2004－DBS－134　(I)　(17)
2004／7／13

ビュースキーマを用いたXMLアクセス制御

村田 真　　戸沢 晶彦　　工藤 道治　　羽田 知史
日本IBM（株）東京基礎研
〒242–8502 神奈川県大和市下鶴間 1623-14 LAB-S77
E-mail: {eb91801,atozawa,kudo,satoshih}@jp.ibm.com

XML 文書のためのスキーマから，アクセス制御ポリシーを静的に適用することによってビュースキーマを生成する．本来のスキーマとは異なり，ビュースキーマにはアクセスが許可されている要素や属性しか存在しない．アクセスが禁止されている要素や属性についての余分な情報を隠蔽できるので，ビュースキーマはプログラマの負担を軽減する．

View Schemas for XML Access Control

Makoto MURATA, Akihiko TOZAWA, Michiharu KUDO, Satoshi HADA
IBM Tokyo Research Lab.
1623-14 Shimotsuruma, Yamato, Kanagawa 242–8502, JAPAN
E-mail: {eb91801,atozawa,kudo,satoshih}@jp.ibm.com

A view schema is derived from an original schema for XML documents by enforcing an access control policy statically. Unlike the original schema, the view schema allows only those elements or attributes which are exposed by the policy. Since the view schema hides superfluous information about access-denied elements or attributes, it is more programmer-friendly than the original schema.

1 Introduction

XML [4] has become an active area in database research. XPath [6] and XQuery [3] from the W3C have come to be widely recognized as query languages for XML, and their implementations are actively in progress. In this paper, we are concerned with fine-grained (element- and attribute-level) access control for XML database systems. We believe that access control plays an important role in XML database systems, as it does in relational database systems. Some early experiences [15, 9, 2] with access control for XML documents have been reported already.

Existing languages (e.g. [15, 9]) for XML access control are typically use XPath [6] as a simple and powerful mechanism for handling an infinite number of paths. For example, to deny accesses to `name` elements that are immediately or non-immediately subordinate to `article` elements, it suffices to specify a simple XPath expression `//article//name` as part of an access control policy.

To efficiently determine whether or not an access is granted by an XML access control policy, we have studied static analysis and run-time checking. They are presented in our previous paper [15] and companion paper, respectively [20]. Run-time check is performed when a query engine accesses an element or attribute in an XML database. Meanwhile, static analysis is performed at *compile time* (when a query expression is created rather than each time it is evaluated). It examines access control policies and query expressions as well

as schemas (if present), but does not examine actual databases. Run-time checking is required only when static analysis is unable to grant or deny access requests.

In this paper, we attempt to help the database programmer by *view schemas*. A view schema is derived from an original schema by enforcing an access control policy statically. Unlike the original schema, the view schema allows only those elements or attributes which are exposed by the policy. Since the view schema hides superfluous information about access-denied elements or attributes, it is more programmer-friendly than the original schema.

1.1 Related Works

Fine-grained access control for XML documents has been studied by many researchers [1, 15, 2, 9, 11]. Their access control policies are similar to ours. They all provide run-time checking of access control policies.

It was [10] that introduced view-based access control for XML. Although their work is restricted to DTDs, it derives view schemas (which they call "security views") by eliminating access-denied information from schemas. Our view schemas are inspired by their work. However, our work is not restricted to DTDs; it can handle modern schema languages such as RELAX NG and W3C XML Schema. Another difference between our work and [10] is the way acccess control policies are enforced. [10] uses query rewriting, which is restricted to XPath queries. Meanwhile, our work relies on run-time checking and static analysis, which are applicable to any XPath-based query languages including XQuery.

2 Preliminaries

In this section, we introduce the basics of XML, schema languages, XPath, and XQuery.

```
<record patientId="0003">
  <diagnosis>
    <pathology type="Gastric Cancer">
      Well differentiated adeno carcinoma
    </pathology>
    <comment>This seems correct</comment>
  </diagnosis>
  <chemotherapy>
    <prescription>5-FU 500 mg</prescription>
    <comment>Is this sufficient?</comment>
  </chemotherapy>
  <comment>How was the operation?</comment>
</record>
```

Fig. 1: An XML document example

2.1 XML

An XML document consists of elements, attributes, and text nodes. We hereafter use Σ^E and Σ^A as a set of element names and that of attribute names, respectively. To distinguish between the symbols in these sets, we prepend '@' to symbols in Σ^A.

An XML document representing a medical record is shown in Figure 1. This XML document describes diagnosis and chemotherapy information for a certain patient. For the rest of this paper, we use this document as a motivating example.

2.2 Schema

A *schema* is a description of permissible XML documents. A *schema language* is a computer language for writing schemas. DTD, W3C XML Schema [19], and RELAX NG [7] from OASIS (and now ISO/IEC) are notable examples of schema languages.

We do not use any particular schema language in this paper, but rather use tree regular grammars [8] as a formal model of schemas. Murata et al. [18] have shown that tree regular grammars can model DTD, W3C XML Schema, and RELAX NG.

A *schema* is a 5-tuple $G = (N, \Sigma^E, \Sigma^A, S, P)$, where:

- N is a finite set of *non-terminals*,
- Σ^E is a finite set of *element names*,

- Σ^A is a finite set of *attribute names*,
- S (*start set*) is a subset of $\Sigma^E \times N$,
- P is a set of production rules $X \to r\ \mathcal{A}$, where $X \in N$, r is a regular expression over $\Sigma^E \times N$, and $\mathcal{A}$ is a subset of Σ^A.

Production rules collectively specify permissible element structures. We separate non-terminals and element names, since we want to allow elements of the same name to have different subordinates depending on where these elements occur. Although examples in this paper can be captured without separating non-terminals and element names, W3C XML Schema and RELAX NG require this separation. Unlike the definition in [18], we allow production rules to have a set of permissible attribute names[1] .

For the sake of simplicity, we do not allow schemas to specify constraints on text nodes or attribute values. In the case of DTDs, this restriction amounts to the confusion of `#PCDATA` and `EMPTY`.

A schema for our motivating example is $G_1 = (N_1, \Sigma_1^E, \Sigma_1^A, S_1, P_1)$, where

N_1 = {Record, Diag, Chem, Com, Patho, Presc},
Σ_1^E = {`record`, `diagnosis`, `chemotherapy`, `comment`, `pathology`, `prescription`},
Σ_1^A = {`@patientId`, `@type`},
S_1 = {`record`[Record]},
P_1 = {Record → (`diagnosis`[Diag]*, `chemotherapy`[Chem]*, `comment`[Com]*, `record`[Record]*) {`@patientId`},
Diag → (`pathology`[Patho], `comment`[Com]*) ∅,
Chem → (`prescription`[Presc]*, `comment`[Com]*) ∅,
Com → ϵ ∅, Patho → ϵ {`@type`},
Presc → ϵ ∅}.

An equivalent DTD is shown below.

```
<!ELEMENT record       (diagnosis*,
                        chemotherapy*,
                        comment*,record*)>
<!ATTLIST record       patientID CDATA #REQUIRED>
<!ELEMENT diagnosis    (pathology,comment*)>
<!ELEMENT chemotherapy (prescription*,comment*)>
<!ELEMENT comment      (#PCDATA)>
<!ELEMENT pathology    (#PCDATA)>
<!ATTLIST pathology     type CDATA #REQUIRED>
<!ELEMENT prescription (#PCDATA)>
```

[1] RELAX NG provides a more sophisticated mechanism for handling attributes [14].

2.3 XPath

XPath is a mechanism for locating certain elements or attributes in XML documents. XPath is widely recognized in the industry and is used by XSLT [5] and XQuery.

XPath uses *axes* for representing the structural relationships between nodes. For example, the above example can be captured by the XPath expression `//p//a`, where `//` is an axis called "descendant-or-self". Although XPath provides many axes, we consider only three of them, namely "descendant-or-self" (//), "child" (/), and "attribute" (@) in this paper. Namespaces and wildcards are outside the scope of this paper, although our framework can easily handle them.

2.4 XQuery

XQuery is an XML query language developed by W3C. The following query lists the pathology-comment pairs for the Gastric Cancer.

```
<TreatmentAnalysis>
{
  for $r in document("medical_record")/record
  where $r/diagnosis/pathology/@type
          = "Gastric Cancer"
  return
    $r/diagnosis/pathology, $r//comment
}
</TreatmentAnalysis>
```

3 Access Control for XML Documents

In this paper, access control for XML documents means element- and attribute-level access control for a certain XML instance. Each element and attribute is handled as a unit resource to which access is controlled by the corresponding access control policies. In the following sections, we use the term *node-level* access control when there is no need to separate the *element-level* access control from the *attribute-level* access control.

3.1 Syntax of Access Control Policy

The access control policy consists of a set of access control rules. Each rule consists of an *object* (a target node), a *subject* (a human user or a user process), an *action*, and a *permission* (grant or denial) meaning that the *subject* is (or is not) allowed to perform the *action* on the *object*. The subject value is specified using a user ID, a role or a group name but is not limited to these. For the object value, we use an XPath expression. The action value can be either *read, update, create,* or *delete,* but we deal only with the *read* action in this paper because the current XQuery does not support other actions. The following is the syntax of our access control policy[2]:

```
(Subject, +/-Action, Object)
```

The subject has a prefix indicating the type of the subject such as role and group. "+" means grant access and "-" means deny access. In this paper, we sometimes omit specifying the subject if the subject is identical with the other rules.

Suppose there are three access control rules for the document described in Section 2.1:

Role: *Doctor*
`+R, /record`

Role: *Intern*
`+R, /record`
`-R, //comment`

Each rule is categorized by the role of the requesting subject. The first rule says that "*Doctor* can read `record` elements". The second rule says that "*Intern* can read `record` elements". The third rule says that "*Intern* cannot read any `comment` elements" because `comment` nodes may include confidential information and should be hidden from access by *Intern*. Please refer to Section 3.2 for more precise semantics.

[2]The syntax of the policy can be represented in a standardized way using XACML [12] but we use our syntax for simplicity.

3.1.1 Using XPath for XML Access Control

Many reports [15, 9, 2, 11] on the node-level access control for XML documents use XPath to locate the target nodes in the XML documents. XPath provides a sufficient number of ways to refer to the smallest unit of an XML document structure such as an element, an attribute, a text node, or a comment node. Therefore it allows a policy writer to write a policy in a flexible manner (e.g. grant access to a certain element but deny access to the enclosing attributes). In this paper, for simplicity, we limit target nodes of the policy to only the elements and attributes. We assume that other nodes such as text and comment nodes are governed by the policy associated with the parent element.

3.2 Semantics of Access Control Policy

Access control policies in general should satisfy the following requirements: *succinctness*, *least privilege*, and *soundness*. Succinctness means that the policy semantics should provide a way to specify a smaller number of rules rather than to specify rules on every single node in the document. Least privilege means that the policy should grant the minimum privilege to the requesting subject. Soundness means that the policy evaluation must always generate either a grant or a denial decision in response to any access request.

To satisfy the above requirements, the semantics of our access control policies are defined as follows:

1. An access control rule with `+R` or `-R` (capital letter) propagates downward through the XML document structure. An access control rule with `+r` or `-r` (small letter) does not propagate and just describes the rule on the specified node.
2. A rule with denial permission for a node overrules any rules with grant permission for the same node.
3. If no rule is associated with a certain node, the default denial permission "-" is applied to that

```
<record patientID="0003">
  <diagnosis>
    <pathology type="Gastric Cancer">
      Well differentiated adeno carcinoma
    </pathology>
  </diagnosis>
  <chemotherapy>
    <prescription>5-FU 500 mg</prescription>
  </chemotherapy>
</record>
```

Fig. 2: The XML document that *Intern* can see

node.

Now we informally describe an algorithm to generate an access decision according to the above definitions. First, the algorithm gathers every grant rule with `+r` and marks "`+`" on the target nodes referred to by the XPath expression. If the node type is an element, the algorithm marks "`+`" on immediate children nodes (e.g. a text and comment nodes) except for the attributes and the elements. It also marks a "`+`" on all the descendant nodes if the action is `R`. Next, the algorithm gathers the remaining rules (denial rules) and marks "`-`" on the target nodes in the same way. The "`-`" mark overwrites the "`+`" mark if any. Finally, the algorithm marks "`-`" on every node that is not yet marked. This operation is performed for each subject and action independently.

For example, the access control policy in Section 3.1 is interpreted as follows: The first rule marks the entire tree with "`+`" and therefore *Doctor* is allowed to read every node (including attributes and text nodes) equal to or below any `record` element. The second and third rules are policies for *Intern*. The second rule marks the entire tree with "`+`" as the first rule does and the third rule marks `comment` elements and subordinate text nodes with "`-`", which overwrites `+` marks. Thus, three `comment` elements and text nodes are determined as "access denied". The XML document that *Intern* can see is shown in Figure 2.

A rule that uses `+R` or `-R` can be converted to the rule with `+r` or `-r`. For example, `(`*Sbj*`,+R,/a)` is semantically equivalent to a set of three rules: `(`*Sbj*`,+r,/a)`, `(`*Sbj*`,+r,/a//*)` and `(`*Sbj* `,+r,/a//@*)`. Thus, `+R` and `-R` are technically syntactic sugar, but enable a more succinct representation of the policy specification.

3.2.1 Denial downward consistency

We require that access control policies satisfy *denial downward consistency*, which is specific to XML access control. Although view schemas can be constructed even when this requirement is *not* satsified, we feel that it ensures simpicity and consistency of access control policies.

Denial downward consistency requires that whenever a policy denies an access to an element, it must also deny the access to its subordinate elements and attributes. In other words, whenever access to a node is allowed, access to all the ancestor elements must be allowed as well. We impose this requirement since we believe that elements or attributes isolated from their ancestor elements are meaningless. For example, if an element or attribute specifies a relative URI, its interpretation depends on the attribute `xml:base` [16] specified in the ancestor elements. Another advantage of denial downward consistency is that it makes implementation of runtime policy evaluations easier.

4 View Schemas

Recall that a schema defines the set of permissible XML documents in terms of elements, attributes, and their structural relationships. When access control is present, however, the elements or attributes permitted by the schemas are not always accessible to the database programmer. In other words, the exposed documents are *different* from the documents permitted by the schema. Such a schema is not only confusing but may also allow malicious programmers to guess hidden information [10].

To overcome this problem, we derive a *view schema* from an input schema. A view schema is equivalent to the input schema except that it does not allow those elements and attributes which are hidden by the policy.

4.1 Automata and XPath expressions

In preparation, we introduce automata and show how we capture XPath expressions by using automata.

A *non-deterministic finite state automaton* (NFA) M is a tuple $(\Omega, Q, Q^{\text{init}}, Q^{\text{fin}}, \delta)$, where Ω is an alphabet, Q is a finite set of *states*, Q^{init} (a subset of Q) is a set of *initial states*, Q^{fin} (a subset of Q) is a set of *final states*, and δ is a *transition function* from $Q \times \Omega$ to the power set of Q [13]. The set of strings accepted by M is denoted $L(M)$.

Recall that we have allowed only three axes of XPath (see Section 2.3). This restriction allows us to capture XPath expressions with automata. As long as an XPath expression contains no predicates, we can easily construct an automaton from it. We first create a regular expression by replacing "/" and "//" with "$\cdot$" and "$\cdot\,(\Sigma^E)^*\cdot$", respectively, where "$\cdot$" denotes the concatenation of two regular sets, and then create an automaton from this regular expression. The constructed automaton accepts a path if and only if it matches the XPath expression.

When an XPath expression r contains predicates, we cannot capture its semantics exactly by using an automaton. However, we can still approximate r by constructing an *over-estimation* $\overline{r}$ and an *under-estimation* $\underline{r}$ and then construct automata for them. To construct $\overline{r}$, we assume that predicates are always satisfied. Meanwhile, to construct $\underline{r}$, we assume that the predicates occurring in r are never satisfied. See our previous paper [15] for further details of over- and under-estimation.

4.2 Creating access control automata

An access control policy consists of rules, each of which applies to some roles. For each role, we create an *access control automaton*. This automaton captures the set of those paths to elements or attributes which are exposed by the access control policy.

In preparation, we replace +R and -R rules with +r and -r rules, respectively (see Section 3.2). Let $r_1, ..., r_m$ be the XPath expressions occurring in the grant rules (`+r`), and let $r'_1, ..., r'_n$ be the XPath expressions occurring in the denial rules (`-r`). For simplicity, we assume that none of $r_1, ..., r_m, r'_1, ..., r'_n$ contain predicates. (This restriction can be lifted by using *under-estimation* for $r_1, ..., r_m$ and *over-estimation* for $r'_1, ..., r'_n$.)

Recall that we interpret the policy according to the "denial-takes-precedence" principle. M^Γ accepts those paths which are allowed by one of $r_1, ..., r_m$ but are denied by any of $r'_1, ..., r'_n$. Formally,

$$\begin{aligned} L(M^\Gamma) \quad = \quad & (L(M[r_1]) \cup \cdots \cup L(M[r_m])) \\ & \setminus \quad (L(M[r'_1]) \cup \cdots \cup L(M[r'_n])) \end{aligned}$$

where $\Sigma = \Sigma^E \cup \Sigma^A$ and "$\setminus$" denotes the set difference. We can construct M^Γ by applying Boolean operations to $M[r_1], ..., M[r_m], M[r'_1], ..., M[r'_n]$.

We demonstrate this construction for the access control policy in Section 3.1. For the role *Intern*, this policy contains a grant rule and a denial rule, both of which propagate downward. The grant rule contains an XPath `/record`, while the denial rule contains an XPath `//comment`. Thus,

$$\begin{aligned} L(M^\Gamma) \quad = \quad & \{\texttt{record}\} \cdot (\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\}) \setminus \\ & (\Sigma^E)^* \cdot \{\texttt{comment}\} \cdot (\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\}) \end{aligned}$$

4.3 Constructing view schemas

The key idea for constructing view queries is to simulate the execution of the access control automaton as well as the derivation of the schema[3]. This is done by using (non-terminal, state) pairs as "non-terminals", where non-terminals are taken from the schema and states are borrowed from the access control automaton. The key observation is that a "non-terminal" comprising a non-final state is used only for deriving access-denied elements or attributes. A view schema can then be obtained by deleting such non-terminals.

[3] This approach is a special case of the schema transformation shown by the first author in [17]

Formally, the view schema G^Γ is defined as follows. Let the access control automaton M^Γ be a deterministic automaton $(\Sigma^A \cup \Sigma^E, Q, q_0, \delta, Q_F)$ where $q_0 \in Q$, $Q_F \subseteq Q$, and δ is a function from $Q \times (\Sigma^A \cup \Sigma^E)$ to Q. We first construct the product of G and M^Γ as below:

1. The set of non-terminals is the cross product of the non-terminal set N of G and the state set Q of M^Γ.
2. The underlying alphabets (namely, Σ^E and Σ^A) for elements and attributes are borrowed from G, but terminals not appearing in any production rules or start sets are deleted.
3. The start set is constructed from S as well as q_0 and δ; for every $e[x]$ in S, we introduce $e[(x, \delta(q_0, e))]$, where $\delta(q_0, e)$ simulates the execution of M^Γ from q_0 via e.
4. The set of production rules is constructed from P as well as δ. For every production rule $x \to r\ \mathcal{A}$ of G and every state q in M^Γ, a production rule is introduced. Its left-hand side is (x, q) and its right-hand side is obtained by replacing each $e[x']$ in r with $e[(x', \delta(q, e))]$, which simulates the execution of M^Γ from q via e.
5. Element or attribute names not appearing in any production rules are removed.

To create a view schema from this product, we only have to make the access-denied elements and attributes invisible. That is, we remove non-terminals containing non-final states, remove attributes leading M^Γ to non-final states, and replace $e[(x_1, q_1)]$ on the right-hand side with an empty sequence where q_1 is a non-final state.

To summarize, a view schema is

$$G^\Gamma = (N \times Q, (\Sigma^E)', (\Sigma^A)', S', P'),$$

where:

$$\begin{aligned}
S' &= \{e[(x,q)] \mid e[x] \in S, q = \delta(q_0, e), \\
&\qquad q \in Q_F\}, \\
P' &= \{(x,q) \to \phi^q(r)\ \mathcal{A}' \mid x \to r\ \mathcal{A} \in P, \\
&\qquad q \in Q_F, \\
&\qquad \mathcal{A}' = \{a \in \mathcal{A} \mid \delta(q, a) \in Q_F\}, \\
(\Sigma^E)' &= \{e \in \Sigma^E \mid e \text{ occurrs in } S' \text{ or } P'\}, \\
(\Sigma^A)' &= \{a \in \Sigma^A \mid a \text{ occurrs in } P'\}.
\end{aligned}$$

where ϕ^q is a homomorphism from $(\Sigma^E \times N)^*$ to $(\Sigma^E \times N \times Q)^*$ defined as

$$\phi^q(e'[x']) = \begin{cases} e[(x', \delta(q, e'))] & (\delta(q, e') \in Q_F) \\ \epsilon & (\text{otherwise}) \end{cases}$$

Note that our view schemas hide elements as well as their subordinate elements and attributes. This is because we believe in the denial downward consistency. However, should we drop denial downward consistency, we can still make view schemas by *renaming* access-denied elements (as in [10]) rather than by deleting them.

4.4 Example

A view schema for our motivating example is $G_1 = (N_1, \Sigma_1^E, \Sigma_1^A, S_1, P_1)$, where

$$\begin{aligned}
N_1 &= \{\text{Record}_1, \text{Diag}_1, \text{Chem}_1, \text{Patho}_1, \text{Presc}_1\}, \\
\Sigma_1^E &= \{\texttt{record}, \texttt{diagnosis}, \texttt{chemotherapy}, \\
&\quad \texttt{pathology}, \texttt{prescription}\}, \\
\Sigma_1^A &= \{\texttt{@patientId}, \texttt{@type}\}, \\
S_1 &= \{\texttt{record}[\text{Record}_1]\}, \\
P_1 &= \{\text{Record}_1 \to (\texttt{diagnosis}[\text{Diag}_1]^*, \\
&\qquad \texttt{chemotherapy}[\text{Chem}_1]^*, \\
&\qquad \texttt{record}[\text{Record}_1]^*) \\
&\qquad \{\texttt{@patientId}\}, \\
&\quad \text{Diag}_1 \to (\texttt{pathology}[\text{Patho}_1]^*)\ \emptyset, \\
&\quad \text{Chem}_1 \to (\texttt{prescription}[\text{Presc}_1]^*)\ \emptyset, \\
&\quad \text{Patho}_1 \to \epsilon\ \{\texttt{@type}\},\ \ \text{Presc}_1 \to \epsilon\ \emptyset\}.
\end{aligned}$$

Note that `comment` elements do not exist in this view schema, since our access control policy (shown in Section 3) has a denial rule (`-R, //comment`).

5 Concluding Remarks

We presented a formal construction of view schemas from input schemas and access control policies. We plan to implement this construction using modern schema languages. We believe that the construction is not difficult when we use RELAX NG, but it becomes much harder when we use XML Schema. Another challenge is to generate comprehensible schemas.

References

[1] E. Bertino, S. Castano, E. Ferrari, and M. Mesiti. Controlled access and dissemination of XML documents. In *The 2nd Workshop on Web Information and Data Management*, pp. 22–27. ACM, November 1999.

[2] E. Bertino, S. Castano, E. Ferrari, and M. Mesiti. Author-X: a Java-based system for XML data protection. In *14th IFIP Workshop on Database Security*, Vol. 201 of *IFIP Conference Proceedings*, pp. 15–26. Kluwer, 2001.

[3] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML query language. W3C working draft 12 November 2003. `http://www.w3.org/TR/xquery/`, November 2003.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0. W3C Recommendation. `http://www.w3.org/TR/REC-xml`, Feburary 2004.

[5] J. Clark. XML Transformations (XSLT) version 1.0. W3C Recommendation, November 1999. http://www.w3.org/TR/xslt.

[6] J. Clark and S. DeRose. XML Path Language (XPath) version 1.0. W3C Recommendation. `http://www.w3.org/TR/xpath`, Nov 1999.

[7] J. Clark and M. Murata. RELAX NG specification. OASIS Committee Specification, December 2001. `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`.

[8] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at `http://www.grappa.univ-lille3.fr/tata`, 1997. release October, 1st 2002.

[9] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Securing XML documents. In *Proceedings of the 7th International Conference on Extending Database Technology (EDBT)*, Vol. 1777 of *Lecture Notes in Computer Science*, pp. 121–135, Konstanz, 2000. Springer.

[10] W. Fan, C. Y. Chan, and M. N. Garofalakis. Secure XML querying with security views. In *Proceedings of the 23rd SIGMOD International Conference on Management of Data, to appear*. ACM, 2004.

[11] A. Gabillon and E. Bruno. Regulating access to XML documents. In *Proceedings of the 15th IFIP WG 11.3 Working Conference on Database Security*, pp. 299–314, July 2001.

[12] S. Godik and T. Moses (Eds). Extensible access control markup language (XACML) version 1.0. OASIS Standard `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`, Feb. 2003.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[14] H. Hosoya and M. Murata. Validation and boolean operations for attribute-element constraints. In *Programming Languages Technologies for XML (PLAN-X)*, October 2002.

[15] M. Kudo and S. Hada. XML document security based on provisional authorization. In *Proceedings of the 7th Conference on Computer and Communications Security*, pp. 87–96. ACM, November 2000.

[16] J. Marsh. XML Base. W3C Recommendation, June 2001. http://www.w3.org/TR/2001/REC-xmlbase-20010627/.

[17] M. Murata. Extended path expressions for XML. In *Proceedings of the 20th Symposium on Principles of database systems*, pp. 126–137, Santa Barbara, May 2001.

[18] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, August 2001.

[19] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema part 1: Structures. W3C Recommendation, May 2001. http://www.w3.org/TR/xmlschema-1/.

[20] 戚, 工藤. パステーブルを用いた XML アクセス制御. 夏のデータベースワークショップ (DBWS), July 2004.