An Access Control Method Based on the Prefix Labeling Scheme for XML Repositories

Shohei Yokoyama, Manabu Ohta, Kaoru Katayama and Hiroshi Ishikawa

Graduate School of Engineering
Tokyo Metropolitan University
1-1 Minamiosawa Hachioji Tokyo, Japan

{shohei,ohta,katayama,ishikawa}@hikendbs.eei.metro-u.ac.jp

Abstract

This paper describes an access control method of the XML repository system, SAXOPHONE, which was implemented at Tokyo Metropolitan University. The main feature of our research is a novel account identifier that is based on the prefix-labeling scheme to realize a hierarchical authorization. SAXOPHONE uses relational databases for XML document storage. Using it, any valid or well-formed XML document is decomposed into events of the SAX parser and is then stored into relational tables using a fixed scheme. Consequently, users can handle the system as a normal SAX parser. This study also illustrates how to realize an access control method of XML documents on relational databases using the account identifier.

Keywords: XML repository, XML database, access control, SAX parser, relational database.

1 Introduction

Recently, interest has been growing in XML repositories. The XML document, in its infancy, was used for data-exchange. However, it has come to be used also for data-accumulation. Therefore, storing and handling massive XML documents is an important issue.

The XML format was originally a text file. When XML is considered for data-accumulation, we think it helpful that the XML data handling system is augmented with some concepts of relational databases because relational databases can process huge amounts of data. In addition, they provide many valuable functions to handle such data, such as transaction, data backup, replication and access control.

Use of relational databases as XML repositories has been studied extensively in recent years. We have also developed SAXOPHONE, which is an XML repository using relational databases.

This paper specifically describes access control of XML documents. It describes how to apply the access control function of relational databases to XML repositories. Our requirements for this system are the following: (1) any valid or well-formed XML document can be stored; (2) users can read XML documents in the repository using a SAX (Simple API for XML) parser; (3) the XML documents consist of nodes that users can read; and (4) any relational database user account is valid for simultaneous use of the XML repository.

The salient feature of this research is that authorization for the XML repository is managed by the relational database management system (DBMS) without any necessary additional programs. The XML repository uses relational databases. For that reason, it is essential to synchronize the account of the XML repository and the relational database because the system should translate XML queries into SQL statements. However, if the system has an independent authorization mechanism, and uses one account to query to the relational database regardless of users, then every user who is using the account can access all data in the relational database, including the XML repository. This fact may present problems in terms of security.

Our XML repository, SAXOPHONE, is a simple wrapper program for relational databases. It can be used with available database management systems. Section 3 presents the overall SAXOPHONE architecture.

We propose a novel account identifier to realize a hierarchical authorization. The identifier is based on the prefix-labeling scheme, which is described in Section 4. We present the hierarchical authorization in Section 5.

Given an account identifier, we need access control implementation with the relational database. We describe it in the Section 6.

First, Section 2 presents the context for our work and related studies.

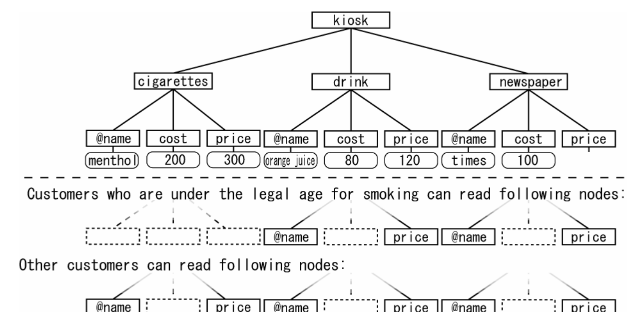


Figure 1: An example of XML tree and its access policies

Copyright (c) 2005, Australian Computer Society, Inc. This paper appeared at the 16th Australasian Database Conference, University of Newcastle, Newcastle, Australia. Conferences in Research and Practice in Information Technology, Vol. 39. H.E. Williams and G. Dobbie, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

2 Outline and related works

An XML document has a tree structure and comprises elements, attributes and text nodes.

Example 1. *Figure 1 shows a simple XML tree. The price list of* kiosk *can be* drink*,* newspaper *and* cigarettes*. Each of them has a* name *as identification, a* cost *and a* price*.*

Example 2. *Figure 1 also shows an instance of access control. The XML repository must be accessible by multiple users. The shop owner should be able to read all XML document data. However, the cost should be hidden from customers. Moreover, all information regarding the cigarettes must be hidden from customers who are under the legal age for smoking.*

The Extensible Access Control Markup Language (XACML; OASIS 2003) is used to express access control policies for entire documents or individual elements. This standard only addresses policy prescriptions. For that reason, it remains unclear how access control can be implemented for XML repositories.

Kudo and Hada (2000), who described implementation of e-business Web applications, proposed a policy-based access control mechanism. However, XML repositories are not specified. If the XML repository, irrespective of a relational database or a native XML database, has an original access control method, the applications or users must manage two different access controls. In our approach, user authorization of an XML repository is synchronized with the relational database.

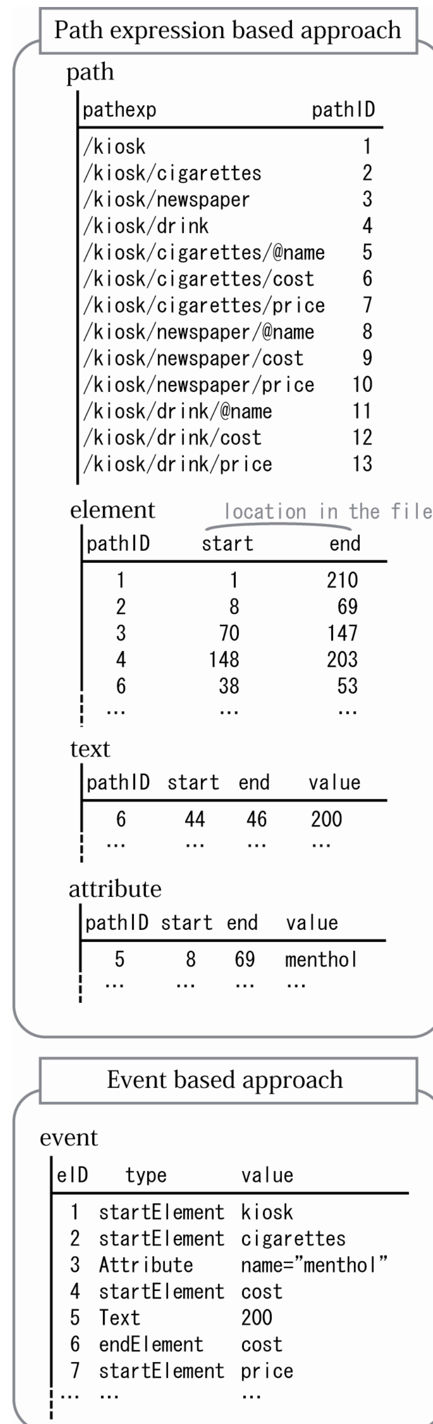
Our XML repository, SAXOPHONE, uses relational databases for following reasons:

1. The relational database is a de facto standard for data accumulation. Therefore, a large amount of non-XML data has already been stored in them. Storing XML data in the same kind of database is useful.
2. Most relational database management systems have transaction mechanisms, an access control method and a processing mechanism. They have been developed for a quarter of a century. In contrast, XML originated in 1998. Therefore, it is pragmatic to cope with relational databases.

SilkRoute (Fernandez 2001) and XPERANTO (Carey 2000) are proposals for efficient publication of relational data as XML. Nevertheless, ordered XML documents are unsupported because relational data are not ordered.

Most closely related to SAXOPHONE is xRel (Yoshikawa 2001). Any valid or well-formed XML documents could be stored to a fixed scheme of xRel. However, the most important difference between the xRel and SAXOPHONE is that xRel is a path-expression-based approach, whereas SAXOPHONE is an event based approach.

Example 3 *Figure 2 shows an example of the path-expression-based approach and the event based approach. The price list in Fig. 1 is divided and stored into the XML repository using relational databases.*



Path expression based approach

path

pathexp	pathID
/kiosk	1
/kiosk/cigarettes	2
/kiosk/newspaper	3
/kiosk/drink	4
/kiosk/cigarettes/@name	5
/kiosk/cigarettes/cost	6
/kiosk/cigarettes/price	7
/kiosk/newspaper/@name	8
/kiosk/newspaper/cost	9
/kiosk/newspaper/price	10
/kiosk/drink/@name	11
/kiosk/drink/cost	12
/kiosk/drink/price	13

element (location in the file)

pathID	start	end
1	1	210
2	8	69
3	70	147
4	148	203
6	38	53
…	…	…

text

pathID	start	end	value
6	44	46	200
…	…	…	…

attribute

pathID	start	end	value
5	8	69	menthol
…	…	…	…

Event based approach

event

eID	type	value
1	startElement	kiosk
2	startElement	cigarettes
3	Attribute	name="menthol"
4	startElement	cost
5	Text	200
6	endElement	cost
7	startElement	price
…	…	…

Figure 2: A storage example of XML documents

3 SAXOPHONE: an XML repository

This section describes SAXOPHONE and its mechanism. SAXOPHONE is an event-based approach to store XML documents into relational databases using a SAX parser. First of all, we illustrate the SAX parser as a general XML parser.

3.1 SAX parser

SAX (Simple API for XML) is a *de facto* standard XML parser. Versions exist for several programming language environments. There are two major types of XML APIs:

1. **Tree-based APIs**: These APIs map XML documents into an in-memory tree structure and thereby allow applications to navigate the tree. The Document Object Model (DOM) working group at the World-Wide Web Consortium (W3C) maintains a recommended tree-based API for XML and HTML documents. These APIs are useful for a wide range of applications, but they normally put a great strain on system resources, especially if the document is large.
2. **Event based APIs**: Application implements handlers to deal with the different events that are generated by the XML parser. The application does not usually build an internal tree. These APIs provide a simpler, lower-level access to an XML document: users can parse documents that are much larger than the available system memory.

Example 4 *Figure 3 shows that the SAX parser reads input XML stream of Fig. 1 and generates various parsing events that an application can handle.*

This paper addresses four types of events: startElement, Attribute, Text and endElement. Herein, we ignore other events such as comment node events and error events.

Strictly speaking, the Attribute event is not an event of SAX APIs because it is an argument of the startElement event. Nevertheless, we regard it as an event in this paper.

An event handler can be implemented by SAX applications. The handler receives notification of basic document-related events such as startElement, Attribute, Text and endElement. A simple Java skeleton code of the handler is given as follows.

```
01:   class MySAXHandler extends DefaultHandler {
02:     public void startElement(java.lang.String uri,
03:                     java.lang.String localName,
04:                     java.lang.String xmlTagName,
05:                     Attributes attributesOfThisElement)
06:       //startElement and Attribute event
07:     }
08:     public void endElement(java.lang.String uri,
09:                     java.lang.String localName,
10:                     java.lang.String xmlTagName)
11:                 throws SAXException {
12:       //endElement
13:     }
14:     public void characters(char[ ] ch,
15:                     int start,
16:                     int length)
17:                 throws SAXException {
18:       //Text event
19:     }
20:   }
```

This class has three methods: a startElement method receives the startElement event and provides the attributes of this element; an endElement method receives an endElement event; a character method receives a Text event.

An important point to emphasize is that SAXOPHONE has a SAX interface. For that reason, applications can handle XML documents in SAXOPHONE via the SAX event handler. Consequently, all SAX applications can adopt SAXOPHONE for storage with no modification.

3.2 Schema of SAXOPHONE

A SAX parser generates a sequence of events that corresponds to the hierarchical structure of the XML instance. Each event consists of an event type and a property. Therefore, the sequence data are considered to be a relational model.

Example 5 *Figure 4 shows relational tables that SAXOPHONE constructs. They hold the event sequence of Fig. 3.*

The basic SAXOPHONE scheme is fixed; it consists of the following two relational schema:

event (rID, eID, type, property)

resource (rID, URI)

The database attributes rID, URI, eID, type, and

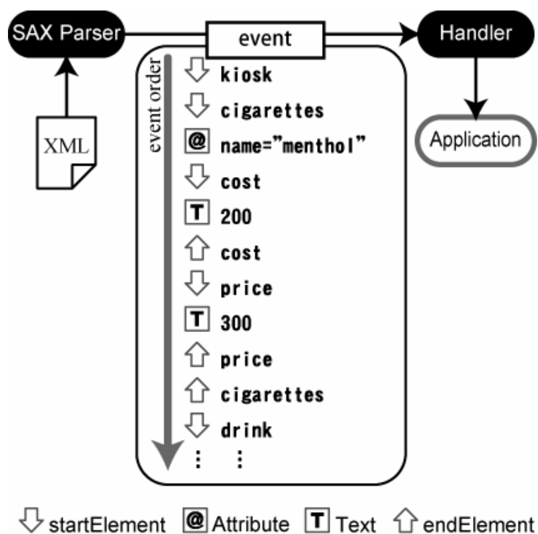


Figure 3: An example of Parsing Events

event

rID	eID	type	property
1	1	1	kiosk
1	2	1	cigarettes
1	3	2	name="menthol"
1	4	1	cost
1	5	3	200
1	6	4	cost
1	7	1	price
…	…	…	…

resource

rID	URI
1	http://hostname/exaple.xml
2	http://hostname/test.xml
…	…

Figure 4: An example of data in the SAXOPHONE

property represent the XML file identifier, the URI of the XML file, the event serial number, the event type identifier, and property of the event, respectively. The set of the database attributes rID and eID is a primary key of the relation event. The database attribute rID of the relation event is a foreign key to the relation resource. The relation resource's attribute rID, which is a primary key, and the URI must be unique. No database attributes allow NULL values.

3.3 Reconstruction of a SAX event sequence

Reconstruction of a SAX event sequence must contain a sort operation because the relational data are not ordered. Creating an index on the database attribute eID is important. The following SQL query reconstructs the event sequence when given a URI of *U*.

```
01:  SELECT      type, property
02:  FROM        event
03:  WHERE       rID = (SELECT   rID
04:                     FROM     resource
05:                     WHERE    URI = U)
06:  ORDER BY    eID
```

3.4 SAX interface of SAXOPHONE

In SAXOPHONE, the relational database is hidden from applications. Users are able to reach SAXOPHONE via the SAX interface. Consequently, both users and developers can access the XML document without concern about storage format. Codes for reading from XML files and for querying SAXOPHONE are identical:

```
01:  //Obtain an instance of SAXParserFactory.
02:  SAXParserFactory spf =
     SAXParserFactory.newInstance();
03:  //Obtain an instance of a parser from the factory.
04:  SAXParser sp = spf.newSAXParser();
05:  //Parse the document.
06:  sp.parse(URI, new mySaxEventHander());
```

SAXOPHONE uses existing SAX applications with no modification. Figure 5 illustrates that transparent access to SAXOPHONE.

4 Account identifier

This section describes a SAXOPHONE account identifier. The identifier expresses hierarchical authorization. By this approach, access policies of user accounts are inheritable of other access policies. To express that relation, the account identifier is based on a labeling scheme for trees.

4.1 Labeling schemes for trees

Several labeling schemes have been proposed (Cohen 2002, Li 2001, Abiteboul 2001). Li et al. proposed a scheme based on a *k*-ary tree – a tree with no more than *k* children for each node. It is inefficient when *k* is large.

Abiteboul et al. proposed another scheme based on prefix labeling. By this approach, node *A*, which has the label *L* is given, and node *B*, which has a label *M* that is a prefix of *L* is an ancestor of *A*. In addition, if node *C* has label *N* and label *L* is a prefix of *N*, then node *C* is a descendant of *A*. However, in this approach, the labels become long because of skew in the tree. Moreover, the labels consist of binary sequences. For that reason, it is difficult to use it to manipulate relational databases, which are designed for text data.

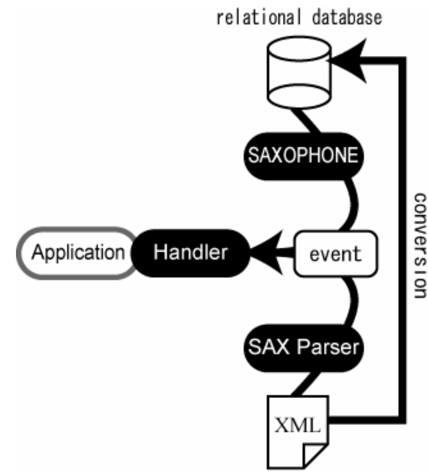


Figure 5: A transparent access to relational database

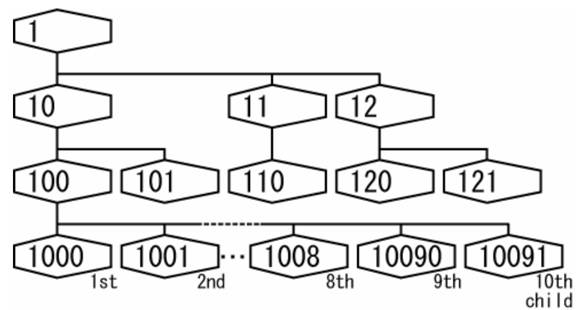


Figure 6: An example of our account identifiers

4.2 Our labeling scheme

The proposed account identifier is also based on prefix labeling. The account identifier is an integer. The ROOT node is labeled '1'. Any label B_x that is a *x*th child of the node has the label *A*, as given by this equation.

$$B_x = 10^{\left\lfloor \frac{x+9}{9} \right\rfloor}(A+1) + (x \bmod 9) - 10$$

Figure 6 shows labels in our scheme. For example, if A is 100, then B_8 and B_9 are calculated as the following.

$$B_8 = 10^1(100+1) + 8 - 10 = 1008$$

$$B_9 = 10^2(100+1) + 0 - 10 = 10090$$

It is noteworthy that the 9 symbol is used as an overflow flag. Similarly, B_{18} is the following.

$$B_{18} = 10^3(100+1) + 0 - 10 = 100990$$

In this prefix labeling scheme, the ancestors of the node 100990 are 10099, 1009, 100, 10, 1, and itself. However, labels ending with 9 do not exist. Therefore, 100, 10, 1, and itself are the ancestors.

4.3 Ancestor query

An SQL statement can represent the ancestor query. If the family tree of Table 1 is given, the query to find my ancestors is as follows:

ID	relative
1	Grandmother
10	Uncle
11	Mother
111	Me

Table 1: An account identifier example of a family tree

```
01:  SELECT   ID,relative
02:  FROM     table1
03:  WHERE    1 = CHARINDEX(ID,'111')
```

The CHARINDEX(ID,'111') is a function of Microsoft SQL server 2000; it returns the starting position of the specified expression in a character string. If the CHARINDEX(*substring*,*string*) returns 1, then the *substring* must be the prefix of the *string*. Most DBMSs have a similar function, e.g., the postgreSQL has position(*substring* in *string*).

Table 2 is returned as a result of this query.

ID	relative
1	Grandmother
11	Mother
111	Me

Table 2: Result set of the ancestor query

Herein, we also consider the nearest ancestor query because an account inherits a policy from the nearest ancestor. We specifically address the length of the account identifiers to find the nearest ancestor. That is, the account that has the longest account identifier is the nearest ancestor. For example, the nearest ancestor of the account identifier 1110 is 'Me'.

5 Hierarchy authorization

The structure of user accounts for our system is a tree whose nodes have the account identifier. We call the authorization mechanism using the account identifier the hierarchical authorization. This section describes the hierarchical authorization in detail.

Example 6 *Figure 7 shows a tree that contains user accounts for Fig. 1. Access policies for the SAXOPHONE are given with these account identifiers.*

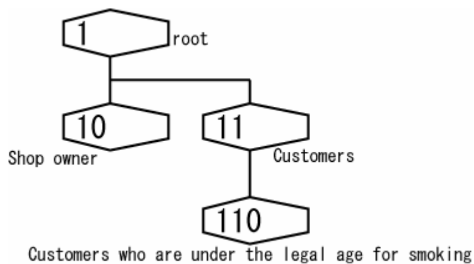


Figure 7: An instance of the account tree.

5.1 Access policy

Events described in this paper are shown in Table 3.

Event Type ID	Event
0	NULL event
1	Start Element
2	Attribute
3	Text
4	End Element

Table 3: Events and its IDs of SAXOPHONE

Events consist of SAX events and a NULL event. A NULL event is an original SAXOPHONE event that means denying access.

A root account can read all XML document events. Other accounts are limited when necessary, e.g., as when a certain event has an account identifier 1 as an Attribute event and, simultaneously, the event also has an account identifier 110 as a NULL event. All users, except those with account identifiers starting with 110, can read the Attribute event.

5.2 Relational schema

In SAXOPHONE, a fixed database scheme is used to store the structure of all XML documents. Section 3.2 shows the basic relation scheme of SAXOPHONE. In addition, the relational tables must contain account identifiers for access control. The augmented scheme is shown in Fig. 8.

The database attribute aID and depth are appended to the basic SAXOPHONE scheme. The set of the database attributes rID, eID and aID is a primary key of the relation event. The aID contains the account identifier that is explained in Section 4.

The database attribute depth represents the account identifier length. A depth is computed automatically from aID. Nevertheless, to explain, we present here the aID as a database attribute in this paper.

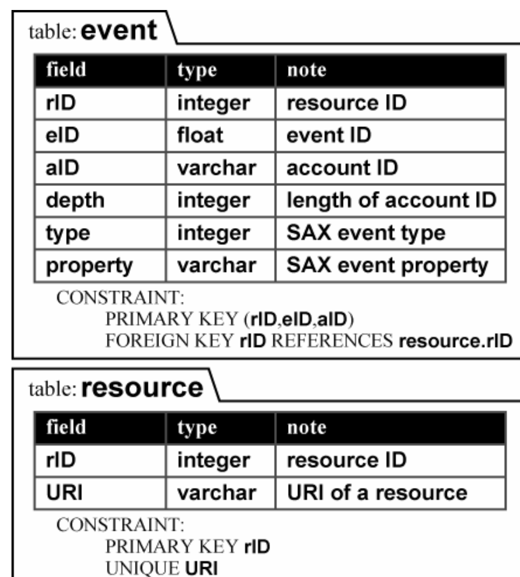


table: **event**

field	type	note
rID	integer	resource ID
eID	float	event ID
aID	varchar	account ID
depth	integer	length of account ID
type	integer	SAX event type
property	varchar	SAX event property

CONSTRAINT:
PRIMARY KEY (**rID**,**eID**,**aID**)
FOREIGN KEY **rID** REFERENCES **resource.rID**

table: **resource**

field	type	note
rID	integer	resource ID
URI	varchar	URI of a resource

CONSTRAINT:
PRIMARY KEY **rID**
UNIQUE **URI**

Figure 8: Relational Schema of SAXOPHONE

5.3 Event sequence representation

In SAXOPHONE, both data and structure of XML documents are stored in a relational scheme. Therefore, an event sequence is expressed by a result of SQL query.

Example 7 *Figure 9 shows a part of relational data for the access-controlled XML document of Fig. 1.*

This event sequence representation algorithm proceeds as follows. **STEP 1**: Resolve rID from file path. **STEP 2:** Find nearest ancestor for every event. **STEP 3:** Represent event sequence. If the URI *@URI* is given, STEP 1 is realized by the following query.

```
01:  SELECT    rID
02:  FROM      resource
03:  WHERE     URI = @URI
```

At this point, we have a resource identifier *@rID* which we want to represent an event sequence. If a user has an access identifier *@aID*, the SQL statement for STEP 2 should be the following:

```
01:  SELECT    eID, MAX(depth) AS depth
02:  FROM      event
03:  INTO      #STEP2
04:  WHERE     1 = CHARINDEX(aID, @aID)
05:  AND       rID = @rID
06:  GROUP BY  eID, rID
```

In this case, this query result is stored into the temporary table #STEP2. However, it is also possible to create a view over STEP 2.

Example 8 In *case of @aID = '110' or @aID = '10', the results of STEP2 are given in Fig. 10.*

Finally, STEP 3 is expressed by the following query.

```
01:  SELECT    event.property, event.type
02:  FROM      #STEP2, event
03:  WHERE     #STEP2.eID = event.eID
04:  AND       #STEP2.depth = event.depth
05:  AND       1 = CHARINDEX (aID, @aID)
06:  AND       event.rID = @rID
07:  AND       type <> 0
08:  ORDER BY  event.eID;
```

Example 9 *Figure 11 shows results of STEP 3 based on the result of STEP 2. The results are ordered. Therefore, these are considered to be event sequences. The account identifier of 110 is for a person who is under the legal age for smoking (see Fig. 7). Its result contains no data regarding cigarettes and cost.*

Because of the query, users can have access to nothing but events that are allowed by the users' account identifier. Arguments of this query are a file name and a user identifier.

5.4 User annotations

As mentioned above, accounts of SAXOPHONE constitute a tree. The access policy of a certain account inherits from its ancestor's access policies. In addition, our system allows users to annotate XML documents with their local data. The annotations must be a part of XML documents. In other words, the annotated XML document must be in well-formed XML style. Annotations must be

resource

rID	URI
1	http://localhost/example.xml

event

rID	eID	aID	depth	type	property
1	1	1	1	1	kiosk
1	2	1	1	1	cigarettes
1	2	110	3	0	NULL
1	3	1	1	2	name="menthol"
1	3	110	3	0	NULL
1	4	1	1	1	cost
1	4	11	2	0	NULL
1	5	1	1	3	200
1	5	11	2	0	NULL
1	6	1	1	4	cost
1	6	11	2	0	NULL
1	7	1	1	1	price
1	7	110	3	0	NULL
1	8	1	1	3	250
1	8	110	3	0	NULL
1	9	1	1	4	price
1	9	110	3	0	NULL
1	10	1	1	4	cigarettes
1	10	110	3	0	NULL
1	11	1	1	1	drink
1	12	1	1	2	name="orange juice"
1	13	1	1	1	cost
1	13	11	2	0	NULL
1	14	1	1	3	80
1	14	11	2	0	NULL
1	15	1	1	4	cost
1	15	11	2	0	NULL
1	16	1	1	1	price
1	17	1	1	3	120
1	18	1	1	4	price
1	19	1	1	4	drink
1	20	1	1	1	newspaper
1	21	1	1	2	name="times"
1	22	1	1	1	cost
1	22	11	2	0	NULL
1	23	1	1	3	100
1	23	11	2	0	NULL
1	24	1	1	4	cost
1	24	11	2	0	NULL
1	25	1	1	1	price
1	26	1	1	3	110
1	27	1	1	4	price
1	28	1	1	4	newspaper
1	29	1	1	4	kiosk

Figure 9: An example of access-controlled XML document as relational data

result (@aID = 10)

eID	depth
1	1
2	1
3	1
4	1
5	1
6	1
7	1
8	1
9	1
10	1
11	1
12	1
13	1
14	1
15	1
16	1
17	1
18	1
19	1
20	1
21	1
22	1
23	1
24	1
25	1
26	1
27	1
28	1
29	1

result (@aID = 110)

eID	depth
1	1
2	3
3	3
4	2
5	2
6	2
7	3
8	3
9	3
10	3
11	1
12	1
13	2
14	2
15	2
16	1
17	1
18	1
19	1
20	1
21	1
22	2
23	2
24	2
25	1
26	1
27	1
28	1
29	1

Figure 10: A result set of STEP 2

result (@aID = 10)

rID	eID	aID	depth	type	property
1	1	1	1	1	kiosk
1	2	1	1	1	cigarettes
1	3	1	1	2	name="menthol"
1	4	1	1	1	cost
1	5	1	1	3	200
1	6	1	1	4	cost
1	7	1	1	1	price
1	8	1	1	3	250
1	9	1	1	4	price
1	10	1	1	4	cigarettes
1	11	1	1	1	drink
1	12	1	1	2	name="orange juice"
1	13	1	1	1	cost
1	14	1	1	3	80
1	15	1	1	2	cost
1	16	1	1	1	price
1	17	1	1	3	120
1	18	1	1	4	price
1	19	1	1	4	drink
1	20	1	1	1	newspaper
1	21	1	1	2	name="times"
1	22	1	1	1	cost
1	23	1	1	3	100
1	24	1	1	4	cost
1	25	1	1	1	price
1	26	1	1	3	110
1	27	1	1	4	price
1	28	1	1	4	newspaper
1	29	1	1	4	kiosk

result (@aID = 110)

rID	eID	aID	depth	type	property
1	1	1	1	1	kiosk
1	11	1	1	1	drink
1	12	1	1	2	name="orange juice"
1	16	1	1	1	price
1	17	1	1	3	120
1	18	1	1	4	price
1	19	1	1	4	drink
1	20	1	1	1	newspaper
1	21	1	1	2	name="times"
1	25	1	1	1	price
1	26	1	1	3	110
1	27	1	1	4	price
1	28	1	1	4	newspaper
1	29	1	1	4	kiosk

Figure 11: A result set of STEP 3

one or more events that include: startElement, Attribute, Text and endElement. The following XML document is an example of annotation using Attribute.

```
01:  <cocktail annotation= "delicious">
02:          Martini
03:  </ cocktail >
```

Any XML element that consists of three events such as startElement, Text, and endElement is allowed as annotations.

```
01:  < cocktail >
02:          Martini
03:          <annotation> delicious </annotation>
04:  </ cocktail >
```

Two different kinds of annotation are defined in our approach:

1. **Descendant local annotation**: Not only annotators, but also their descendants can read the annotations.
2. **User local annotation**: Only annotators themselves have permission to read the annotations.

Annotations have a resource identifier, an event identifier, an account identifier, a depth, an event type, and an event property. The event identifier of annotations must be larger than that of the preceding event and smaller than that of the succeeding event. All event identifiers of XML document in SAXOPHONE are integers. Therefore, the event identifier of annotations should have a decimal fraction to insert between existing events. The following formula produces an xth (x is an integer which must be equal to or smaller than the number of events to annotate.) event identifier $E_a(x)$ between E_p and E_s.

$$E_a(x) = \frac{E_s - E_p}{x + 1} + E_p$$

5.4.1 Descendant local annotation

This annotation can be read by annotating users and their descendants. It consists of an event sequence that has an access identifier of the user who annotated.

Example 10 *Consider the annotation of XML documents that is shown in Fig. 1. The following SQL statement is an example of annotation of a user who has the account identifier of 11.*

```
01:  --           For cigarettes
02:  INSERT INTO       event
03:  VALUES            (1, 3.5, 11, 2, 2, 'smell="cool"');
04:  --           For a drink
05:  INSERT INTO       event
06:  VALUES            (1, 12.5, 11, 2, 2, 'taste="good"');
07:  --           For a newspaper
08:  INSERT INTO       event
09:  VALUES            (1, 21.5, 11, 2,2,  'type="right"');
```

This query inserts the *annotation* Attribute event next to *name* attribute in the event sequence.

In the case of Example 10, not only users with the account identifier of 11, but also their descendants, such as 110, can read the annotation. However, according to the access policy, annotation for cigarettes should be hidden from users who have the account identifier of 110. Consequently, it is necessary to apply the user local annotation.

5.4.2 User local annotation

The user local annotation is only readable by annotators themselves. Therefore, a NULL event is required to hide the annotation.

Example 11 *The following SQL statement adds the User local annotation to the element of* cigarettes.

```
01:  --           For cigarettes
02:  INSERT INTO       event
03:  VALUES            (1, 3.5, 11, 2, 'smell="cool"');
04:  INSERT INTO       event
05:  VALUES            (1, 3.5, 110, 0, 'NULL');
```

In this example, 110 users read the NULL event; hence the annotation is hidden from their descendants.

Figure 12 shows results of the query to the annotated XML document based on Examples 10 and 11.

6 Authorization

Section 5 shows access control of the XML repository, SAXOPHONE. This section describes implementation of access control with the relational database. We use the Microsoft SQL Server 2000 for this implementation.

The salient portion of this study is the unification of user authorization. If users must use two individual accounts for an XML repository and a relational database, it is both inconvenient and insecure. Figure 13 shows two different accounts in which account X is for the XML repository and account DB is for the relational database. When users select an XML document from the XML repository, the XML repository first reads an access policy for account X. It then queries the relations using account DB. In this procedure, the account DB must be allowed to SELECT from all instances of relations. This procedure engenders the problem that users who know the account DB and its password can read all data of XML documents in the relations, even if their access policy does not allow it.

result (@aID = 11)

rID	eID	aID	depth	type	property	
1	1	1	1	1	kiosk	
1	2	1	1	1	cigarettes	
1	3	1	1	2	name="menthol"	
1	**3.5**	**11**	**2**	**2**	**smell="cool"**	annotation
1	7	1	1	1	price	
1	8	1	1	3	250	
1	9	1	1	4	price	
1	10	1	1	4	cigarettes	
…	…	…	…	…	…	

result (@aID = 110)

rID	eID	aID	depth	type	property	
1	1	1	1	1	kiosk	
1	11	1	1	1	drink	
1	12	1	1	2	name="orange juice"	
1	**12.5**	**11**	**2**	**2**	**taste="good"**	annotation
1	16	1	1	1	price	
1	17	1	1	3	120	
1	18	1	1	4	price	
1	19	1	1	4	drink	
1	20	1	1	1	newspaper	
1	21	1	1	2	name="times"	
1	**21.5**	**11**	**2**	**2**	**type="right"**	annotation
1	25	1	1	1	price	
1	26	1	1	3	110	
1	27	1	1	4	price	
1	28	1	1	4	newspaper	
1	29	1	1	4	kiosk	

Figure 12: Annotated event sequences

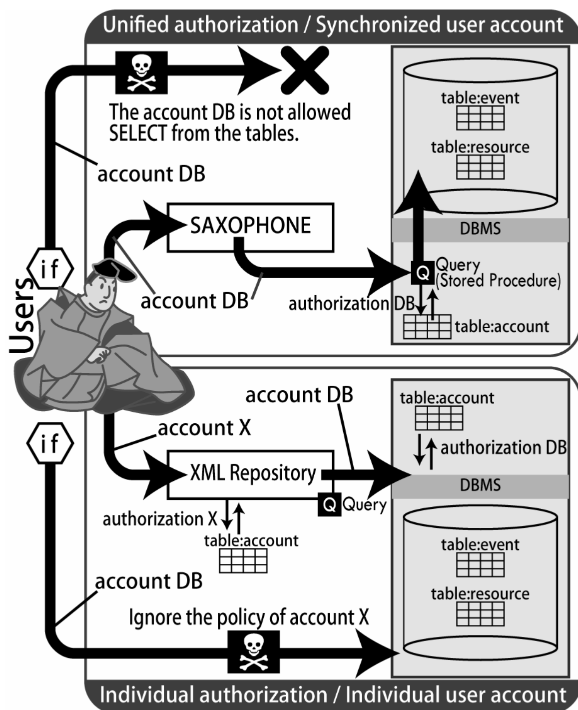


Figure 13: User Authorization

The following sections illustrate a method to unify the two kinds of authorization and synchronize access policies between the XML repository and relations.

6.1 Interface

As mentioned above, the relational database is hidden from applications. SAXOPHONE has a SAX interface and the applications use the XML repository via a SAX parser. SAX parsers usually require only a file name. Therefore, SAXOPHONE needs a file name, provided that the file name is represented in URI-style as follows.

http://username:password@localhost/example.xml

This statement contains scheme, username, password, hostname, and file path. The username and the password must be a valid account of the DBMS.

6.2 Account mapping

Section 4 described the account identifier. The SAXOPHONE access policy is written using it, but users access this system via a username and password. Therefore, it is necessary to map the username to the account identifier. Consequently, this correspondence is stored in a relation.

The relation named account has two database attributes aID and username. The following query resolves on the account identifier via username:

```
01:   SELECT   aID
02:   FROM     account
03:   WHERE    username = @accountDB
```

6.3 Access policy of the relational database

Figure 13 shows that SAXOPHONE in this case has three tables and one query. The query is in a stored procedure. A stored procedure is a set of SQL statements that can be stored in the server. The account DB in the figure is a unified account of the DBMS and SAXOPHONE. Users can access XML documents in SAXOPHONE via an account DB.

The DBMS needs a username and a password of the account DB to access the database. The account is allowed to execute the stored procedure, but is not allowed to select, update and delete from the relations. For that reason, users never acquire XML documents and its SAX events without access control.

In SAXOPHONE, if users have already got accounts of DBMS, the user can access the XML repository via the accounts and can gain access to controlled XML documents.

7 Conclusion

This paper has outlined implementation of an access control method on SAXOPHONE, the XML repository. The salient point of this research is to realize an access control method of XML repository synchronized with the DBMS' access control mechanism. This paper also describes a novel access identifier based on a prefix-labeling scheme for hierarchical authorization. Account policies for hierarchical authorization are inheritable. The access identifier can represent inherited account policies efficiently.

Our contribution is the expression of a method to store numerous XML documents into relational databases and control access on such as XML document. Generally, to deal with large XML documents, it is important that users be able to search the documents. For that reason, the future direction of this study will be to implement an XML query interface on SAXOPHONE.

8 References

SAX, http://www.saxproject.org/. Accessed 1 Sep 1004.

XACML: OASIS, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml. Accessed 1 Sep. 2004.

Kudo, M. and Hada, H. (2000): XML Document Security based on Provisional Authorization. Proc. of the 7th ACM Conference on Computer and Communications Security, Nov. 2000, 87-96.

Fernandez M.F., Morishima A., and Suciu D. (2001): Efficient Evaluation of XML Middle-ware Queries. In SIGMOD.

Fernandez M.F., et al., (2001): Publishing Relational Data as XML: The SilkRoute Approach. IEEE Data Engineering Bulletin 24(2).

Carey et al., (2000): XPERANTO: Publishing Object-Relational Data as XML. In Workshop on Web and Databases (WebDB).

Yoshikawa M., Amagasa T., Shimura T., and Uemura S. (2001): XRel: A Path-Based Approach to Storage and Retrieval of XML Documents using Relational Databases, ACM Transactions on Internet Technology, Vol. 1, 1: 110–141.

DOM, The World Wide Web Consortium (W3C), http://www.w3.org/DOM/. Accessed 1 Sep. 2004.

Cohen H., Kaplan H., and Milo T. (2002): Labeling dynamic XML trees, Proc. 21st Symposium on Principles of Database Systems (PODS), Madison, USA, ACM SIGACT-SIGMOD-SIGART.

Li Q. and Moon B. (2001): Indexing and querying XML data for regular path expressions. Proc. 27th International Conference on Very Large Data Bases (VLDB), Rome, Italy.

Abiteboul S., Kaplan H., and Milo T. (2001): Twelfth Annual ACM-SIMAM Symposium on Discrete Algorithms (SODA), Washington, D.C., USA. 547-556.

PostgreSQL, http://www.postgresql.org/. Accessed 1 Sep 2004.