A Framework for Constraint Checking Involving Aggregates for Multiple XML Databases using Schematron

Albin Laga and Praveen Madiraju

Department of Mathematics, Statistics, and Computer Science
Marquette University
P.O. Box 1881, Milwaukee, WI 53201

{alaga, praveen}@mscs.mu.edu

ABSTRACT

Many internet and enterprise applications now not only use XML (eXtensible Markup Language) as a medium for communication but also for storing their data either temporarily for an application or permanently as a means to represent their data. Most of these applications need to follow a set of rules, which are called as integrity constraints in the context of databases. We assume a setting in which data is distributed across multiple sites. In this paper, we introduce a notation for representing constraints affecting multiple XML databases, *Global XConstraints*. A single update on one site can cause these global XConstraints to be violated. Hence, we propose a framework for checking these constraint violations using Schematron. As a proof of concept, we present a prototype of our system implementation. Most of the processing in our approach happens in compile time; hence we save time during run time.

Keywords
XML Constraints, Global XML Constraint Checking, Schematron, XSLT, XML Databases

1. INTRODUCTION

When building or modifying any existing database, we need to follow certain rules describing its syntactic and semantic rules. We call these as constraints in the context of databases. In relational databases, these are implemented using primary keys, foreign keys, check constraints, assertions, triggers, and global constraints (for distributed databases). These are all part of relational schema. XML DTD [2], XML Schema [6], and RELAX NG [3] are three most commonly used schema languages for XML databases. It is now well understood that DTDs are not as expressive as XML schema but are very useful for associating some sort of structure to a very loosely structured XML document. XML Schema is verbose and can only implement constraints such as domain constraints, primary keys (using `<xs:key>`), and foreign keys (using `<xs:keyref>`). It fails to implement semantic integrity constraints. RELAX NG is becoming more popular because it is a much simpler way of expressing constraints. However, it suffers from the same drawbacks as XML Schema.

There are different constraint languages introduced, which are discussed in detail in Section 5. However, most of these languages are verbose and not user-friendly. Additionally, as discussed earlier, many of these languages do not support semantic integrity constraints including aggregates (sum, max, min, avg, and count). We have earlier introduced a notation for representing constraints for XML databases, called XConstraints [17], [18]. Global XConstraints are XML constraints affecting multiple XML databases. In this paper we extend our XConstraint notation to include aggregates. These XConstraints are based on datalog style notation. The logic based language provides a basic foundation for concise representation of constraints.

In this paper, we consider the setting in which we have multiple XML databases. When XML data at one site is changed, it can potentially violate global XConstraints. Hence, we propose a framework to check for these constraint violations using a Schematron [7] based approach. The schematron processor internally uses a XSLT [16] processor making it easier to implement and portable as even recent versions of browsers can act as XSLT processors. We also demonstrate the system implementation. The whole application is developed using Java JDK 1.5, and hence is platform independent. Checking for these integrity constraints is significant in the context of semantic query optimization, data cleaning, data integration systems [13].

The rest of the paper is organized as follows: In Section 2, we present a sample XML database which will be used throughout the paper and introduce XConstraint representation. We discuss our general framework for checking constraints in Section 3. We then present the system implementation in Section 4. In Section 5 we

describe related work and finally present our conclusions in Section 6.

2. PRELIMINARIES

Here we give an example healthcare XML database and introduce our notation for defining XConstraints.

2.1 Example Database

Consider a sample *healthdb.xml* shown in Figure 1. Figure 1 gives the logical representation of the HEALTHDB XML databases from different sites. Physically, information is distributed across multiple sites:

Site S_1: PATIENT information such as *SSN (primary key)*, *PName* and *HealthPlan* is stored. CASE information with *CaseId (primary key – like a sequence number)*, *SSN*, and *InjuryDate* is also stored.

Site S_2: patient's CLAIM information such as *CaseId (primary key)*, *ClaimDate*, *Amount* and *Type* is recorded.

Site S_3: TREATMENT information such as *CaseId (primary key)*, *DName* (doctor name), *TDate* (Treatment Date), and *Disease* is stored.

Note that a patient can suffer multiple injuries uniquely identified by their *CaseId* at Site S_1, and can also make multiple claims identified by their *CaseId* at site S_2.

```
1: <?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
2: <HEALTHDB>
3: <!-- S1 indicates site S1 -->
4:     <S1_PATIENTS>
5:         <PATIENT>
6:             <SSN>123</SSN>
7:             <PName>John</PName>
8:
    <HealthPlan>B</HealthPlan>
9:         </PATIENT>
10:        <PATIENT>
11:            <SSN>234</SSN>
12:            <PName>Clark</PName>
13:
    <HealthPlan>C</HealthPlan>
14:        </PATIENT>
15:    </S1_PATIENTS>
16:    <S1_CASES>
17:        <CASE>
18:            <CaseId>1</CaseId>
19:            <SSN>123</SSN>
20:
    <InjuryDate>10/14/2003</InjuryDate>
21:        </CASE>
22:        <CASE>
23:            <CaseId>2</CaseId>
24:            <SSN>234</SSN>
25:
    <InjuryDate>06/24/2004</InjuryDate>
26:        </CASE>
27:        <CASE>
28:            <CaseId>3</CaseId>
29:            <SSN>123</SSN>
30:
    <InjuryDate>10/12/2004</InjuryDate>
31:        </CASE>
32:    </S1_CASES>
33: <!-- S2 indicates site S2 -->
34:    <S2_CLAIMS>
35:        <CLAIM>
36:            <CaseId>3</CaseId>
37:
    <ClaimDate>11/14/2004</ClaimDate>
38:            <Amount>25000</Amount>
39:            <Type>Inpatient</Type>
40:        </CLAIM>
41:    </S2_CLAIMS>
42:    <!-- S3 indicates site S3 -->
43:    <S3_TREATMENTS>
44:        <TREATMENT>
45:            <CaseId>1</CaseId>
46:            <DName>Mike</DName>
47:
    <TDate>10/15/2003</TDate>
48:
    <Disease>SmallPox</Disease>
49:        </TREATMENT>
50:        <TREATMENT>
51:            <CaseId>3</CaseId>
52:            <DName>Blake</DName>
53:
    <TDate>10/14/2004</TDate>
54:
    <Disease>LegInjury</Disease>
55:        </TREATMENT>
56:    </S3_TREATMENTS>
57: </HEALTHDB>
```

Figure 1. healthdb.xml document indented with line numbers [17]

2.2 XML Constraint Representation

Semantic integrity constraints can be considered as a general form of assertions. They specify a general condition in the database which needs to be true always. Constraints of this type deal with information in a single state of the world. Throughout the paper, we denote semantic integrity constraints for XML database as *XConstraints*. *Global XConstraints* are the constraints spanning multiple XML databases. Here we give the constraint representation for global XConstraints.

A datalog rule (expressed as `Head ← Body`) without a `Head` clause is referred to as a denial. It is customary to represent integrity constraints in the logic databases as range restricted (safe or allowed) denials.

Definition 2.1: In order to represent global XConstraint in the context of XML database as query evaluation, we consider global XConstraint in the form of range restricted denials (datalog style notation) given below:

$C \leftarrow X_1 \wedge X_2 \wedge, \ldots, X_n$, where C is the name of the global XConstraint and each X_i is either an *XML literal* or A*rithmetic literal* or *Aggregate literal*. ■

We define XML literal, arithmetic literal and aggregate literal below. The definition of XML literal is chiefly inspired by Buneman *et al.* (2001) [10] and Chen *et al.* (2002a) [11]. Semantics for representing key constraints for a single XML database are given in [10] and [11]. We extend their semantics by introducing user defined variables, term paths and XML literals for representing global XConstraints for multiple XML databases.

Definition 2.2: An *XML literal* is defined as follows:

$X_i : (Q_i, (Q_i', [V_{i1} = t_{i1}, V_{i2} = t_{i2}, \ldots, V_{ik_i} = t_{ik_i}]))$

Using the syntax from [10], [11], Q_i, Q_i' and $t_{i1}, t_{i2}, \ldots, t_{ik_i}$ are path expressions corresponding to X_i . $V_{i1}, V_{i2}, \ldots, V_{ik_i}$ are user defined variables corresponding to $t_{i1}, t_{i2}, \ldots, t_{ik_i}$. Q_i is called the context path, Q_i' the target path and $t_{i1}, t_{i2}, \ldots, t_{ik_i}$ are the term paths. Context path Q_i identifies the set of context nodes, *c* and for each *c*, $V_{i1}, V_{i2}, \ldots, V_{ik_i}$ are the set of user defined variables corresponding to the term paths, $t_{i1}, t_{i2}, \ldots, t_{ik_i}$ reachable from *c* via Q_i'. ■

Definition 2.3: *Arithmetic literal* is defined as: *expression* θ *expression,* where expression – is a linear expression made of variables occurring in *XML literals,* integer constants, and the four arithmetic operator +, -, *, /; θ – is a comparison operator (=, <, >, <=, >=, <>). Joins between nodes are expressed either as an equality (=) between two variables in an arithmetic literal or by having the same variable name appear in different XML literals within the same global XConstraint. Note that variables with the same name cannot appear in the same XML literal. We also assume date arithmetic and string arithmetic. ■

Definition 2.4 : An *Aggregate literal* is expressed as

$A_i(\hat{s}, \alpha(y){:}v){:}\text{-}\ B$

Where (i) *B* is a conjunction of XML literals, (ii) $\hat{s}$ is the grouping list of variables that must appear some where in the body of the rule - B, (iii) α is aggregate function such as *avg, count, max, and min*, (iv) *y* is the aggregate variable, and (v) *v* is the result of applying the aggregate function. We assume that the aggregate literals are not recursive. ■

Now, we are ready to define the satisfiability of a global semantic integrity constraint (global XConstraint), C.

Definition 2.5: An XML tree T is said to satisfy a global integrity constraint (global XConstraint), C, if and only if the conjunction of $X_1, X_2, \ldots, X_n$ evaluates to *false* ■

The motivations behind using our constraint representation and negative semantics for checking the satisfiability of a global semantic integrity constraint are: 1) constraint representation using our approach resembles query evaluation for heterogeneous databases (logic, relational, XML) and hence is very generic due to the inherent logic based approach used in representing the XConstraints. 2) Global XConstraints generated using this approach are easier to translate into Schematron schema document as explained in "Schematron Generator" in Section 3. Note that each Q_i ,Q_i', user defined variables and the term paths corresponding to each XML literal - X_i has the site information referred to as S_j and can only refer to a single site. However, a global XConstraint has one or more XML literals and hence can refer to multiple XML databases. In case of Arithmetic literal or Aggregate literals, the variables in the expression could belong to different sites. If two variables are not the leaf nodes, the equality join among the two variables is similar to the node equality considered in [10].

Example 2.1: Consider a global XConstraint C_1 defined on *healthdb.xml*. Constraint C_1 states that a patient with HealthPlan 'B' may not claim more than 40000 dollars on a case diagnosed with 'SmallPox'.

```
C1:-
 (//S1:PATIENTS,
 (./PATIENT,[ssn=./SSN,healthplan=./HealthPlan])),
(//S1:CASES,(./CASE,[caseid=./CaseId,ssn=./SSN])),
(//S2:CLAIMS,(./CLAIM,[caseid=./CaseId,amount=
./Amount])),
(//S3:TREATMENTS,(./TREATMENT,[caseid=
./CaseId,disease=./Disease])),
 healthplan = 'B',disease = 'SmallPox',
 amount > 40000.
```

For the example contained in Figure 1, C_1 is satisfied.

C_1 is satisfied for the healthdb.xml as one of the arithmetic

literals *amount (line 38, value = 25000) > 40000* returns false and hence the whole conjunction for C_1 evaluates to false.

3. SYSTEM ARCHITECTURE

Here, we present a schematron based constraint checker for multiple XML databases in Figure 2.

Schematron Generator

The schematron generator shown in Figure 2 takes as input an XConstraint and generates a schematron document.

As new XConstraints are introduced to the system, we add it to the base schematron document. Here, we illustrate the generation of schematron from an XConstraint using the following steps.

Consider the following XConstraint, C_3 which states that "the sum of claim amounts for each patient with healthplan 'B' may not be more than 100000". This can be represented using our notation from Section 2 as follows:

```
C3:-   A(SUM(amount):v), v > 100000
A(SUM(amount):v):-  (//S1:PATIENTS,
(./PATIENT,[ssn=./SSN,healthplan=./HealthPl
an])),
(//S1:CASES,(./CASE,[caseid=./CaseId,ssn=./
SSN])),
(//S2:CLAIMS,(./CLAIM,[caseid=./CaseId,
amount=./Amount])), healthplan = 'B'.
```

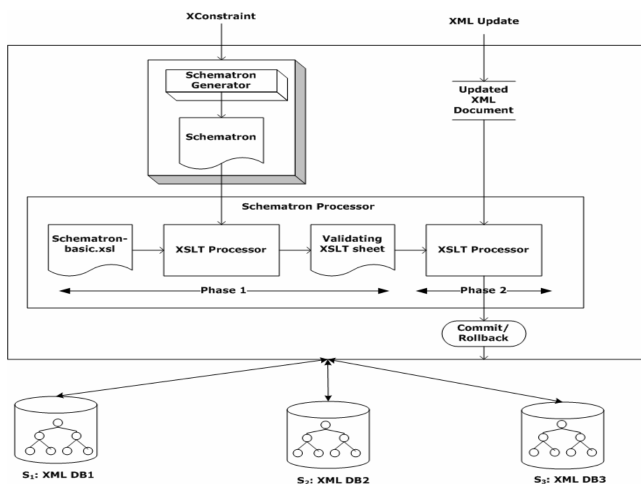

Figure 2. System Architecture

Constraint C3 will be translated using the Schematron Generator into a Schematron document using the following steps:

Step 1: Add the header

Each constraint is coded inside a <sch:pattern> tag. Any of the nodes can be considered for the context attribute of <sch: rule> element, including the root of the document.

```
<?xml version="1.0" encoding="UTF-8"?>
<sch:schema
xmlns:sch="http://www.ascc.net/xml/schematr
on">
 <sch:pattern name="C3">
  <sch:rule context="S2/S2_CLAIMS">
```

Step 2: XConstraint variables to <let> clause

XConstraint variables are mapped to <let> clause variables, and any subsequent occurrence of the same XConstraint variable is referred as $variable in the schematron document.

```
   <!-- Assign values to the variables,
collect data from different sites -->
   <sch:let name="caseId"
value="./CLAIM/CaseId"/>
   <sch:let name="amount" value=" sum(
./CLAIM[CaseId = $caseId]/Amount)"/>
   <sch:let name="ssn" value="
//S1_PATIENTS/PATIENT[CaseId =
$caseId]/SSN"/>
   <sch:let name="healthplan" value="
//S1_PATIENTS/PATIENT [SSN =
$ssn]/HealthPlan"/>
```

Step 3: Test variables for null or constant

Here, we are testing if any of the variables are null. If any variable is null, the "assert" test element will evaluate to false and as a result a fail message is displayed. We also check for simple conditions for variables such as for healthplan.

```
   <!-- Test whether variables are not null
-->
   <sch:assert test="$caseId">Value of
"$caseId" has not been applied</sch:assert>
   <sch:assert test="$amount">Value of
"$amount" has not been applied</sch:assert>
   <sch:assert test="$ssn">Value of "$ssn"
has not been applied</sch:assert>
   <sch:assert test="$healthplan =
'B'">Value of "$healthplan" has not been
applied</sch:assert>
```

Step 4: The Main Test Condition

Here, we are performing the comparison of the sum of claim amounts and the maximum possible value for the health plan of each person. A fail message is displayed if any variable was null or the test expression of an "assert" element evaluates to false.

```
   <sch:assert test="100000 &gt;= $amount"
>The amount claimed is too
high!</sch:assert>
  </sch:rule>
  </sch:pattern>
```

For each new constraint, we repeat steps 2-4, by adding new `<sch:pattern>` elements. At the end of all constraints, we close the schematron document using `</sch:schema>`

Schematron Processor

From users point of view, the schematron processor (see Figure 2) takes as input (i) schematron document generated from schematron generator, (ii) an updated document XML document , say D' and produces as output a decision to commit or rollback. If none of the constraints are violated, the updates are committed; otherwise the updates are rolled back.

As shown in Figure 2, the schematron processing occurs in two phases. In phase 1, the XSLT processor takes as input the schematron document generated from schematron generator and a schematron-basic.xsl (available from [7]) and generates a validating XSLT document. This phase can be done in compile time. In the second phase, the XSLT processor takes as input the validating XSLT document generated from phase 1, the updated XML document and generates as output if the constraints are violated. This phase occurs at run time. If none of the constraints are violated an action to commit to changes into the database is initiated. Otherwise, the changes are rolled back.

The advantages of schematron based approach are that it uses XSLT processor internally. Hence, for smaller sized documents, even recent versions of browsers can act as XSLT processors making it easier to check for constraint violations of XML documents on the fly without installing any special software. Also note that most of the processing is done in compile time. The schematron generation and Phase 1 of schematron processing happens in compile time. Hence, we save time during run time as only the Phase 2 of schematron processing happens at rum tine. However, in the current approach the entire XML document is checked for each update operation, and so performance is expected to suffer.

4. IMPLEMENTATION

The system architecture given in Section 3 has been implemented using JDK version 1.5. A prototype of the system implementation is given in Figure 3. Our implementation gives three options:

- `com.icl.saxon.StyleSheet` [4]
- `net.sf.saxon.Transform` [5]
- `org.apache.xalan.xslt.Process` [1]

The schematron panel (top panel) by default loads the basic schematron-xml1-6.xsl (available from [7]) and uses Saxon XSLT 1.0 [4] processor. The user can change these, if needed. The middle panel works as an editor for XML document or schematron schema given as tabbed windows in the panel. The bottom panel of the figure has two buttons, "Hide Details", and "Validate". When the user clicks "Validate", before the XML document is saved, it is checked for well-formed ness. Any violation of this prevents the application from saving the changes or proceeding further.

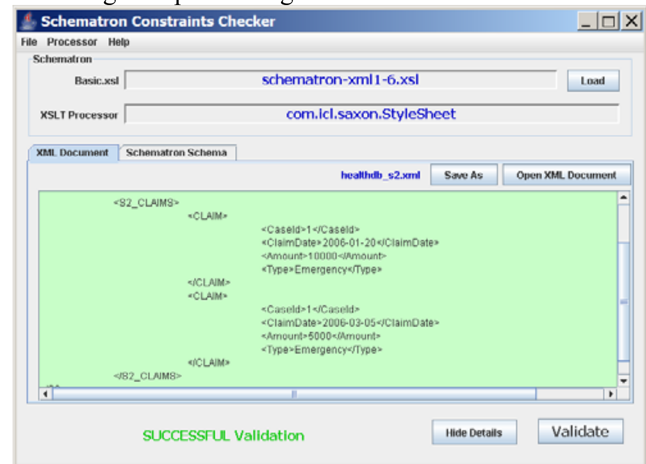

Figure 3. Constraint Checker Implementation

The XML document is then checked for any constraint violations against the schematron document using the steps illustrated in Section 3. If any of the constraints are violated, the changes to the XML document are rolled back. The "Hide Details" button gives the user options to control the verbose output from the system.

A sample screen shot of the application is shown for a constraint violation in Figure 4. The GUI has many user friendly features. Prior to validation, if any parameters are in red, it indicates a problem with that particular parameter. After validation a green status message is displayed if the validation was successful, otherwise a red status message is displayed.

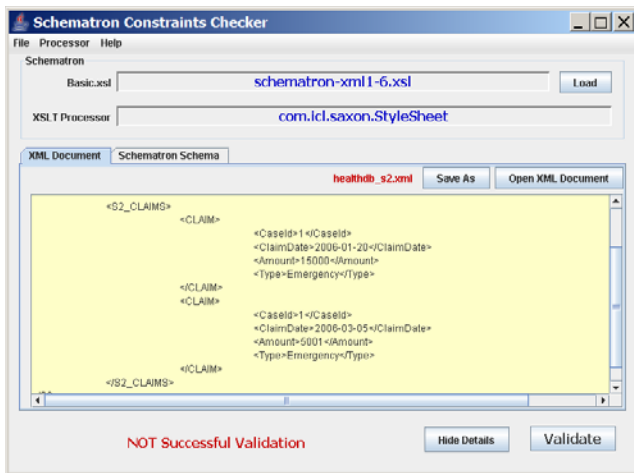

Figure 4. Main Window after Unsuccessful Validation

5. RELATED WORK

Constraint languages are complementary to XML DTD [2], W3C XML Schema [6] and RELAX NG [RELAX] schema languages. The major constraint languages used nowadays are:

Schematron [7] is a rule-based language with four level hierarchies (phases, patterns, rules, assertions). It is an assertion language based on presence or absence of names and values of elements and the attributes along the path. It is declarative and uses XML notation. It allows us to directly express rules without creating a whole grammatical infrastructure [20] because it offers extraordinary power in conjunction with other schema languages [22]. We have used the ideas discussed in [22], where we integrate our constraint representation to be able to use Schematron for constraint checking. Many advanced features like abstract pattern makes it even more expressible and flexible. CXiL [19], Constraint Language in XML is an assertion language based on first order logic and XPath. Similar to Schematron, CXiL is also an assertion language and both are path based. CXiL is based on first order logic while Schematron is based on Boolean logic and where Schematron requires scripting, CXiL can handle it natively. XCML [15], an eXtensible Constraint Markup Language adopts Unified Modeling Language (UML) and Object Constraint Language (OCL) to support visual specification and automation generation of SCML instances and XML schemas. It supports assertion-based constraints, simple rules-based constraints, composite rule-based constraints and it supports parameters for expressing dynamic constraints. XCSL [21], XML Constraint Specification Language is a domain specific language. It is implemented on XSLT platform language similar to Schematron; however they differ in some fundamental concepts. Each XCSL specification is defined as an XML instance and it is composed of one or more tuples. Each tuple has three parts: (i) *Context Selector*: Selects the context where we want to enforce constraint, (ii) *Context Condition*: The condition we want to enforce, and (iii) *Action*: The action we want to trigger when the condition does not hold. This is similar to the concept of using active database technology for enforcing constraints.

The idea of keys and foreign keys for XML was introduced in [10], [11]. The basic approach is to express constraints using path expressions. We have also studied the constraint representation in distributed databases. We have extended the approach of [10], [11] with datalog style notations and also used the concepts from [14] in representing XConstraints. A survey of recent languages for constraint specification is given in [13]. Research on validating keys for XML can be found in [8], [9], and [12]. However, we deal with constraint checking for semantic integrity constraints.

6. CONCLUSIONS

Semantic integrity constraints are rules that affect the consistency of XML documents. There are different constraint checking mechanisms introduced so far. A schematron based constraint checking is easy to implement, as it is based on XSLT approach. In this paper we have introduced aggregate constraint representation for XConstraints. We have proposed architecture for constraint checking involving aggregates for multiple XML databases using schematron. Our architecture is efficient as most of the steps happen in compile time; hence, we save during run time. We have also implemented a prototype for the system architecture.

We will extend this work by performing experimental evaluation of our system by comparing it against other approaches to constraint checking. The parameters we will consider are total time taken for different constraints, constraint validation time under varying loads of updates, and constraint checking time for different file sizes of documents. We also intend to develop a system for optimizing the constraint checking process for multiple XML databases.

7. REFERENCES

[1] Xalan Java Version 2.7.0 Available from: http://xml.apache.org/xalan-j/

[2] Extensible Markup Language (XML) 1.0 (Fourth Edition), W3C Recommendation 16 August 2006. Available at: http://www.w3.org/TR/REC-xml/

[3] RELAX NG home page. Available at: http://www.relaxng.org/

[4] About Saxon at: http://saxon.sourceforge.net/saxon6.5.5/

[5] Saxon XSLT 2.0 Processor. Available from : http://www.saxonica.com/documentation/index/intro.html

[6] XML Schema Part 0: Primer Second Edition, W3C Recommendation 28 October 2004. Available at: http://www.w3.org/TR/xmlschema-0/

[7] The Schematron 1.5 Specification. Available from: http://www.schematron.com/spec.html

[8] Benedikt, M., Chan, C.Y., Fan, W., Freire, J. & Rastogi, R. (2003). Capturing both Types and Constraints in Data Integration. *Proceedings of the ACM SIGMOD Conference on Management Of Data*

[9] Bouchou, B., Halfeld-Ferrari-Alves, M. & Musicante, M. (2003). Tree Automata to Verify XML Key Constraints. *International Workshop on the Web and Databases*.

[10] Buneman, P., Davidson, S., Fan, W., Hara, C., & Tan, W. (2001). Keys for XML. *World Wide Web*, pp. 201-210.

[11] Chen, Y., Davidson, S.B., & Zheng, Y. (2002a). Constraint Preserving XML Storage in Relations. *International Workshop on the Web and Databases*.

[12] Chen, Y., Davidson, S.B., & Zheng, Y. (2002b). XKvalidator: A Constraint Validator for XML. *Proceedings of ACM Conference on Information and Knowledge Management*.

[13] Fan, W. (2005). XML Constraints: Specification, Analysis, and Applications. First International Workshop on Logical Aspects and Applications of Integrity Constraints (LAAIC).

[14] Gupta, A., & Widom, J. (1993). Local Verification of Global Integrity Constraints in Distributed Databases. *Proceedings of the ACM SIGMOD Conference on Management of Data*.

[15] Hu, J., Tao, L. (2004). An Extensible Constraint Markup Language: Specification, Modeling, and Processing. Available at: http://www.idealliance.org/proceedings/xml04/papers/81/xml-2004-hu.html

[16] Kay, M (2003). XSL transformations (XSLT) version 2.0. *W3c working draft, World Wide Web Consortium*,. http://www.w3.org/TR/2003/WD-xslt20-20031112

[17] Madiraju, P., Sunderraman, R., Navathe, S.B., & Wang, H. (2006). Semantic Integrity Constraint Checking for Multiple XML Databases. *Journal of Database Management*, Vol. 17, No. 4, pp. 1-19.

[18] Madiraju, P., Sunderraman, R. & Navathe, S.B. (2004). Semantic Integrity Constraint Checking for Multiple XML Databases. *Proceedings of 14th Workshop on Information Technology and Systems* (WITS 2004), Washington D.C., December, 2004

[19] Marconi, M., Nentwich, C. (2004). CLiX Language Specification Version 1.0. Avilabe at: http://www.clixml.org/clix/1.0/clix.xml

[20] Ogbuji, U (2004). A hands-on introduction to Schematron. Available from: http://www-128.ibm.com/developerworks/edu/x-dw-xschematron-i.html

[21] Ramalho, J. (2001). XML Constraint Specification Language. *XML Europe Conference*. Avilable from: http://www.di.uminho.pt/~jcr/PROJS/xcsl-www/

[22] Robertson, E. (2002). Combining Schematron with other XML Schema languages. Available from: http://www.topologi.com/public/Schtrn_XSD/Paper.html