Access Control of XML documents in Relational Database Systems

Kian-Lee Tan Mong Li Lee Yue Wang
Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 117543

Abstract ***In this paper, we present the design and implementation of an access control system for XML documents. Our system stores XML documents as relational tables in a relational database management system, and operates in three phases. In phase 1, the authorization rules are pre-processed to determine the information that should be protected. In phase 2, only the necessary information are retrieved from the database, i.e., the protected information are not accessed. Finally, in phase 3, the retrieved data are verified against the (remaining) authorization rules to filter away those that cannot be disclosed.***

Keywords: Access control model, XML, relational database systems, authorization, scalable.

1 Introduction

In recent years, there is an increasing amount of information being distributed and shared in XML format over corporate Intranets and the global Internet. As a result, it becomes critical to define and enforce access restrictions on XML documents to ensure that only authorized users have access to the information. Clearly, the simple method of protecting an entire document at the file level is unattractive (since it limits the number of users) and unnecessary (as different users may be allowed to access different portions of the document).

One promising access control model proposed in the literature is to enforce access restrictions directly on the structure and content of XML documents [1, 2]. In this way, information in XML format can be protected at a finer level of graularity than the whole document, e.g., at the tag level. As an example, a user may be blocked from accessing information tagged by `XLink` and `XPointer`, while another may be allowed to access the entire document.

The design of an access control system based on this model was reported in [1, 3]. The system essentially represents an XML document as an object tree, according to the Document Object Model (DOM) Level 1 specification [7]. DOM provides an object-oriented Application Program Interface (API) for HTML and XML documents. To enforce security, the DOM tree is repeatedly traversed to mark out nodes that should be denied access. This step is done based on the authorization rules for the user. Finally, the DOM tree is traversed to prune away the nodes that are marked, and the remaining DOM structure represents information that can be accessed by the user. This architecture has three main problems. First, it requires the entire DOM tree to be memory resident. This problem is exacerbated by the fact that a DOM tree is typically much larger (about 5 to 10 times) than its XML document. Thus, the scheme is not scalable for large XML documents. Second, the initial response time is long. This is because all the steps must be completed before any answers are returned to the user. Third, the scheme is inefficient. This is because the DOM tree has to be repeatedly traversed and its nodes labelled. More importantly, a lot of unnecessary information is also loaded (in the sense that the system loads the entire XML document and then prunes away information that should not be disclosed.).

In this paper, we present the design and implementation of an access control system that overcomes the above limitations. The system has the following features. First, it can regulate access control at a fine graularity (e.g., at the tag level). Second, it stores XML documents as tables in relational databases. Third, it is efficient compared to existing systems as it only examines the relevant data. Fourth, it is scalable as it can handle very large XML documents that may not fit into the main memory. Finally, it provides very fast initial response time.

2 Background

Our proposed access control system is largely influenced by several recent technologies by two research communities. On the one hand, the XML community has looked at securing XML documents. On the other hand, the database community has focused on internal storage organization of XML documents. We review some of these technologies here.

In [1, 2], an authorization model for regulating access to XML documents was proposed. The model defines an XML markup for a set of security elements to describe the protection requirements of XML documents at the granularity of XML elements at both instance and schema levels. Coupled with a user's identification and its associated group memberships, and with support of both permissions and denials of access, the security markup allows different protection requirements to be easily expressed. By enforcing the requirements stated by the authorizations, a requester only receives the information that he is granted access. This model is very powerful, and in this paper, we also adopt it. However, our system makes use of XML schema [8] instead of DTD.

In [1, 3], a prototype system that is based on the above access control model was implemented. As discussed in the introduction, the system is not efficient in terms of response time and memory utilization, and it is also not scalable.

The database community, on the other hand, have focused on how to store XML documents in relational databases [5, 6]. These techniques essentially transform XML data into relational tables that can be stored in a relational database management system. A request on an XML document may then involve merging multiple tables to produce the answers. Experimental results on prototype systems generally show that such a method can produce fast initial response time.

It is interesting to note that while we can store XML documents as relations in a DBMS, we cannot exploit database security features to protect the information in existing commercial DBMSs. In a DBMS, access control is facilitated by GRANT commands or the view mechanism that specify what a user is allowed to see. However, this basically pushes the responsibility to the user, i.e., the user must be aware of what (s)he wants to access. This is unlike access control policies for XML where a user need not be aware of his access rights, and yet the system will evaluate his/her query, and returns to the user a view of what (s)he is granted to access.

3 XENA: An XML Security Enforcement Architecture

Our aim is to integrate the strengths of the two research communities to design a scalable XML access control system. Our system, XENA (Xml sEcurity eNforcement Architecture), have several desirable characteristics. First, from the user (end user or admininstrator) point of view, there is only one "language" and one format. In other words, a query is specified either in a XML query language (e.g., XML-QL [4]) or by an URL (if an XML document is to be accessed). Similarly, the access control administrator specifies his access control policies in an XML format. We note that the access control can be defined on both XML instances and XML schema. While we have adopted the access control model in [1], we have implemented the model so that it operates on XML schema (which is expected to replace DTD very soon). Thus, in XENA, every XML document has its own XML Access Sheet (XAS) that specifies the authorization policy for the document. The XAS is also specified in XML format.

Second, XML documents are stored as tables in relational databases. This allows us to tap into the strengths of relational databases. In particular, by storing XML documents as tables, we only need to bring in the necessary data – attributes that are not needed can be projected out, and tuples that are not needed can also be filtered out easily. In addition, the iterator model of evaluating queries allows answers to be returned as soon as they are produced. This facilitates fast initial response time. To store XML documents as tables, we need to map the content of XML documents into tuples of tables. In this work, we have adopted the XStorM mapping strategy [6]. XStorM employs a data-mining strategy to automatically identify the relational schema of an XML document. The content of the XML documents can then be mapped into tuples in these relations. Note that a single XML file can be mapped into multiple tables. In other words, we may need to merge multiple tables to produce the original XML document (using outerjoin or join operations). Since our focus is not on the mapping strategy here, readers are referred to [6] for further details.

Third, the system is scalable. Because operations at the backend (XML storage) are performed on tables (in fact, on projected attributes of the tables), the memory requirement is independent of the size of the XML documents. As such, XENA can handle very large XML documents.

Figure 1 shows the architecture of XENA for a stand-alone system. (The architecture can be easily extended to a networked environment by allowing two levels of access control [1].) Initially, all XML documents are mapped into relational tables by the **XML-Relation Transformer** module. The process also maintains the mapping information in the metadata database, including the correspondences between the XML document objects and the relation names, and the XML document attributes and the relation attributes. New documents are preprocessed and added in a similar manner.

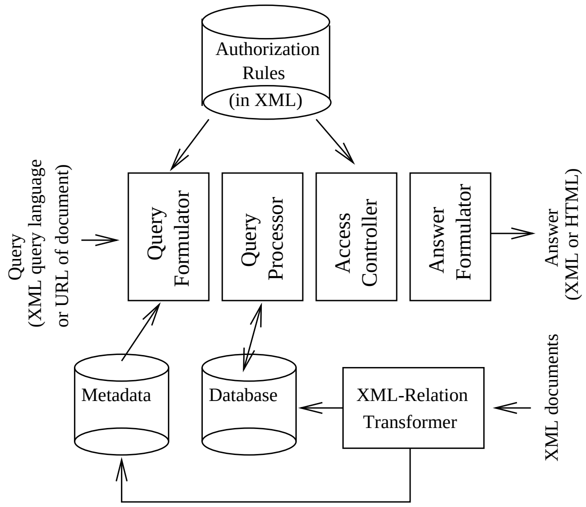


Figure 1: Architecture of XENA.

When a query arrives, the **Query Formulator** parses the query and maps it into an internal query. This module essentially examines the authorization rules and the metadata corresponding to the requested documents, and produces a query that will retrieve only those information that can be accessed by the requester. For example, if the requester is not allowed to see a certain attribute, then this attribute will be filtered out (by not specifying it in the target list) so that it is not necessary to access it from the database. Similarly, those tuples that the requester are not allowed to access are also filtered out (by introducing a selection predicate to prune them away).

The internal query is then evaluated by the **Query Processor**. This module accesses the data from the database according to the internal query, i.e., only the relevant information will be accessed.

Note that the requester may still not be able to view all the retrieved data. This is because there may be special cases that cannot be handled by the Query Formulator. For example, the requester may be allowed to view attributes A, B and C. However, (s)he is not allowed to look at those tuples whose B values are say 5, 14, 27 and 39. Note that while we can introduce selection predicates on B, this may be very tedious if the list of exceptions are not small and does not follow any pattern. As such, it may be easier to allow them to be accessed first, and then prune them away later. As another example, a more complex policy may involve some conditions that invoke a routine external to (i.e., not supported by) the DBMS. This is exactly the task of the **Access Controller** module. It basically applies the set of authorization rules that cannot be enforced by the Query Formulator on the retrieved tuples. We note that for many applications that involve only simple access rules, this phase need not be applied!

Finally, the **Answer Formulator** formulates the output in a form (e.g., XML or HTML) for the requester. We note that XENA only allows the users to view what (s)he is allowed to access and nothing else.

4 An Example

In this section, we shall illustrate the process of the proposed system with an example. DreamToFly.org is a travel agent. Its staff's personal data is defined by the XML schema `staff_info.xsd`, and the XML source of the data is stored in `staff_info.xml`. Figure 2(a) shows the schema, and Figure 2(d) shows a sample XML instance of `staff_info.xml` with two staff: Jannie and Manuel. DreamToFly.org stores the rest of its data in a similar way.

In XENA, under XStorM, `staff_info.xml` will be stored as two tables `Person(name, age, phone, photo)` and `CreditCard(name, creditCardNo)`. Here, we assume that `name` is the key of relation Person, and creditCardNo is the key of table CreditCard. The content of the two files for our example are as shown in Figure 3.

To gain access to view the organization's staff documents, a person must be a registered member of the organization. All registered persons are kept in `users.xml` which follows the schema `users.xsd`. Figure 2(b) and (e) illustrates an example with some sample instance.

To define some protection requirements for a DreamToFly.org's xml instance document, a normal xml-like instance document is attached to it. Using our staff information example (i.e., `staff_info.xml`), the corresponding file that specifies the protection requirement is `staff_authorization.xss` (see Figure 2(f)).

In general, an instance document can possess more than one authorization specification file to define the protection requirements for it. On the other hand, an authorization specification file can apply its authorization rules to more than one XML instance document as long as the XML instance document has a link to it.

Like the XML instance documents, such an authorization specification file needs to follow some schema. In DreamToFly.org, all the authorization specification files (i.e. all .xss files) must have the same format and so they share one common schema definition file. In our running example, this file is `staff_authorization.xsd` (see Figure 2(c)). Basically, this schema defines authorization rules, each of which has five fields [1]:

- Subject: the user name or user's client machine IP address or Symbolic name
- Object: Xpath 1.0 expression
- Action: read
- Sign: +/- (+ or grant, and - for denial)
- Type: LDH, RDH, L, R, LD, RD, LS, RS which represent types of authorization (e.g., L*ocal*, R*ecursive* - see [1, 2] for details).

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
    <xsd:element name="age" type="xsd:string"/>
    <xsd:element name="creditcard" type="xsd:string"/>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="person">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="name"/>
                <xsd:element ref="age"/>
                <xsd:element ref="phone"/>
                <xsd:element ref="creditcard" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="photo" type="xsd:string"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="phone" type="xsd:string"/>
</xsd:schema>
```

(a) staff_info.xsd.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
    <xsd:element name="UsersGroups">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="groups"/>
                <xsd:element ref="users"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="belongtoGroup">
        <xsd:complexType>
            <xsd:attribute name="name" type="xsd:string" use="required"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="group">
        <xsd:complexType>
            <xsd:sequence minOccurs="0" maxOccurs="unbounded">
                <xsd:element ref="belongtoGroup"/>
            </xsd:sequence>
            <xsd:attribute name="name" type="xsd:string" use="required"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="groups">
        <xsd:complexType>
            <xsd:sequence maxOccurs="unbounded">
                <xsd:element ref="group"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="user">
        <xsd:complexType>
            <xsd:sequence minOccurs="0" maxOccurs="unbounded">
                <xsd:element ref="belongtoGroup"/>
            </xsd:sequence>
            <xsd:attribute name="firstname" type="xsd:string"/>
            <xsd:attribute name="lastname" type="xsd:string"/>
            <xsd:attribute name="login" type="xsd:string" use="required"/>
            <xsd:attribute name="password" type="xsd:string" use="required"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="users">
        <xsd:complexType>
            <xsd:sequence maxOccurs="unbounded">
                <xsd:element ref="user"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

(b) users.xsd.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
    <xsd:element name="action">
        <xsd:complexType>
            <xsd:attribute name="value" use="required">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:NMTOKEN">
                        <xsd:enumeration value="read"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="authorization">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="subject"/>
                <xsd:element ref="object"/>
                <xsd:element ref="action"/>
                <xsd:element ref="sign"/>
                <xsd:element ref="type"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="object" type="xsd:string"/>
    <xsd:element name="set_of_authorizations">
        <xsd:complexType>
            <xsd:sequence maxOccurs="unbounded">
                <xsd:element ref="authorization"/>
            </xsd:sequence>
            <xsd:attribute name="about" type="xsd:string" use="required"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="sign">
        <xsd:complexType>
            <xsd:attribute name="value" type="xsd:string" use="required"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="subject" type="xsd:string"/>
    <xsd:element name="type">
        <xsd:complexType>
            <xsd:attribute name="value" use="required">
                <xsd:simpleType>
                    <xsd:restriction base="xsd:NMTOKEN">
                        <xsd:enumeration value="L"/>
                        <xsd:enumeration value="R"/>
                        <xsd:enumeration value="LD"/>
                        <xsd:enumeration value="RD"/>
                        <xsd:enumeration value="LW"/>
                        <xsd:enumeration value="RW"/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

(c) staff_authorization.xsd

```xml
<people>
    <person photo="./pic/bjennie.jpg">
        <name>Jannie</name>
        <age>24</age>
        <phone>98620389</phone>
        <creditcard>SB34-76142</creditcard>
        <creditcard>020-28282-0</creditcard>
    </person>

    <person photo="./pic/bjuan.jpg">
        <name>Manuel</name>
        <age>33</age>
        <phone>65-8731354</phone>
        <creditcard>200-38264-1</creditcard>
    </person>
</people>
```

(d) staff_info.xml.

```xml
<UsersGroups>
    <groups>
        <group name="Public"/>
        <group name="A">
            <belongtoGroup name="Public"/>
        </group>
        <group name="B">
            <belongtoGroup name="Public"/>
        </group>
    </groups>
    <users>
        <user firstname="Jannie"
                lastname="Busboom"
                login="jannie"
                password="2c 5f fe 83 3e 09 89 6b 0e ff
1a 6a 53 ed 7e 1f 48 0e d4 8d ">
            <belongtoGroup name="A"/>
        </user>
        <user firstname="Juan"
                lastname="Manuel"
                login="manuel"
                password="ad c1 6f a4 1a 38 b1 74 23 2f
20 6e 0b 2b d0 06 ba aa ce 68 ">
            <belongtoGroup name="B"/>
        </user>
    </users>
</UsersGroups>
```

(e) users.xml.

```xml
<set_of_authorizations>
    <authorization>
        <subject>Public[*|*</subject>
        <object>/people</object>
        <action value="read"/>
        <sign value = "+" />
        <type value="R"/>
    </authorization>
    <authorization>
        <subject>Public[*|*</subject>
        <object>/people/person/creditcard</object>
        <action value="read"/>
        <sign value="-"/>
        <type value="L"/>
    </authorization>
    <authorization>
        <subject>jannie|*|*</subject>
        <object>/people/person[./name="Jannie"]</object>
        <action value="read"/>
        <sign value="+"/>
        <type value="RDH"/>
    </authorization>
</set_of_authorizations>
```

(f) staff_authorization.xss.

Figure 2: An example.

Person			
name	**age**	**phone**	**photo**
Jannie	24	98620389	./pic/bjennie.jpg
Manuel	33	658731354	./pic/bjuan.jpg

CreditCard	
name	**creditCardNo**
Jannie	SB34-76142
Jannie	020-28282-0
Manuel	200-38264-1

Figure 3: Relations for staff_info.xml.

In our example, in `staff_authorization.xss`, 3 rules are defined:

- Rule 1: By default, the whole document can be accessed by everybody
- Rule 2: However, staff's credit card numbers cannot be seen by anybody
- Rule 3: But, Jannie can see everything about herself.

Consider a registered user, say Jannie, who submits a request for document `staff_info.xml`. Jannie is granted access to the system since she is one of the registered user. The system checks all attached authorization specification files related to `staff_info.xml`. Suppose there is only one authorization file called `staff_authorization.xss` attached to it. According to Jannie's access right, she can only view everything about herself, but not the credit card number of Manuel. The resulting view is shown in Figure 4.

```
<people>
    <person photo="./pic/bjennie.jpg">
        <name>Jannie</name>
        <age>24</age>
        <phone>98620389</phone>
        <creditcard>SB34-76142</creditcard>
        <creditcard>020-28282-0</creditcard>
    </person>
    <person photo="./pic/bjuan.jpg">
        <name>Manuel</name>
        <age>33</age>
        <phone>65-8731354</phone>
        <creditcard></creditcard> <--Note Manuel's creiditcard disappeared
    </person>
</people>
```

Figure 4: View of query.

XENA evaluates this query as follows. Since Jannie cannot view everybody's credit card number (except her own), the system only need to retrieve all the information from table Person, but only those tuples involving Jannie in table CreditCard. This is in contrast with the system in [1, 3] where it loads the entire `staff_info.xml`.

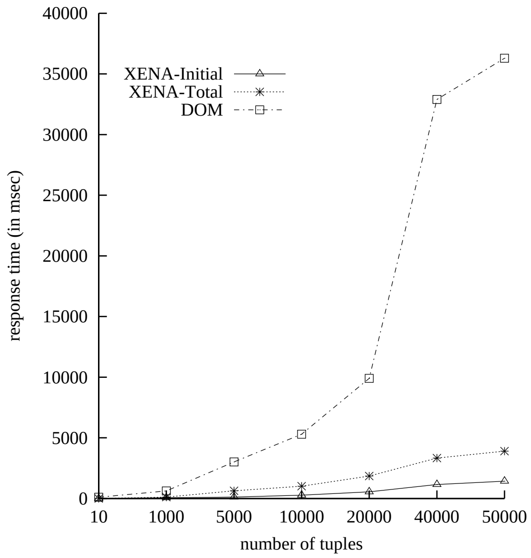
5 Current Status and Preliminary Results

We have implemented a prototype version of XENA in Java and the system runs on a Pentium 700 MHz PC. Our preliminary experience with the various components have been good. For example, XStorM shows that storing XML documents as tables can cut down the initial (i.e., first answer is produced) and total (i.e., all answers are produced) response time significantly compared to accessing the entirety of the XML documents. In particular, as soon as one answer is retrieved, it can be returned to the requester. The access control model can also correctly grant and deny accesses.

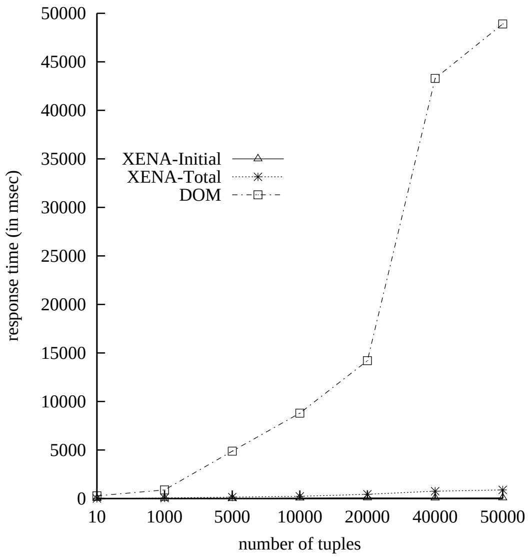
Figure 5 shows two representative results. We experimented with our running example with different number of tuples: 10, 1000, 5000, 10000, 20000, 40000 and 50000. The database size for 50000 tuples is about 1.8 MB. The traditional method is denoted DOM. There are also two values obtained for XENA — the inital and total response time. We also evaluated two users with different access rights. The first user is granted access to all information, while the second is only granted access to the information in Person (i.e., credit card information are to be protected from this second requester). Figure 5(a) shows the result for the former user, and Figure 5(b) shows the result for the latter. As shown, the proposed system is clearly superior to the DOM approach. In particular, for the second user, XENA can further reduce the response time significantly because it brings in only the relevant information (i.e., no access to table CreditCard).

6 Conclusion

This paper presents the design of a scalable XML access control system that integrates technologies from both the XML and database communities. The system can regulate access control at a fine graularity (e.g., at the tag level), is efficient in terms of memory consumption and response times (both initial and to-



(a) Query involving two relations.



(b) Query involving only one relation.

Figure 5: A comparative study.

tal), and is scalable. Our preliminary evaluation shows that the proposed approach is indeed scalable, effective and efficient. Our immediate task is to perform a more comprehensive experimental study.

Acknowledgement

This work is partially supported by a research grant RP960668 funded by the National University of Singapore and the National Science and Technology Board. We would like to thank the authors of [1] for providing us with their source code.

References

[1] E. Damiani, S. Vimercati, S. Paraboschi, and P. Samarati. Design and implementation of an access control processor for xml documents. In *Proceedings of the 9th International WWW Conference*, Amsterdam, May 2000.

[2] E. Damiani, S. Vimercati, S. Paraboschi, and P. Samarati. Securing xml documents. In *Proceedings of the 2000 International Conference on Extending Database Technology (EDBT'2000)*, Konstanz, Germany, March 2000.

[3] E. Damiani, S. Vimercati, S. Paraboschi, and P. Samarati. Xml access control systems: A component-based approach. In *Proceedings of the 14th IFIP 11.3 Working Conference in Database Security*, Amsterdam, August 2000.

[4] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Xml-ql: A query language for xml. In *Proceedings of the Query Languages Workshop (QL'98), http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/*, 1998.

[5] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. DeWitt, and J. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proceedings of the 1999 VLDB Conference*, pages 302–314, 1999.

[6] W.Q. Wang, M.L. Lee, B.C. Ooi, and K.L. Tan. Xstorm: A scalable storage mapping scheme for xml data. In *Proceedings of the 10th International WWW Conference (Poster)*, 2001.

[7] World Wide Web Consortium (W3C). Document object model (dom) level 1 specification verison 1.0. In *http://www.w3.org/TR/REC-DOM-Level-1*, 1998.

[8] World Wide Web Consortium (W3C). Xml schema part 0: Primer. In *http://www.w3.org/TR/2000/CR-xmlschema-0-20001024*, 2000.