JESS-based web interface for XML document validation

Buhwan Jeong a,*, Jungyub Woo a, Boonserm Kulvatunyou b, Hyunbo Cho a

a Department of Industrial and Management Engineering, Pohang University of Science and Technology (POSTECH), San 31, Hyoja, Pohang, Kyungbuk 790-784, South Korea
b Manufacturing Systems Integration Division, National Institute of Standards and Technology (NIST), 100 Bureau Drive, Gaithersburg, MD 20899, USA

Abstract

Ensuring consistency among data exchange specifications in XML is critical to seamless integration of various business-to-business (B2B) applications. To this end, a specification should be thoroughly verified in manifold perspectives such as grammar/syntax conformance, design compliance, and canonical semantics accordance. A single hand-woven testbed implementation will fail to versatily support such a variety of tests. In this paper, we propose a novel validation tool of XML documents using an expert system and open validation rule scripts. Particularly, we illustrate a web-based implementation to ensure an XML schema’s design compliance to design and naming rules (NDR), namely a quality of schema design (QoD) test.

1. Introduction

The issues about interoperability have become a phenomenon in industry as well as in academia. It is well known that the costs occurring due to lack of interoperability among systems are beyond our estimate (for instance, see two reports for details (Brunnermeier & Martin, 1999; Gallaher & Gilday, 2004)). The interoperability may be achieved through long discussions with all the parties involved. However, practical constraints make it impossible.

Let us focus on development of standard data exchange specifications for further explanation. Inconsistencies among such specifications are inevitable due to the nature of distribution, evolution, and customization. The development is distributed because (1) project team members are physically distributed over the world, and moreover, (2) the communication among them, perhaps, having different viewpoints is inefficient due to concurrent development to meet a short budget cycle. Therefore, they are rush to propose standards without enough validation time to reach to a solid agreement. Even an agreed-upon standard is also evolving over time because it is impossible (1) to obtain resources enough to complete a project in a budget cycle, (2) to capture all the needs and constraints at the beginning of the project, and (3) to complete the entire project in a short period from the management perspective. Sometimes, it is due to immaturity of technologies. Furthermore, a standard specification needs to be customized for different industrial domains – different environments and assumptions for implementation.

For those reasons, the standard XML specifications must be thoroughly validated before deployed. At least four conformance tests should be conducted: schema grammar conformance, schema semantics conformance, schema design conformance, and canonical semantics conformance (Kulvatunyou, Ivezic, & Jeong, 2004). We propose to use a rule-based expert system for that purpose. More specifically, we address to construct and automatically transform an XML document into a knowledge base, and to encode...
test scripts from various design requirements. A web-based implementation and related issues are discussed.

The remainder of this paper is organized as follows: Section 2 delineates the rule-driven testbed architecture that supports generic XML document validation. Section 3 gives methods to construct a knowledge base and to encode test cases. Section 4 describes a case study, XML schema design quality testing, with issues related to web-based testbed implementation. Finally, Section 5 provides conclusion remarks.

2. Rule-driven XML document validation

We address the use of a rule-based expert system to validate XML documents. XML document validation checks for an XML schema/instance document according to test assertions on a validation engine. Fig. 1 depicts the highest view of rule-based XML validation in the IDEF0 model (Lingzhi, Leong, & Gay, 1996). The test assertions state testable requirements including the W3C XML grammar, design conventions, and/or canonical semantic models. For simplicity, we take the schema design quality (QoD) testing, in the context of which the rule-based expert system finds design violations against conventions and best practices on design. The validation rules are expressed in a declarative manner in order to keep them open. For example, the Schematron language1 encodes primitive rules in XPath expressions, and JESS script (Friedman-Hill, 2006; Jovanovic, Gasevic, & Devedzic, 2004) flexibly encodes more complex rules.

The XML validation further breaks down into three functionalities as shown in Fig. 2. In order to facilitate the testing, the XML schema documents must be first represented in a compatible format with the internal expert system. This process, namely knowledge base construction, transforms the input XML documents into a collection of knowledge nuggets, called (unordered) facts, which are fed to the test execution process as an input. The XSLT language is useful to automatically transform an XML document into the facts. The second process is to encode the design conventions in validation rule scripts. Since the rules can trigger actions based on the contents of one or more facts, they are used as controls for test execution. The last process is to fire actions (i.e., generate test results) in the test scripts depending on satisfaction of condition clauses.

3. Knowledge base construction

3.1. Templates and facts

The rule-based expert system maintains a knowledge base that is a collection of facts. In the (unordered) knowledge base, the facts must be stored in a certain structure. The structure must be declared before defining facts by using the deftemplate construct (whose syntax is depicted in Fig. 3A). The (deftemplate-name) is the head of facts being created. For a template, there may be an arbitrary number of slots, which describe the properties of a fact (deftemplate-name). Every (slot-name) must be an atom and the default slot qualifier states that its default value in a new fact is given by (value) (whose default value is the atom nil). The default-dynamic version will evaluate the given value each time a new fact using this template is asserted. Finally, the type slot qualifier specifies what data type the slot is allowed to hold and an acceptable value is one of any, integer, float, number, atom, string, lexeme, and object. For example, the templates for XML documents are defined in Fig. 3B. The first three templates are for element nodes, attribute nodes, and text nodes. The last relation template is to relate a parent element and its child node, which is one of element, attribute, or text node according to the type slot. They keep the original tree structure of a given document.

The facts are inserted into a knowledge base with the constructs assert and deffacts in forms of the templates above. The assert construct, i.e., (assert((fact-name) ((slot) (value)) *)), inserts a fact at each time, while the deffacts construct, a named list of facts, does these facts at a time, i.e., (deffacts (fact-list-name) ((fact-name) ((slot-name)(slot-value)) *)).

3.2. XML document transformation

The W3C XSLT (eXtensible Stylesheet Language Transformations) is a standard specification language for transforming an XML document into other forms such as another XML document, an HTML document, and even a plain ASCII document (Jovanovic & Gasevic, 2005). A transformation in the XSLT language is expressed as a well-formed XML document and describes transformation rules from a source (XML-native DOM) tree into a result tree. The result tree is separated from the source tree; thus the structure of the result tree can be completely different from that of the source tree by filtering and reordering elements, and adding arbitrary structures.

The transformation is achieved by associating patterns with templates. A pattern is matched against nodes in the source tree and a template is instantiated to create part of the result tree. When a template is instantiated, each instruction is executed and replaced by the result tree fragment.

---

Consequently, the instruction selects and processes its descendant elements. Processing a descendant element creates a result tree fragment by finding the applicable template rules and instantiating its template. The result tree is constructed by finding the template rule for the root node (possibly, \texttt{xsd:schema} when the default namespace prefix is \texttt{xsd} in XML Schema transformations) and instantiating its template.

Fig. 4 depicts partial XSLT transformation rules for XML schema documents into unordered facts. The XSLT conforms the XSLT namespace and its root element is \texttt{xsl:stylesheet} (or, \texttt{xsl:transform} is possible as well). The type of the result tree is specified as ‘text’, since the facts are represented in a plain text. The knowledge base templates above (Fig. 3B) are declared at first, and then facts assertion rules are defined for each element, attribute, and text node. The function \texttt{generate-id()} in XSLT is used to specify each node’s unique ID.

### 3.3. Test scripting

A test script, called as a rule, is defined by using the \texttt{defrule} construct. The construct consists of two parts separated by the \texttt{⇒} token. The left hand side (LHS) includes condition clauses, whereas the right hand side (RHS) defines actions executed when all the condition clauses in the LHS are satisfied. The condition clauses are interpreted as conjunctions (and relationships) in default when no other indicator is specified such as \texttt{or} and \texttt{not}. Since the condition clauses are sequentially and partially matched, it should be noted that in order to improve the expert system’s efficiency the most specific and accurate condition clauses (which will match the fewest facts) must be placed in the top of each rule’s LHS. That means the condition clauses in the top of an LHS are used to determine the context of the test, whereas those in the bottom are to identify specific conflicts in the context.

Fig. 5 illustrates a sample rule to detect anonymous types, which are locally defined in an element definition. If a knowledge base has all the five conditions (or facts), the message in the action section is printed out in a prompt window. In the condition clauses, \texttt{element}, \texttt{attribute}, and \texttt{relation} are from the templates above; and \texttt{str-index} and \texttt{isType} are functions, among which the \texttt{isType(?name)} function is a user-defined one to check whether the associated element is to define either \texttt{simpleType} or \texttt{complexType}. The user-defined function is created with the \texttt{deffunction} construct. It is noted that, instead of using hard equality (\texttt{eq} function), we used weak equality (\texttt{str-index} function) to make the rule independent of a particular namespace.

### 4. An illustrative testbed

#### 4.1. Schema design quality test

We define the quality of schema design (QoD) as how much schema appearance accords to organizational best practices on design and naming (Kulvatunyou et al., 2004). That is, it realizes look-and-feel schemas among integration parties involved – homogeneous use of structures and naming, ease to understand, uniformity to implement, etc. Hence, the QoD test verifies an XML schema

![Diagram of detailed functional model for XML document validation.](image)
against design and naming conventions, which are not necessarily a standard but agreed-upon among parties involved. The design conventions include conventions about naming, namespace, structure and declaration, documentation, versioning, etc. (Kulvatunyou et al., 2004).

4.2. Web implementation

Fig. 6 illustrates the web-based implementation of the QoD testbed, which consists of a web interface, an expert system, and a document transformation engine. We used Java Servlet Technology\(^2\) on Apache Tomcat Server\(^3\) to interface between users and internal systems; JESS for an expert system; and XSLT for document transformation. JESS is used as the underlying rule-based expert system because of its ability to use native Java functions. A user uploads XML documents to be tested and test scripts on the testbed. The testbed transforms the input XML documents into a list of facts, and then feeds the facts into the

\[ \text{Fig. 4. XSLT fragment necessary for transforming an XML schema into facts (XML2Fact).} \]

\[ \text{Fig. 5. A sample rule to detect 'anonymous type'.} \]

Java Servlet Technology\(^2\) on Apache Tomcat Server\(^3\) to interface between users and internal systems; JESS for an expert system; and XSLT for document transformation. JESS is used as the underlying rule-based expert system because of its ability to use native Java functions. A user uploads XML documents to be tested and test scripts on the testbed. The testbed transforms the input XML documents into a list of facts, and then feeds the facts into the

\[ \text{Fig. 6. A sample rule to detect 'anonymous type'.} \]

\[ \text{Fig. 5. XSLT fragment necessary for transforming an XML schema into facts (XML2Fact).} \]

\[ \text{Fig. 4. XSLT fragment necessary for transforming an XML schema into facts (XML2Fact).} \]
expert system with test scripts. Accordingly, the testbed reports test results to the user.

Fig. 7 demonstrates how to embed the JESS engine in native Java codes (the Server-side application) to upload an XML schema in JESS facts and test scripts (rules) to the JESS engine and to retrieve the execution results using the Rete object (i.e., the rule engine itself) and execute-Command function. It is observed that the JESS rule (i.e., Fig. 7A) freely uses the native Java objects/classes (e.g., java.lang.String) and methods (e.g., addElement).

Fig. 8 shows two snapshots of the QoD testbed. The first snapshot is the test execution screen of selecting test cases and uploading a target XML schema, while the second is the test result screen.

4.3. Implementation issues

When defining a test requirement and checking a schema against it, the schema type should be taken into consideration, since each test requirement has a different test scope. For example, a test requirement is ‘a control schema must identify one global element declaration’. Thus, the test cases derived from this test requirement are applied only to control schemas that include business document and interface level construct schemas described below. On the other hand, most of the test requirements (e.g., conventions about naming, namespace, etc.) can be applied across schemas. We classify four levels of schema documents as follows:

- **Lowest level construct schema.** A schema in this type defines the schema construct primitives (e.g., object entities and data types defined in CodeLists.xsd and Fields.xsd in the OAGIS specifications⁴), which are the basic building blocks of the consequent schemas. Hence, each primitive may be meaningless until it is used as part of other schemas.

- **Aggregate level construct schema.** This type schema includes logical and/or physical object components. An object component is restricted from; or composed of either primitives or other object components, or both. While the lowest level construct schema defines simple and primary data types (e.g., City, State, Country, CurrencyCode, ItemName), this defines more complex and higher level data types (e.g., Address, EmployeeInformation, Product). The Components.xsd in OAGIS falls into this type of schema. Like the lowest level construct schema, the components are seldom used independently as meaningful objects.

- **Business document level construct schema.** This type of schema defines business documents meaningfully organized with the lowest and aggregate level construct schemas. In OAGIS, the Noun parts (e.g., PurchaseOrder, BillOfMaterial, etc.) are classified into this schema type.

- **Interface level construct schema.** This kind schema is the actual BODs (business object documents) exchanged among involved parties. In OAGIS, a BOD (e.g., AddPurchaseOrder, ShowBillOfMaterial) consists of a Verb (e.g., Add, Show, Sync) and a Noun.

Another important testing issue is how to select test cases to cover all the users’ requirements of interest. In the QoD implementation, two methods of test cases selection are used: user-driven (i.e., test profile) and test requirement-driven. The first is manually done by composing a test profile, thus it is not necessary that the test cases in a test profile have similar properties (i.e., derived from the same test requirement). The second method automatically executes all the test cases derived from a test requirement selected by a user. Further, the test cases derived from the test requirements associated with that requirement should or may also be selected. To automate the selection and execution, we further need an advanced functionality by exploiting and inferring dependency and association among test requirements. This is because the conventions have concurrently and independently been amassed by scattered individual organizations without consistent management by a central authority, and as a result, many duplicate and/or similar test requirements among the

---

A (defrule testrule
   (element {e-id ?e-id} {e-name ?e-name & (isType ?e-name)})
   --
   (bind ?str (str-cat "" "A local type detected ["" ?a-val "]")
   (call ?vt addElement (new java.lang.String ?str))
   // Use the external variable "?vt" defined in Java code (as Vector object)
)

B import jess.*; // import JESS classes
-
   Rete r = new Rete();
   r.executeCommand("(reset)");
   r.executeCommand("(bind ?vt (new java.util.Vector))"); // externally define a variable (in Vector object) to temporally store and carry out test results
   r.executeCommand(jessFact); // 'jessFact' is a list of unordered facts from XML
   r.executeCommand(jessRule); // 'jessRule' is a list of rules for testing
   r.executeCommand("(run)"); // execute 'testing'
   Value v = r.executeCommand("(call ?vt toString)"); // get test results in String
   System.out.println(v.toString());

Fig. 7. Server-side Java implementation for JESS.

---

A Snapshot of selecting test cases and uploading a target document

- All of Organizations
- CCIB
- NEC
- NDR
- NST
- DAI
  - [DAGI-050]: Check for non-determinism
  - [DAGI-055]: Check for non-determinism [JESS]
  - [DAGI-100]: Check for non-determinism [DAI]
  - [DAGI-150]: Check for improper use of anonymous type
  - [DAGI-200]: Check for use of weak typing
  - [DAGI-300]: Improper use of enumeration type
  - [DAGI-550]: Enforce order/other pattern in the enumeration type
- URL

[XML Schema To Be Tested: Single File Upload]
[XML Schema To Be Tested: Multi File (zip upload)]
[XML Schema Type (*)]

Start XML Schema Test

B Snapshot of reporting test results

<table>
<thead>
<tr>
<th>The schema tested does not comply with below schema design rule(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAGI-050 [DAGI-050]: Check for non-determinism</td>
</tr>
<tr>
<td>DAGI-050 [DAGI-055]: Check for non-determinism [JESS]</td>
</tr>
<tr>
<td>DAGI-100 [DAGI-100]: Check for non-determinism [DAI]</td>
</tr>
<tr>
<td>DAGI-150: Check for improper use of anonymous type</td>
</tr>
<tr>
<td>DAGI-200: Check for use of weak typing</td>
</tr>
<tr>
<td>DAGI-300: Improper use of enumeration type</td>
</tr>
<tr>
<td>DAGI-550: Enforce order/other pattern in the enumeration type</td>
</tr>
</tbody>
</table>

The attribute 'language' is hidden.
The attribute 'languageID' is hidden.
The attribute 'unitCodeListID' is hidden.
The attribute 'unitCodeListName' is hidden.
The attribute 'unitCodeList' is hidden.

Fig. 8. Snapshots of the QoD testbed: (A) test execution and (B) test result.
conventions exist and consequently much more duplicates of test cases. A simple heuristics is to use a test requirement dependency table indicating interrelationships among test requirements. The dependency includes ‘equality/equivalence’, ‘belonging to/sub’, ‘inclusion/super’, ‘overlapped’, ‘conflict’, and ‘difference’, to each of which we can set a selection priority depending on the degree of relevance. A more close investigation on this test selection must be advanced. Upon a decision threshold (e.g., the number of associated requirements or selected test cases), test cases from more relevant requirements are selected and executed.

5. Conclusion

The consistency in data exchange specifications (and their instances) is important in success of any enterprise integration projects. The consistency can be ensured through several types of testings, e.g., grammar/syntax conformance, quality of design (QoD), information mapping, and canonical semantics. Among them, we detailed the test for schema design quality, which checks schemas against organizational conventions and best practices on design, using a rule-based expert system. As part of testbed construction, we outlined how to construct a knowledge base and how to encode test scripts. The use of JESS enabled us not only to utilize full powerful functionalities of expert systems (e.g., openness, flexibility), but also to implement a Java/web-based interface. The testbed, as a syntax-perspective tool, needs to enhance advanced testing capabilities to reason semantics behind XML documents, namely SQoD (semantic schema design quality) and SIMT (semantic information mapping test). An intelligent test selection is also a topic to be achieved.

Disclaimer

Certain commercial software products are identified in this paper. These products were used only for demonstration purposes. This use does not imply approval or endorsement by NIST, nor does it imply that these products are necessarily the best available for the purpose.

References


