model includes two kinds of read privileges: one privilege which allows knowing the existence of a node (we call it the `position` privilege) and another privilege which allows knowing both the existence and the value (we call it simply the `read` privilege).

The security designer has now two possibilities:

- either he/she considers that user `s` *should not be aware* of the existence of node `/E2` and denies `position` privilege on node `/E2` to user `s`. In that case, the view-based access control strategy we define in section 3 computes the view of figure 2. Note that in such a situation, it would not make sense to grant `read` privilege to user `s` on some descendant nodes of node `/E2` since it would not be possible to show these nodes to user `s` without disclosing the existence of node `/E2`.
- or he/she considers that only the value of node `/E2` should be hidden from user `s` and grants `position` privilege on node `/E2` to users `s`. Figure 4 shows the same example as in figures 2 and 3 but with a major difference: user `s` has been granted the `position` privilege (denoted by **`P`**) on element `/E2`. The right tree corresponds to the view user `s` is permitted to see according to the view-based access control strategy we define in section 3. One can see that element `/E2` has been replaced by the special value `RESTRICTED`. This value was first used by Sandhu and Jajodia in the context of multilevel databases [SJ92]. Its semantics is "the value exists but you are not allowed to see it".

3. ACCESS CONTROL MODEL

In this section, we define an access control model for tree data structures. In section 4 we apply our model to the special case of `XML` trees.

3.1 Definition of a Tree

A tree is a hierarchy of *nodes*. Each node is the parent of zero or more *child* nodes. Each node has one and only one parent except one fundamental node which has no parent and which is called the *root* node (or simply the root). Trees are often called *inverted trees* because they are normally drawn with the root at the top.

A node is defined by its *position* and its *value*. The position of a node is represented by a sequence of positive integers. The empty sequence is denoted by ε. A tree is defined by a set of nodes.

For example, the tree in figure 5 is defined by the following set:

`{(ε,a),(1,b),(2,c),(11,a),(12,d),(121,b)}`

Position of the root node is ε and its value is `a`. Position of the next node is `1` and its value is `b`. Position `1` means that this node is the first child of the root (from left to right). Position of the next node is `2` and its value is `c`. Position `2` means that this node is the second child of the root. Position of the next node is `11` and its value is `a`. Position `11` means that this node is the first child of the first child of the root … etc.

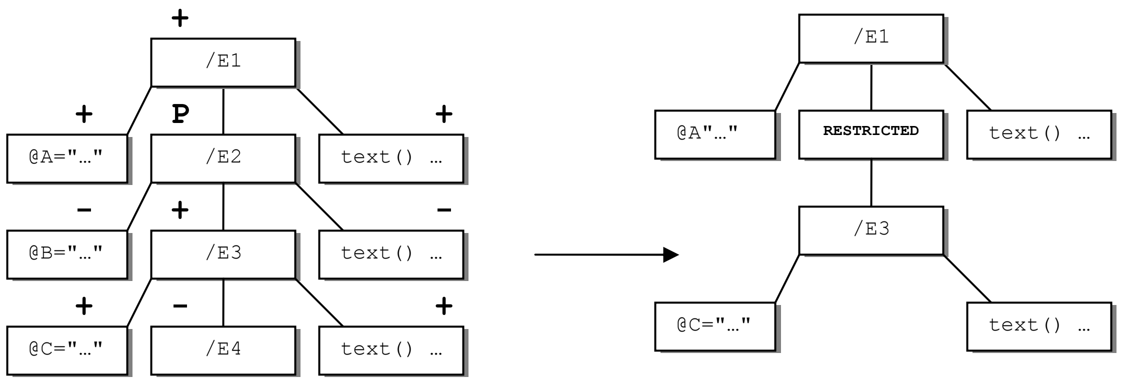

Fig. 4. View access control strategy in our model

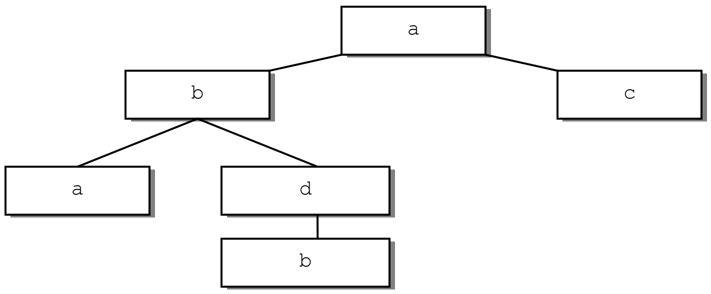

Fig. 5. Example of tree

3.2 Subjects and Objects

A subject is a user or a role. A security object is a node. The user who creates the root of a tree is the *owner* of that tree. This user has all privileges on all the nodes of the tree. Our model also includes a special subject: the DataBase Administrator (`DBA`) who has all privileges.

3.3 Security Policy

The security policy of our model includes the following privileges:

`{position, read, delete, insert, update}`

- if user `s` is granted privilege `position` on node `v` then user `s` is granted the right to know the existence of `v`.
- if user `s` is granted privilege `read` on node `v` then user `s` is granted the right to see node `v`.
- if user `s` is granted privilege `insert` on node `v` then user `s` is granted the right to add a new sub-tree to node `v`.
- if user `s` is granted privilege `update` on node `v` then user `s` is granted the right to update node `v` (i.e. change its value).
- if user `s` is granted privilege `delete` on node `v` then user `s` is granted the right to delete the sub-tree of which node `v` is the root.
- each of these privileges can be associated with the `grant_option`. If user `s` is granted privilege `p` on node `v` with the `grant_option` then user `s` is granted the right to transfer privilege `p` on node `v` to other users (with or without the `grant_option`).

The owner of a tree is the user who created the root node of that tree. The owner of a tree holds all privileges (with the `grant_option`) on all the nodes of that tree even the nodes which are inserted by other users.

Privileges should not be confused with *operations*. Operations need privileges to complete. Operations depend on applications. For example, in one application, we might need an *append* operation which always performs the insertion of a sub-tree at the last position (the rightmost position), whereas in another application one might need an *insert* operation which is able to perform the insertion of a sub-tree at any position. Both operations need the `insert` privilege to complete.

3.4 Security Policy Control Language

In this section, we define the core of a security policy control language with which users administrate privileges on their trees.

Our language is based on the `GRANT and REVOKE` commands which are well known in the database community:

```
GRANT <set of privileges> ON <set of nodes> [/P]
TO <set of subjects> [with grant_option]
```

```
REVOKE <set of privileges> ON <set of nodes>
[/P] FROM <set of subjects>
```

`<set of privileges>` is a subset of `{position, read, insert, delete, update}`.

`<set of nodes>` addresses nodes which belong to the source tree. Nodes can be addressed by their positions. We could also define a language which would allow us to select nodes on the basis of their content. In section 4, we shall use the `XPath` language to address nodes of an XML tree. Option /P means "propagate". It extends the effect of the GRANT/REVOKE commands to the descendant nodes of the nodes addressed by <set of nodes>.

<set of subjects> addresses users or roles by means of their identity.

Option grant_option indicates that subjects are also granted the privilege to grant/revoke the permissions which are defined by the command.

There is no difference between the behavior of the GRANT and REVOKE commands of our model and the behavior of the GRANT and REVOKE commands of SQL for object privileges. Let v be a node, let p be a privilege, let s be a user:

User s may grant privilege p on node s to other users if:

- user s is the DBA or,
- user s is the owner of the tree to which node v belongs or,
- user s holds privilege p on node v with the grant_option.

User s may revoke privilege p on node v from another user s′ if:

- user s is the DBA or,
- user s previously granted privilege p to user s′. Note that if user s granted privilege p to user s′ with the grant_option and user s′ granted privilege p to some other users, then there is *cascade revocation*. Privilege p is also revoked from all these other users. More precisely, privilege p is revoked from all the descendant users of s in the *authorization graph* of p (see the SQL GRANT/REVOKE scheme for more details).

Like in SQL, creation of subjects is done via the following commands:

```
CREATE USER <login_name>
```

```
CREATE ROLE <role_name>
```

And assignment of roles to users (or roles) is done via the following command:

```
GRANT <roles> TO <set of subjects>
```

Creating subjects and assigning roles to users are system privileges which can be held only by the DBA (and a limited number of application designers).

Finally there is another system privilege which gives the right to create a tree.

```
CREATE TREE <tree>
```

tree obeys the definition given in section 3.1.

3.5 Access Control Algorithms

In this section we propose two access control algorithms. The first algorithm, which we call View Control algorithm, is taken from the model in [GB01]. We refined it so that it takes into account the position privilege. It computes the view of the source tree a given user is permitted to see. The second algorithm which we call Write Control algorithm is able to determine whether a given user is permitted to perform a write operation on a particular node.

View Control Algorithm

```
1. Let S be the user for which the view has to
   be computed
2. Let L be an empty list of nodes
3. Insert the root node into L
4. Let R be an empty list of nodes
5. While L is not empty Do
6.    N ← the first node of L
7.    If S holds the read or the position
      privilege on N then
8.       If S does not hold the read privilege
         on N then
9.          Replace the value of N with the
            special value RESTRICTED
10.      Append N to R
11.      Replace N in L with all the child nodes
         of N
12.   Else
13.      Remove N from L
```

This algorithm traverses the tree in pre-order. After the algorithm terminates, R contains the pre-order list of the nodes which belong to the view.

Regarding this algorithm, we can make the following comments:

- Line 9: S holds the position privilege on N i.e. S is permitted to know the existence of node N but not its value. Therefore, value of node N will be equal to RESTRICTED in the view.
- Line 10: Node N is inserted in the view.
- Line 11: Node N is replaced in L with all its child nodes so that the while loop can start a new iteration with the first child node of N.
- Line 13: S neither holds the read privilege nor the position privilege on N. Therefore, node N should not appear in the view user S is permitted to see. Node N is removed from L and is not appended to R. Note that N is *not* replaced in L with its child nodes. The child nodes of N (and consequently all the descendant nodes of N) cannot be included in the view. Authorizations referring to the descendant nodes of N are not even checked.

We can easily deduce that the tree which is computed by the View Control algorithm is a *pruned* version of the source tree (however, a few node values may have been replaced by RESTRICTED).

Let us apply the View Control algorithm to a sample medical files database (see figure 6). Let s be a user. Label **r** (respectively **p**) attached to a node represents the fact that user s holds the read (respectively position) privilege on that node.

The view user s is permitted to see is represented in figure 7. User s is permitted to read illnesses (most probably for statistic purposes) without knowing the patients. Note that some nodes in the view are not given their absolute position in the original source tree. Indeed, disclosing the absolute position of these nodes *conflicts* with the fact that the existence of some other nodes should be hidden. For example, user s should not be informed that the actual position of each diagnosis node in the source tree is respectively 12 and 22 because it would disclose the existence of hidden doctor nodes 11 and 21.

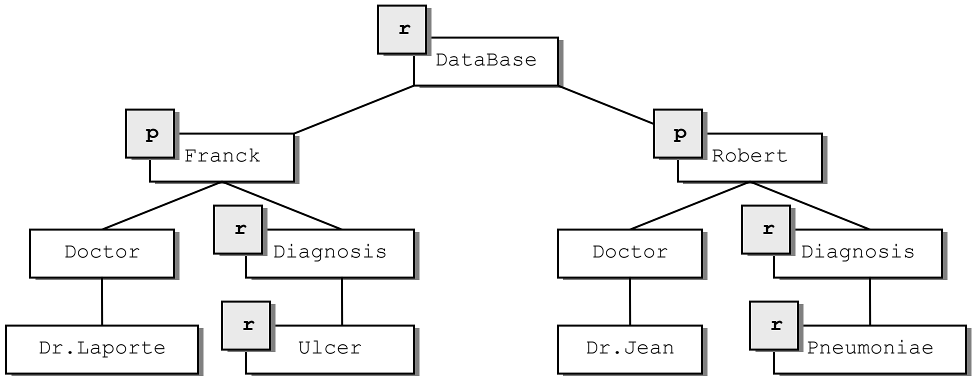

Fig. 6. Medical files database

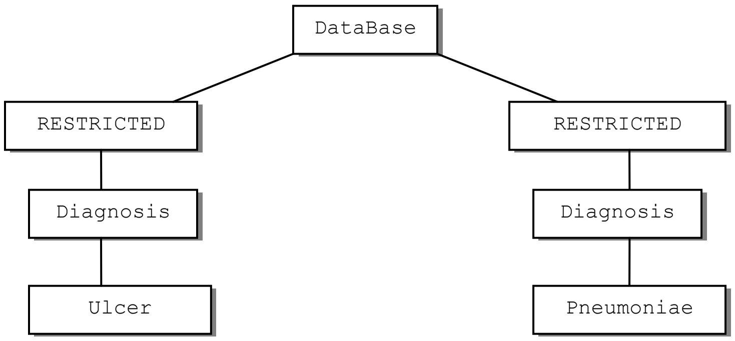

Fig. 7. View for subject `s`

Whenever a user starts a session to query the database, the view the user is permitted to see is computed first[1]. Then queries are evaluated on the view.

Let us now present the `Write Control` Algorithm. In order to answer the question "Is user `s` permitted to perform an operation which requires the `delete|insert|update` privilege on node `n` ?", we apply the following simple strategy:

`View Control Algorithm`

```
1. Let S be the user performing the operation O
   which requires the WRITE[2] privilege on node
   N.
2. If S holds the WRITE privilege on node N then
3.    operation O is accepted
4. Else
5.    operation O is rejected
```

The `write control` algorithm does nothing except checking whether the user is holding the appropriate privilege or not. *The algorithm does not check whether node `N` belongs or not to the view the user is permitted to see*. This means that our security model enables *blind writes*. This approach fully conforms to the `SQL` security model. In `SQL`, a user may perform a write operation (`update`, `insert` or `delete`) on a table without holding the `select` privilege on that table. This approach has the advantage of being very flexible but requires a lot of care when designing the security policy. Indeed it is possible to build covert channels by using blind writes. For example, let us consider `user_A` who is the owner of the `employee` table and who has granted to `user_B` the sole `update` privilege on `employee`.

`user_B` is not permitted to see `user_A`'s `employee` table,

```
SQL> select * from user_A.employee;
ERROR ORA-01031: insufficient privilege
```

but `user_B` is permitted to update `user_A`'s `employee` table:

```
SQL> update user_A.employee set salary = salary
+ 100 where salary > 3000;
2 rows updated
```

Although `user_B` is not permitted to see `user_A`'s employee table, he has been able to learn, through an update command, that there are two employees with a salary greater than 3000.

In our model, similar covert channels appear whenever a user grants some `write` privilege(s) on a node without granting the `read` privilege on that node. Therefore, it is the responsibility of each user, who acts as a security administrator for his/her trees, to be aware of these potential covert channels.

Another issue which should be carefully taken into consideration when designing the security policy is related to the `delete` privilege. If a user holds a `delete` privilege on a given node then the user is allowed to delete the sub-tree of which that node is the root even though the user does not hold the `delete` privilege on the descendant nodes. When designing the security policy for their trees, users have also to be aware of this fact since it can be seen as a threat against integrity.

[1] "Computing the view of the database each user is permitted to see" is purely conceptual. How to implement our model with techniques offering good performances is beyond the scope of this paper.

[2] Recall that `WRITE` stands for either `INSERT`, `DELETE` or `UPDATE`

4. APPLICATION TO XML

In this section, we first make a short presentation about native XML databases. Then, we apply to XML databases, the model we defined in the previous section.

4.1 Native XML database

Native XML databases are tailored to the storage of XML documents. According to the XML:DB initiative, a native XML Database (NXD),

- defines a (logical) model for an XML document and stores and retrieves documents according to that model. Examples of such models are the XPath data model, the XML Infoset, and the models implied by the DOM and the events in SAX 1.0.
- has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage.
- is not required to have any particular underlying physical storage model. For example, it can be built on a relational, hierarchical, or object-oriented database, or use a proprietary storage format such as indexed, compressed files.

Many native XML databases support the notion of *collection*. A collection plays a role similar to a directory in a file system. Native XML databases can store XML documents without knowing their schema (DTD), assuming one even exists [Lio02].

Native XML databases have a variety of strategies for updating documents. As a general rule, each product that can modify fragments of a document has its own language, although a number of products support the Xupdate language from the XML:DB initiative.

4.2 Subjects

A subject is a user or a role. Each user has an identifier which can be the login name. Each user is assigned to one or several roles. DBA creates users and the roles hierarchy with the CREATE USER and CREATE ROLE commands (see section 3.4). Here is an example:

```
CREATE ROLE staff        CREATE USER mrobert
CREATE ROLE secretary    CREATE USER laporte
CREATE ROLE doctor       CREATE USER beaufort
CREATE ROLE nurse        CREATE USER durand
CREATE ROLE patient
GRANT staff TO secretary, nurse, doctor
GRANT secretary TO beaufort
GRANT doctor TO laporte
GRANT nurse TO durand
GRANT patient TO mrobert
```

In this example, each user is assigned to at least one role. There are two basic roles: staff and patient. staff role has three sub-roles: doctor, nurse, and secretary.

4.3 Objects

A security object is a node of an XPath tree. The user who creates an XML document is the *owner* of that document. This user has all privileges on all the nodes of the document. We replace the CREATE TREE command of the section 3.1 with the following command:

```
CREATE DOCUMENT <document>
```

document is a well formed XML document.

Objects are selected via Xpath patterns from the source document which is to be protected. We assume the XPath tree representing the document is traversed. Nodes that match the pattern are selected.

4.4 Security Policy

Users define the security policy for their documents by using the GRANT/REVOKE commands defined in section 3. Here are some examples of GRANT and REVOKE commands which are issued by the owner of the document shown in figure 1:

```
1.  GRANT read /P ON files TO staff
2.  GRANT read /P ON files TO doctor WITH
    grant_option
3.  REVOKE read ON diagnosis/text() FROM
    secretary
4.  GRANT position ON diagnosis/text() TO
    secretary
5.  REVOKE read ON record/@login FROM staff
6.  GRANT position ON files TO patient
7.  GRANT read ON record[@login=$user] /P TO
    $user
8.  GRANT insert ON files TO secretary
9.  GRANT insert ON name TO secretary
10. GRANT update ON name/text() TO secretary
11. GRANT insert ON diagnosis TO doctor
12. GRANT update, delete ON diagnosis/text() TO
    doctor
13. GRANT update ON record[@login=$user]/@login
    TO $user
```

Staff members have the permission to see the whole document (line 1 – note that option /P is used). Doctors have the privilege to administrate the read privilege on nodes of the document (line 2). Secretaries are forbidden to see the content of diagnosis elements (line 3). They see RESTRICTED instead (line 4). Staff members are forbidden to see login attributes (line 5). Patients cannot see the document. They see RESTRICTED instead (line 6). However each patient can see his/her own record (line 7). $user is a variable which is instantiated with the login of the user currently accessing to the XML database. Secretaries have the permission to insert new records (line 8) and to write patient names (lines 9 & 10). Doctors have the permission to write diagnosis (lines 11 & 12). Patients have the permission to update their own login name (line 13).

Like in SQL, the last issued command has the *priority* over the previous ones and possibly cancels them. For example, command 1 which says that staff members have the permission to see the whole document is partially cancelled by command 3 which revokes the right to see part of the document from some staff members (secretaries).

The fact that users can transfer their privileges (with the grant_option) cannot create conflicts in the security policy.

Indeed, in the GRANT/REVOKE scheme, nobody can revoke a privilege that he/she has not granted (see section 3.4). This has an implicit consequence: if somebody is granted the same privilege from two different users then that person will loose that privilege only if both users revoke it (see the GRANT/REVOKE scheme of SQL for more details).

Almost all existing discretionary models for XML define the concept of *instance level* and *DTD level* authorisation. An instance level authorisation applies to a specific XML document. A DTD level authorisation applies to a set of documents which conforms to a specific DTD. In our model, we do not have DTD-level authorizations. Users define the security policy for their documents. As mentioned in section 4.1, many existing native databases for XML organize XML documents without schema. However, most of these databases include the concept of collection and very often collection names can be used in XPath expressions, meaning it is possible to write XPath expressions across documents. Therefore, if a user owns several documents which are arranged into a collection then that user may issue GRANT and REVOKE commands addressing several documents.

The model in [Dam00] addresses the problem of preserving the validity of the views with respect to the DTD of the original document. Indeed, if a required node in the DTD is not shown in the view the user is permitted to see, then that user can infer about the existence of such a hidden node. Their model does not include the possibility of protecting portions of DTDs. Therefore, they suggest applying a loosening transformation to the DTD so that the view becomes valid with respect to the loosened version of the DTD. Our model does not address this issue which actually belongs to the general problem of inference analysis. A DTD is nothing more than a set of integrity constraints. The security administrator might decide to make a (complex) analysis of possible inference channels which may arise because of inconsistencies between the policy applying to the integrity constraints and the policy addressing the data. Such an analysis is sometimes made for databases which require a high confidentiality level such as multilevel databases [CG99]. In our model, if we use XML schema instead of DTDs then nothing prevents the owner of the schema from granting to other users the permission to see or partially see the integrity constraints. Indeed an XML schema is an XML document. However, the problem of closing inference channels which may arise because of inconsistencies between the policy addressing the schema and the policy addressing the instances is beyond the scope of this paper and is rarely put forward in the context of discretionary models like the SQL security model for instance.

4.5 Querying the database

Queries expressed by users are evaluated on views which are computed by the algorithm we give in section 3.5. Queries can be expressed with XPath or any other query language for XML. Figures 8, 9 and 10 show the views for different users. Doctors and nurses can read the whole document except logins (figure 8). The only difference between nurses and doctors is that doctors have been granted the grant_option for the read privilege on the whole document. Secretaries can read everything except diagnosis and logins (figure 9). Martin Robert can access to his own record only (figure 10).

```
<files>
 <record>
  <name>Martin Robert</name>
  <diagnosis>
    Pneumonia
  </diagnosis>
 </record>
 <record>
   …
 </record>
</files>
```

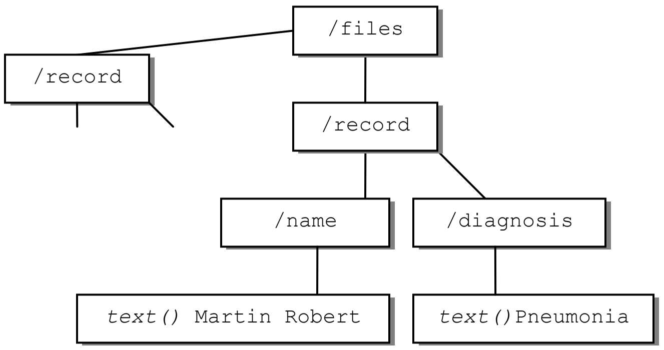

Fig. 8. View for doctors and nurses

```
<files>
 <record>
  <name>Martin Robert</name>
  <diagnosis>
    RESTRICTED
  </diagnosis>
 </record>
 <record>
   …
 </record>
</files>
```

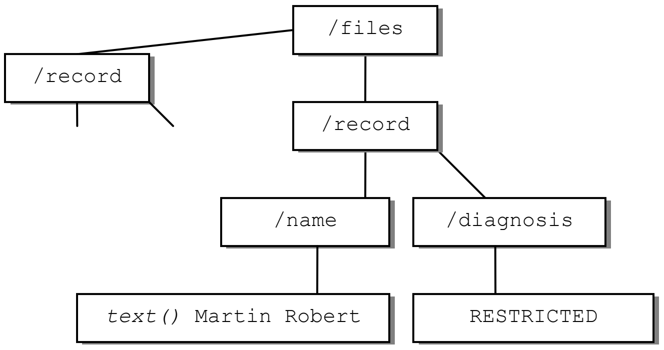

Fig. 9. View for secretaries

```
<RESTRICTED>
 <record login="mrobert">
  <name>Martin Robert</name>
  <diagnosis>
    Pneumonia
  </diagnosis>
 </record>
</RESTRICTED>
```

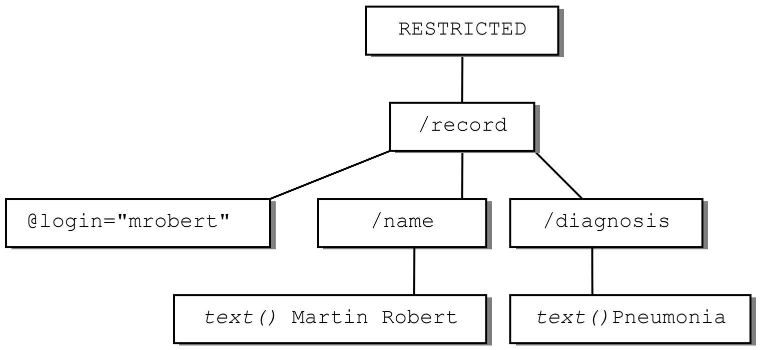

Fig. 10. View for Martin Robert

4.6 Updating the database

Updating XML data is still a research issue (see [Tat01][Sur04][Bru03] for instance). Today Xupdate is the most achieved solution to update XML data. Other proposals like update operators for XQuery are not standardized. As we said before, our security model is to be implemented in a native XML database supporting the Xupdate language.

An update in the XUpdate language is expressed as a well-formed XML document. XUpdate makes use of the XPath language for selecting nodes to update. An update is represented by an xupdate:modifications element in an XML document.

Xupdate operations have a required select attribute, which specifies the *context node* selected by an XPath expression.

In this section we present through some examples the main operations of the Xupdate language (see [LM00] for a complete description of Xupdate). We also specify the permission that each operation needs for being accepted by the write control strategy we defined in section 3.5.

Creating node operations

There are three XUpdate instructions for creating XML fragments: insert-before, insert-after and append.

insert-before inserts the given node as the preceding sibling of the selected context node, where insert-after inserts the given node as the following sibling of the selected context node. The append operation allows a node to be created and appended as a child of the context node.

For being accepted by the write control strategy we defined in section 3.5, these operations need the following privilege:

- Operation insert-before requires the insert privilege on the parent node of the context node.
- Operation insert-after requires the insert privilege on the parent node of the context node.
- Operation append requires the insert privilege on the context node.

The following example shows the insertion of a new medical record into the document which is represented in figure 1.

```
<xupdate:modifications version="1.0">
   <xupdate:insert-before
   select="/files/record[1]">
     <xupdate:element name="record">
        <xupdate:attribute
        name="login">pfranck</xupdate:attribute>
          <name>Patricia Franck</name>
          <diagnosis/>
       </xupdate:element>
    </xupdate:insert-before>
</xupdate:modifications>
```

This document uses an insert-before operation (context node is the element /files/record[1]) and performs the insertion of the following XML fragment into the document represented in figure 1:

```
<record login="pfranck">
  <name>Patricia Franck</name>
  <diagnosis/>
</record>
```

According to the policy we defined in section 4.4, only secretaries (and the owner of the document) may successfully submit this operation to the database. Indeed, only secretaries hold the insert privilege on node /files which is the parent node of the context node.

This second example performs the insertion of a text node as a child of the Patricia Franck's diagnosis element:

```
<xupdate:modifications version="1.0">
   <xupdate:append select=
   "/files/record[@login="pfranck"]/diagnosis">
     <xupdate:text>
        Ulcer
     </xupdate:text>
   </xupdate:append>
</xupdate:modifications>
```

This document uses an append operation. The context node is the element,

/files/record[@login="pfranck"]/diagnosis.

Now, Patricia Franck's record is as follows:

```
<record login="pfranck">
  <name>Patricia Franck</name>
  <diagnosis>Ulcer</diagnosis>
</record>
```

According to the policy we defined in section 4.4, only doctors may successfully submit this operation to the database. Indeed, only doctors hold the insert privilege on the context node. Note that the fact that doctors are not permitted to see logins would not make this operation to fail although the login attribute is referred to by the select attribute of the xupdate:append element. As we said before, our model enables blind writes. It is exactly the same in SQL: a user holding the update privilege on a table may perform on that table an update operation containing a where clause, even though this user does not hold the `select` privilege on that table.

Updating node operations

There are two `XUpdate` instructions for updating `XML` nodes: `update` and `rename`

`update` can be used to update the content of existing nodes. `rename` allows an attribute or element node to be renamed after its creation.

- If the context node is an element, then operation `update` requires the `update` privilege on the content (text node) of the context node.
- If the context node is an attribute, then operation `update` requires the `update` privilege on the context node.
- Operation `rename` requires the `update` privilege on the context node.

Renaming an attribute or updating its value requires the `update` privilege on the context node (which is the same for the two operations) which encapsulates both the attribute and its value.

On the contrary, renaming an element requires the `update` privilege on the context node and updating its content requires the `update` privilege on the content node itself.

The following example changes the name of `Patricia Franck` to `Pamela Franck`.

```
<xupdate:modifications version="1.0">
   <xupdate:update select=
   "/files/record[@login="pfranck"]/name">
       Pamela Franck
   </xupdate:update>
</xupdate:modifications>
```

This document uses an `udpate` operation. The context node is the element `/files/record[@login="pfranck"]/name`.

According to the policy we defined in section 4.4, only secretaries may successfully submit this operation to the database. Indeed, only secretaries hold the update privilege on node `/files/record[@login="pfranck"]/name/text()` which is the content of the context node.

The next example renames all elements `name` in `full_name`.

```
<xupdate:modifications version="1.0">
   <xupdate:rename select="/files/record/name">
       Full_name
   </xupdate:rename>
</xupdate:modifications>
```

This document uses a `rename` operation. Context nodes are elements `/files/record/name`.

According to the policy we defined in section 4.4, no user (except of course the owner of the document) may successfully submit this operation to the database. Indeed, nobody has the permission to update nodes `/files/record/name/`.

Deleting node operations

There is one `XUpdate` instruction for deleting `XML` nodes: `remove`

`remove` operation allows a node to be removed from the result tree.

- Operation `remove` requires the `delete` privilege on the context node.

The following example deletes Pamela Franck's medical file

```
<xupdate:modifications version="1.0">
   <xupdate:remove select=
   "/files/record[@login="pfranck"]"/>
</xupdate:modifications>
```

According to the policy we defined in section 4.4, no user may successfully submit this operation to the database. Indeed, nobody has the permission to delete medical files.

The next example deletes the content of the Pamela Franck's diagnosis element

```
<xupdate:modifications version="1.0">
  <xupdate:remove select=
  "/files/record[@login="pfranck"]/
  diagnosis/text()"/>
</xupdate:modifications>
```

According to the policy we defined in section 4.4, doctors have the permission to delete the content of diagnosis elements.

Copying/Moving node operations

Xupdate does not include explicit copying/moving operations but allows performing such actions by using the `variable` and `value-of` operations.

`variable` defines a variable which is bound to the context node. `value-of` returns the value of an existing variable.

- Operation `variable` requires the `read` privilege on the context node.
- Operation `value-of` does not require any privilege.

The following example inserts a copy of the Pamela Franck's medical file at the end of the document.

```
<xupdate:modifications version="1.0">
  <xupdate:variable name="myvar"  select=
  "/files/record[@login="pfranck"]"/>
     <xupdate:append select="/files">
        <xupdate:value-of select="$myvar"/>
     </xupdate:append>
</xupdate:modifications>
```

Variable `myvar` is bound to Pamela Franck's medical file. The `append` operation inserts the value of `myvar` at the end of the document.

According to the policy we defined in section 4.4, secretaries may successfully submit this copy operation since they have the permission to read `record` nodes and they hold the `insert` privilege on node `/files`.

The last example moves the Martin Robert's medical file at the beginning of the document.

```
<xupdate:modifications version="1.0">
  <xupdate:variable name="myvar" select=
  "/files/record[@login="mrobert"]"/>
     <xupdate:insert-before select=
     "/files/record[1]">
        <xupdate:value-of select="$myvar"/>
     </xupdate:insert-before>
     <xupdate:remove select=
     "/files/record[@login="mrobert"]"/>
</xupdate:modifications>
```

Variable `myvar` is bound to Martin Robert's medical file. The `insert-before` operation inserts the value of `myvar` at the beginning of the document. The `remove` operation deletes the original file.

According to the policy we defined in section 4.4, no user except the owner of the document may successfully submit this move operation since nobody has the permission to delete `record` nodes.

Regarding macro operations like the copy operation or the move operation we have two possibilities for implementing the write control strategy:

- If we consider that each `Xupdate` file is an atomic operation then the write control strategy would reject any operation for which the user does not hold all the necessary privileges. Regarding the previous example, the move operation would be accepted only if it is submitted by a user who holds the `read` and the `delete` privileges on Martin Robert's record and the `insert` privilege on the root.
- Now, if we consider that only basic `Xupdate` operations are controlled then a macro operation like the previous move operation may partially be executed. For example, the `insert-before` operation succeeds while the next `remove` operation fails. In that case, according to the principle of atomicity in database transactions, the `insert-before` operation has to be undone after the `remove` operation is rejected.

5. CONCLUSION

In this paper we have defined a security model for a native `XML` database which supports the `Xupdate` language. Our model is inspired by the `SQL` security model. We have implemented a prototype which simulates an `XML` database supporting our model. Due to space limitations we could not show how we have implemented the authorizations and the access controls. Let us however mention that our prototype is based on `Cocoon` [Coc] and the `Xindice` database [Xin]. We have implemented the view-based access control strategy with an `XSLT` processor [Cla99]. For implementing the write control strategy we have adapted the technique which is used in Oracle's `FGAC`. We modify on the fly the `select` attribute of a `Xupdate` operation with a condition which reflects the permissions of the user submitting the operation.

Our model can be refined. We are planning to work on the subjects. We might extend the concept of ownership. In the future version of our model, we might explicitly include the concept of collection and have three types of *owner*: the owner of a collection, the owner of a document and the owner of a node as it is done in [Gab02].

A minor issue that we could also investigate is how to dissociate attributes from their value. Since attributes and their value are encapsulated within a single `XPath` node, it is not possible to dissociate an attribute from its value with the `XPath` language. Therefore permissions referring to an attribute apply to the attribute itself and its value.

6. REFERENCES

[Ber00] E. Bertino, S. Castano, E. Ferrari and M. Mesiti. "Specifying and Enforcing Access Control Policies for XML Document Sources". World Wide Web Journal, vol. 3, n. 3, Baltzer Science Publishers. 2000.

[Bou99] Ronald Bourret. "XML and Databases". http://www.rpbourret.com/xml/XMLAndDatabases.htm

[Bra00] T. Bray et al. "Extensible Markup Language (XML) 1.0". World Wide Web Consortium (W3C). http://www.w3c.org/TR/REC-xml (October 2000).

[Bru03] E. Bruno, J. Le Maitre and E. Murisasco, "Extending XQuery with Transformation Operators", Proceedings of the 2003 ACM Symposium on Document Engineering (DocEng 2003), ACM Press, Grenoble, France, November 20-22 2003, pp. 1-8. [Réf. F75].

[CD99] J. Clark and Steve DeRose . "XML Path Language (XPath) Version 1.0". World Wide Web Consortium (W3C). http://www.w3c.org/TR/xpath (November 1999).

[CG99] F. Cuppens, A. Gabillon. Logical Foundations of Multilevel Databases. Data and Knowledge Engineering, vol. 29, 1999, pp. 259-291. Elsevier.

[Coc] Apache software foundation. Cocoon, http://xml.apache.org/cocoon/index.html

[Cla99] J. Clark. "XSL Transformations (XSLT) Version 1.0". World Wide Web Consortium (W3C). http://www.w3c.org/TR/xslt (November 1999).

[Dam00] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati, "Securing XML Documents," in Proc. of the 2000 International Conference on Extending Database Technology (EDBT2000), Konstanz, Germany, March 27-31, 2000.

[Dam02] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati, " A Fine-Grained Access Control System for XML Documents," in ACM Transactions on Information and System Security (TISSEC), vol. 5, n. 2, May 2002, pp. 169-202.

[Feu99] Steven Feuerstein. Oracle PL/SQL guide. Guide to Oracle8i Features. Deploying Fine-Grained Access Control (chapter 8). O'Reilly 1999.

[GB01] Alban Gabillon and Emmanuel Bruno. "Regulating Access to XML documents". Fifteenth Annual IFIP WG 11.3 Working Conference on Database Security. Niagara on the Lake, Ontario, Canada July 15-18, 2001.

[Gab02] A. Gabillon, M. Munier, JJ. Bascou, L. Gallon, E. Bruno. "An Access Control Model for Tree Data Structure". Infomation Security Conference. Sao Paulo, Brasil. October 2002.

[KH00] M. Kudo and S. Hada. "XML Document Security based on Provisional Authorisation". Proceedings of the 7th ACM conference on Computer and communications security. November, 2000, Athens Greece.

[Lim03] C. Lim, S. Park, and S. H. Son, " Access Control of XML Documents considering Update Operations," ACM Workshop on XML Security, Fairfax, VA, Oct. 2003

[Lio02] Matt Liotta. “Apache's Xindice Organizes XML Data Without Schema”. http://www.devx.com/xml/article/9796. October 30, 2002.

[LM00] A. Laux et L. Martin. XML Update (XUpdate) language. XML:DB working draft, http://www.xmldb.org/xupdate. September 14, 2000

[San98] R. Sandhu. "Role-Based Access Control". Advances in Computers. Vol 48. Academic Press. 1998.

[SF02] A. Stoica and C. Farkas, “Secure XML Views,” In Proc. 16th IFIP WG11.3 Working Conference on Database and Application Security, 2002

[SJ92] R. Sandhu and S. Jajodia. Polyinstantiation for cover stories. In European Symposium on Research in Computer Security. Toulouse, France.1992. Springer Verlag.

[Sur04] Gargi M. Sur, Joachim Hammer, and Jerome Simeon, "UpdateX - An XQuery-Based Language for Processing Updates in XML." International Workshop on Programming Language Technologies for XML (PLAN-X 2004), Venice, Italy, January 2004

[Tat01] I. Tatarinov, Zachary G. Yves, Alon Y. Halevy, Daniel S. Weld. “Updating XML”. In ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA.

[TH98] Marlene Theriault and William Heney. “Oracle Security”. O’Reilly 1998.

[Vid98] Vidur Apparao et al. Document Object Model (DOM) Level 1 XPath Specification. W3C. 1 October 1998,

[XDB] XML DataBase Initiatitive :XML :DB. http://www.xmldb.org.

[Xin] Apache software foundation. Xindice, http://xml.apache.org/xindice.