Intelligent Correction and Validation Tool for XML

by

Abhishek Shivadas

B.Tech. (Information Technology), University Of Madras, Chennai, India, 2001

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

____

____

____

____

CONTENTS

1. **Introduction**
    1.1 Motivation
    1.2 Goals
    1.3 The Bigger Picture IKME
2. **Technical Background**
    2.1 XML Data Model
    2.2 The XML Schema Model
    2.3 XPath
    2.4 OOP in PERL
    2.5 Reusable Learning Objects
3. **Design Considerations**
    3.1 Choosing a set of primitives
    3.2 Control Files: - Required or not?
4. **Implementation**
    4.1 System Design
    4.2 The Parser
    4.3 CGI 1 – Generate Control Files
        4.3.1 Overview
        4.3.2 Module Design
        4.3.3 Call Methods Module
    4.4 CGI 2- Process Control File
        4.4.1 Overview
        4.4.2 Module Design
5. **Evaluation**
    5.1 Test 1 – A small XML File
    5.2 Test 2 – A complete learning object
6. **Future Work**
    6.1 n-gram Analysis
    6.2 Schema Matching
    6.3 User History

ABSTRACT:

As the World Wide Web continues to expand its reach at an increasing rate, the Extensible Markup Language (XML) has emerged as the standard of data representation and data exchange [1]. In many applications, specific schemas are designed to stipulate a strict set of rules (grammar) to which the documents are expected to conform. However, as more and more XML documents are produced and schemas continue to be reviewed, both the XML documents and schema tend to change. Current software tools provide minimal support for such data or schema changes. In this vein, we have developed software that intelligently enumerates a list of primitives that need to be applied to a XML document to make it conform to such changed schema. These primitives are consistency preserving, i.e., for a given change; they ensure that the modified XML document conforms to the schema both structurally and semantically.

1 INTRODUCTION:

1.1 Motivation:

XML has become immensely popular as the data exchange and representation format over the Web. XML is used in a wide variety of applications ranging from B2B transactions to e-Learning. Although XML in itself is self describing, most application domains tend to use schema or DTD to enforce a strict set of rules for the XML documents. The schema specifies almost everything from structure to constraints. Thus, the schema assumes the role of types in programming language or a definition table in a RDBMS.

Intelligent Knowledge Management Environment (IKME) is an ongoing project at the University of Kansas aimed at assisting the Defense Information Technology Test bed (DITT)/University after Next (UAN) in providing an advanced reach-back capability for commanders, staff, and other users who have time-critical needs.

Knowledge management, or KM, is defined as the process through which organizations generate value from their intellectual property and knowledge-based assets. KM involves the creation, dissemination, and utilization of knowledge [2]. IKME is based on the idea of using the Extensible Markup language as the data format for intelligent Knowledge Management. Intelligent because not only do we exploit the reusability and extensibility features offered by XML but also take educated decisions for the users. Knowledge creators use the environment to create learning objects, which are stored as XML documents. These learning objects are based on an XML schema developed by the "Center for Army Lessons Learned". These learning objects can in turn be reused to create lesson objects, which in turn are used to create a manual. This setup facilitates ease of creation and also faster delivery of content to the end users. Also, any modification to a learning object gets immediately reflected in lesson and manual objects. Thus, the system assumes that the schema will not change over the life of the learning objects and hence continues to produce learning objects that conform to the schema.

However, change is a fundamental aspect of persistent information and data-centric systems [1]. As the schema continues to be reviewed, content developers realize the need to modify the schema to reflect a change in the user requirements, a change in the real world operation, mistakes in the early design, or to allow for incremental maintenance [1]. These changes often make several learning objects invalid with respect to the modified schema. Current software tools provide minimal support (at best) for making the required changes to already existing XML documents.

This system is being deployed within IKME project. However, the results are applicable to any XML schema driven content production.

1.2 Sample Scenario:

Consider the following examples of how Schema changes affect XML documents. Figure 1 depicts a Schema and a XML document conforming to the schema. Changes in such a rigid schema can make several of the XML documents, invalid. Some of the changes that can affect XML documents are:

- Order Change: Assume that the schema is modified such that now the element restriction appears before the element classification. In this case the XML document would have to be modified such that restriction tag is inserted before classification.

- Tag Rename: Assume that the restriction tag in the schema is renamed to restrictions. Then the XML document has to be modified so that it is renamed and also it should retain its previous value.

- Adding/Deleting a Tag: Any Tag Deletion/Addition in the schema must be reflected upon the XML document as well.

- Changing Tag/Attribute Value: Assume that the attribute value, “Secret” is removed from the restriction tag. Then the XML document must be modified such that “Secret” is

removed as the attribute value of restriction and allocated with some other valid value. The same argument holds for a change in tag value.

1.3 Goals:

The Goals of this project are, to develop a tool that 1. Intelligently identifies such changes in a schema 2. Suggest the required changes in the XML 3. Perform these changes when prompted by the user and 4. Develop a grammar that allows such editing to be performed on XML documents.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
	<xs:element name="object">
		<xs:complexType>
			<xs:sequence>
				<xs:element name="metadata">
					<xs:complexType>
						<xs:sequence>
							<xs:element name="classification">
								<xs:complexType>
									<xs:simpleContent>
										<xs:extension base="xs:string">
											<xs:attribute name="type" use="optional">
												<xs:simpleType>
													<xs:restriction base="xs:string">
														<xs:enumeration value="Unclassified"/>
														<xs:enumeration value="Confidential"/>
														<xs:enumeration value="Secret"/>
														<xs:enumeration value="Top Secret"/>
														<xs:enumeration value=""/>
													</xs:restriction>
												</xs:simpleType>
											</xs:attribute>
										</xs:extension>
									</xs:simpleContent>
								</xs:complexType>
							</xs:element>
							<xs:element name="restriction" minOccurs="0" maxOccurs="unbounded">
								<xs:complexType>
									<xs:simpleContent>
										<xs:extension base="xs:string">
											<xs:attribute name="type" use="optional"/>
										</xs:extension>
									</xs:simpleContent>
								</xs:complexType>
							</xs:element>
						</xs:sequence>
					</xs:complexType >
				</xs:element >
			</xs:sequence >
		</xs:complexType >
	</xs:element>
</xs:schema>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<object  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="C:\Documents and
Settings\shivadas\Desktop\july7th.xsd">
	<metadata>
		<classification type="Top Secret">Unclassified</classification>
		<restriction type="Secret"/>
	</metadata>
</object>
```

***FIGURE 1*: A sample schema and One Valid XML document.**

2. TECHNICAL BACKGROUND

In the IKME, we use XML as the data format for publishing. The XML documents that are called as learning objects are constrained by a Schema provided by the US Army. The programming language used to create, modify and import these learning objects is PERL. This particular system was developed using the OOP principles in PERL. The XML data model, Schema, OOP in PERL and learning objects will be discussed in the following sections.

2.1 The XML Data Model:

XML is a textual language used for data representation and exchange on the Web. Nested, tagged elements are the building blocks of XML. Each tagged element may contain zero or more attributes or tagged sub-elements or tag less plain text data. An XML document always has an implicit order that may or may not be relevant. An XML document should conform to the "well formedness constraints" [3]. These constraints require that elements correctly nest within each other and use other markup syntax properly. Alternatively, an XML document is usually accompanied with a DTD or a Schema that is essentially a grammar for restricting the tags and attribute of the XML documents. A XML document conforming to the Schema or DTD is said to be Valid.

2.2 The XML Schema Model:

The W3C XML Schema Definition Language [4] is an XML language for describing and constraining the content of XML documents. W3C XML Schema is a W3C Recommendation. XML Schema Definition Language provides a collection of mark up constructs to write schemas.

The purpose of a schema is to define and describe a class of XML documents by using these constructs to constrain and document the meaning, usage and relationships of their constituent parts: data types, elements and their content, attributes and their values, entities and their contents and notations [4]. Schema constructs also provide for specification of default values and they also contain their documentation format.

2.3 Xpath

XPath is the result of an effort to provide a common syntax and semantics for functionality shared between XSL Transformations and XPointer. The primary purpose of XPath is to selectively address parts of an XML document [5]. XPath models an XML document as a tree of nodes. There are different types of nodes, including element nodes, attribute nodes and text nodes. XPath defines a way to compute a string-value for each type of node. Some of the XPath notations *[6]* used in this chapter are:

- *para:* Selects the *para* element children of the context node.
- * : Selects all element children of the context node.
- *para [1]:*Selects the first *para* child of the context node.
- *para [last ()]:* Selects the last *para* child of the context node.
- */doc/chapter[5]/section[2]:* Selects the second *section* of the fifth *chapter* of the *doc.*

2.4 OOP in PERL:

Perl does not provide any special syntax for defining objects, classes, or methods. Instead, it reuses existing constructs to implement these three concepts. Hence, an *object* is simply a reference or a referent, a *class* is simply a package and a *method* is only a subroutine. All objects are references, but not all references are objects. A reference will not work as an object unless its referent is specifically marked to tell Perl to which package it belongs. The act of marking a referent with a package name, and therefore its class, is known as *blessing.* The *bless* function takes one or two arguments. The first argument is a reference and the second is the package into which referent should be blessed. If the second argument is omitted, the current package is used [7].

```
$obj = { };            #Get Reference to anonymous hash.
bless ($obj);          #Bless hash into current package
bless($obj, “Tag”)     #Bless hash into class Tag
```

So, just by taking a reference to something and giving it a class by blessing, one can make objects in Perl. Thus, the author of a class needs to hide the *bless* inside a method called *constructor*, which creates and returns instances of the class. Because bless returns its first argument, a typical constructor can be as simple as:

```
package TAG
sub new { bless {}; }
```

Or, more explicitly

```
package TAG;
sub new {
        my $self = {};          #Reference to an empty anonymous hash
        bless $self, "TAG";     #Make that hash a TAG object
        return $self;           #Return freshly generated TAG
}
```

With that definition in hand, here's how one might create a TAG object.

```
$root = TAG->new;
```

2.5 Reusable Learning Objects

Traditional content development techniques face many problems in terms of extensibility or reusability. Any modification to or reuse of the content requires effort and time from the publisher. Reusable Learning Objects represent an alternative approach to content development. In this approach, content is broken down into chunks of information. Object-orientation highly values the creation of components called "objects" that can be combined or recombined in multiple contexts to create the customizable learning tracks. This is the fundamental idea behind Learning Objects. Instructional designers can build small components that can be reused a number of times in different learning contexts. One main advantage of following the Learning Objects approach is that when appropriate content already exists, they reduce development time considerably. Content creators can create customized manuals or lessons at much faster rate than using conventional techniques.

3. DESIGN CONSIDERATIONS

A need for such a system was realized midway when the schema was revised. The idea of primitives was conceived when we realized that many of the schema changes could be implemented in permutations of some these primitives. Hence, the first design issue was to choose a set of primitives. The second issue was to design a control structure that would enable the user to perform the XML data changes in offline mode. The control files can also be extended in the future to support batch processing of learning objects.

3.1 Choosing a set of primitives:

A set of primitives is chosen so that they violate neither the content model nor the invariants. The goals for choosing the primitives are:

- Consistency Preserving: These set of primitives do not affect the system's integrity in terms of well formedness of the XML document and they continue to conform to the schema.
- Complete: All schema changes that have taken place since inception of the project have been considered.

The set of primitives are listed in *Table 1*.

Sno	XML Data primitive	description
1	add-new(name, pos)	Add an empty element to position pos
2	move(pos1, pos2)	Move element form pos1 to pos2
3	delete(name)	Delete the element and its sub- elements from the XML document
4	delete-att(attname)	Delete the attribute attname from the element.
5	modify-att-value(new value)	Modify the contents of the attribute to new value.
6	modify-tag-value(new value)	Modify the contents of the tag to new value.
7	rename(old name, new name)	Rename the name of the element from old name to new name.

***Table 1*: The Set of XML Data Change Primitives**

1. add-new:

The add-new primitive adds a new element in the position specified. The position can either be after or before a particular tag. The tag is initially added as an empty tag. Value can later be filled in using the modify-tag-value. E.g.

add-new /object/metadata/object_type after /object/metadata/proponent

2. move:

The move primitive moves a tag from one position to another. Again, the source and the destination position are specified in terms of before or after a particular tag. E.g.

move /object/security [2]/portionmark after /object/security [2]/classification

3. delete:

The delete primitive deletes a tag and all its sub elements and attributes from the XML document. E.g.

delete /object/metadata/description/doc_version

4. delete-att:

The delete-att primitive deletes the attribute of a particular element and its corresponding value. E.g.

delete-att /object xmlns:xsi

5. modify-att-value

The modify-att-value primitive modifies the attribute value of an element to the value specified. The value can be suggested by the user or it defaults to an empty string. E.g.

modify-att-value /object/metadata/security [1]/classification confidential

6. modify-tag-value:

The modify-tag-value primitive modifies the content of the tag to the value specified. The value can be suggested by the user or it defaults to an empty string. E.g.

modify-tag-value /object/metadata/security/restriction unclassified

7. rename:

The rename primitive renames the tag specified to a new name. The content / attribute value remain the same. E.g.

rename /object/tracking/docversion /object/tracking/object_version

3.2 Control files – required or not?

The project was initially designed to run in batch mode with program intelligently identifying the discrepancies and modifying the learning object on the fly. But, several reviews later a need for a control structure was felt and thus the idea of a control file was conceived. The control file allows the user to input the list the changes to be performed on the learning object. Thus, the control file acts like a script with the primitives being the commands allowed. The control file also allows the program to be reapplied again and again on the same learning object until it is valid against the schema.

4. IMPLEMENTATION

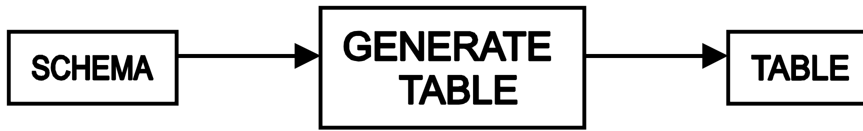
4. 1 Overview:

The system is implemented in three phases. The first phase breaks down the hierarchical structure of the schema into a flat comma delimited file. This phase has to be repeated every time the schema is changed. The second phase uploads the learning object and performs validation and primitive generation. Both the validation and primitive generation is done in a single pass through the learning object. The third phase takes the control file and the uploaded learning object and performs the data changes as directed.

4.2 Software Requirements:

i. Apache Server
ii. Perl Version 5.0 and later
iii. Additional Perl Modules
    - XML::Twig
    - CGI

4.3 System Design:



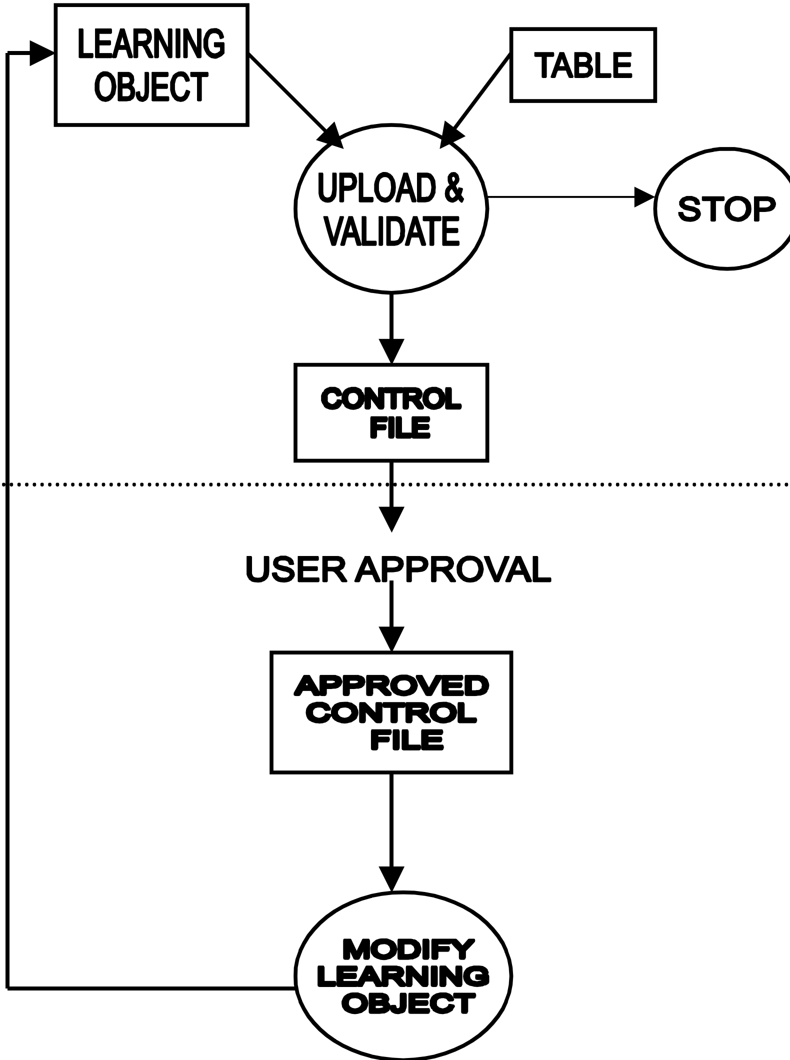


FIGURE 2: The System Design

4.4 Generate Table:

Script name: migrate_parse_schema.PL

Overview:

This script parses the schema and produces a comma delimited file as output. This file, which will henceforth be referred as table, breaks down the hierarchical structure of the schema into a flat file. The table contains an entry for each element, along with all the "immediate properties" of the element. Thus the table can be hashed and an *O (1)* access can be performed to extract all the relevant details of the context tag. *Table 2* displays the table generated by parsing the schema of *Figure 1*.

Sample Output:

Path	REQUIRED TAGS	ENUMERATION	MINIMUM	MAXIMUM	SEQUENCE	ATTRIBUTES	BOOLEAN
/object	metadata	-	1	1	metadata	-	f
/object/metadata	classification	-	1	1	classification restriction	-	f
/object/metadata/classification	-	Unclassified Confidential Secret Top Secret	1	1	-	type	t
/object/metadata/restriction	-	-	0	U	-	type	f

***Table 2*: The table produced by parsing the schema of figure 1**

These eight columns represent all the restrictions and order placed by the schema in a neat and elegant manner. The first column, based on which the table is hashed stores the path of an element from its root. The second column stores all the required sub elements. The third column stores all the legal values that the tag can comprise of. A value of - indicates that there is no restriction on the values that the tag can contain. The fourth column stores the minimum number of times the element can occur in the document, the fifth column stores the maximum. The sixth column stores the sequence of every sub – element as they occur in the schema. Hence the second row, of the table contains only '*classification*' as the required sub element but contains both '*classification and restriction*' in the sequence column, indicating the order in which they occurred in the schema. Thus the sixth column captures the inherent ordering present in the schema. The seventh column stores all the valid attributes of the element. The eighth column a Boolean, says if the element holds the enumeration within the attribute or within its text.

The table thus acts as the reference against which each element will be checked. To summarize each element will be checked if:

i. The element has all the required sub tags.
ii. The element has a legal value stored either as attribute or as text.
iii. The element has occurred at least the minimum number of times.
iv. The element has not occurred more than the maximum number of times
v. The sub elements are in the right sequence.
vi. The element has the right attribute.

4.5 Validate Learning Object

Script name: generate_control_files.PL

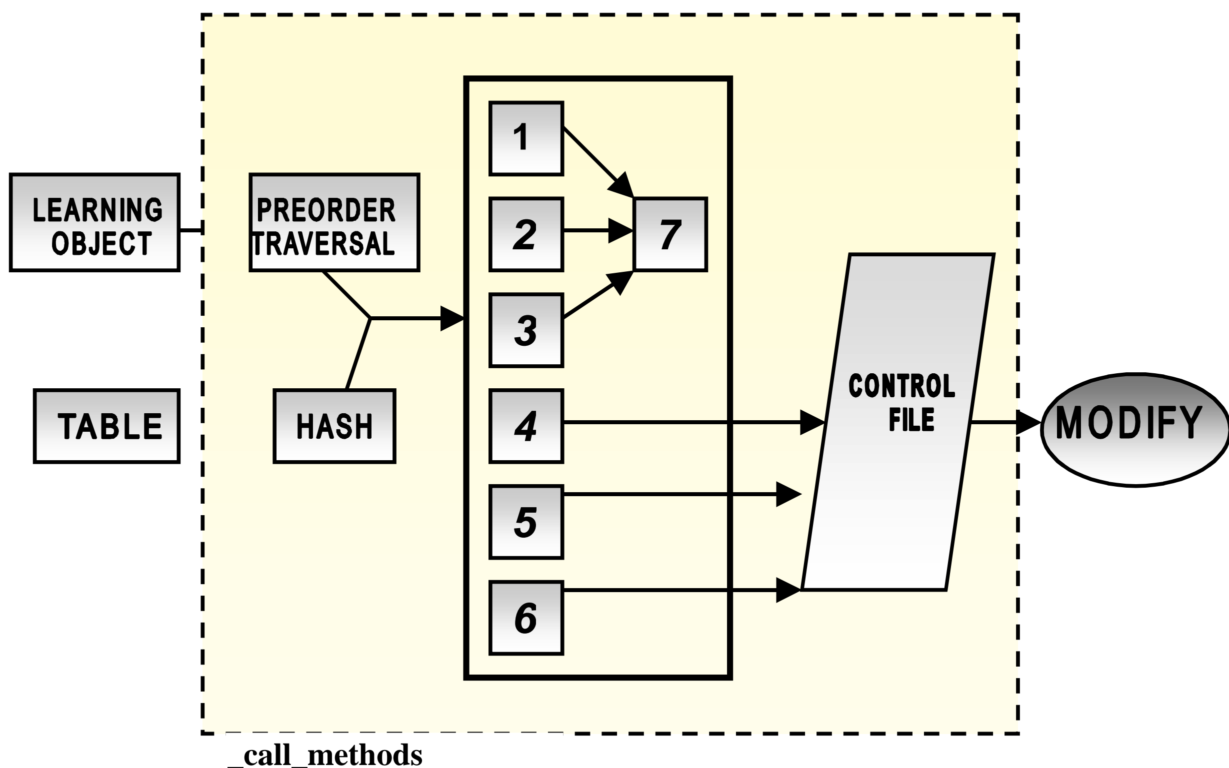
4.5.1 Overview:

This script is the backbone of the project and hence will be dealt with in detail. The user calls this program by selecting the learning object to upload and clicking the *upload* button .When called, it uploads the learning object in the working directory and begins

validating the file. The recursive traversal is performed such a way that the root is first checked against the table for inconsistencies and then each sub element is exploded and further checked. Thus, the preorder traversal checks each element for any inconsistency and inconsistencies found are further analyzed for a solution. The solution thus found is entered as an entry into the control file.

4.3.2 Module Design:

Validate Learning Objects is implemented as a collection of seven modules. Each module conducts a check for a particular constraint and, if the check fails, analyzes the type of failure and identifies and outputs the required fix to the control file. The checks are done in sequence in a single pass. Therefore, occasionally a learning object will need multiple passes to fix all problems.



_call_methods

KEY:

1.	*_check_ required*	5.	*_check_attributes*
2.	*_check_sequence*	6	*_check_enumeration*
3.	*_check_min*	7.	*_find_right_position*
4.	*_check_max*		

***FIGURE 3*: Validate Learning Objects design**

4.3.3 Call Methods Module:

The *call methods* module gets called for every element present in the learning object. This module in turn calls the six check methods in sequence. Each of the six methods tests a particular property of the tag. For example, the '*required_tag*' module checks if all the required child elements are present. If any child element is absent, the method appends an entry "add-new new-tag-name" into the control file and calls a method '*find_position*' to find the appropriate position to add the new tag. Similarly, the '*check_max*' method counts the number of times the tag has occurred in the document. If the number of times present exceeds the maximum number of occurrences for the tag, the function appends an entry "delete tag name" into the control file.

The algorithm for all the six sub modules present in the call method module will be discussed in the following sections:

4.3.3.1 Required Tag Module:

Step 1: Get all the required elements of the tag from the hash table

Step 2: If the only required element is the element itself, return

Step 3: Get all the children of the element

Step 4: Traverse the required array to see if any required element is missing

Step 5: Store all the elements missing in an array and return the array.

Step 6: Return an array of zero elements if all the required elements are present.

Title	_check_required
Pupose	Checks all the required subelements
Returns	An array of all the required elements to insert
Input	The context element
Note	None

Table 3*: Input and Output parameters of the *required tag module

4.3.3.2 Check Maximum Occurrence Module:

Step 1: get the maximum count of the element from the hash table

Step 2: If the count equals 'U' (denotes unbounded), return true.

Step 3: Get a count of the previous siblings with the same tag name.

Step 4: If the count is greater than the maximum count, return false.

Title	_check_max_occur
Pupose	Checks if the element has occurred more than the maximum number of times permissible.
Returns	A Boolean
Input	The context element
Note	Since there are no types in PERL this function returns a string "VIOLATION" for false and a string "OK" for true.

Table 4*: Input and Output parameters of the *check maximum occurrence module

4.3.3.3 Check Minimum Occurrence Module:

Step 1: Get the minimum count from the hash table.

Step 2: Get the parent of the element.

Step 3: Count the number of times the element has occurred within the parent

Step 4: If this is less than the minimum count, return false.

Title	_check_min_occur
Pupose	Checks if the element has occurred more than the maximum number of times permissible.
Returns	A Boolean
Input	A single element

Table 5: Input and Output parameters of the *check minimum occurrence module*

4.3.3.4 Check Attribute Name Module:

Step 1: Get the list of valid attributes of the element from the hash table.

Step 2: If the element does not have an attribute return an empty array.

Step 3: Get all the attributes of the element into a list.

Step 4: Using the two lists (from above) check if all the attributes are valid.

Step 5: Push the invalid attributes into an array and return.

Title	_check_attributes
Pupose	Checks if the attributes of element are valid or not.
Returns	An array of invalid attributes.
Input	A single element

Table 6*: Input and Output parameters of the *check attribute name module

4.3.3.5 Check Enumeration Module:

Step 1: Get the list of valid enumeration of the element from the hash table.

Step 2: Ascertain if the attribute or tag stores the enumerated values.

Step 3: Check the value on the tag or attribute with the valid list

Step 4: If the value is found return *true* otherwise return *false*.

Title	_check_enumeration
Pupose	Checks the enumerated value of the element
Returns	A Boolean
Input	A single element
Note	The assumption made is that empty string is a valid enumeration value.

Table 7*: Input and Output parameters of the *check enumeration module

4.3.3.6 Check Sequence Module:

Step 1: Get the parent of the element.

Step 2: Get the *right sequence list* for the parent from the hash table.

Step 3: Get the previous sibling and the next sibling of the element.

Step 4: Split the list from Step 2, into predecessor's sibling list and successor's sibling list.

Step 5: Check if previous sibling occurs in the predecessor's sibling list.

Step 6: Check if next sibling occurs in the successor's sibling list.

Step 7: If either of the checks performed in Step 6 and Step 7 fails, return *false*. If both the check succeed return *true.*

Title	_check_sequence
Pupose	Checks if the element has occurred in the right order within its parent.
Returns	A Boolean
Input	A single element
Note	None

Table 8*: Input and Output parameters of the *check sequence module

4.3.3.7 Find Position Module:

The '*find position*' module is invoked by the '*call methods*' module whenever a new element needs to be inserted or if the order of an element needs to be altered. This method identifies the right position to insert the new/existing element. The method returns a string that contains either a *'before'* or *'after'* followed by one of the previous siblings or one of the subsequent siblings name. This concept of *before* or *after* is inspired from XUPDATE specifications. The algorithm is explained below:

Step 1: Get the parent of the element.

Step 2: Get a list of the parent's children.

Step 3: Get a list of all the previous siblings from the children list.

Step 4: Get a list of all the subsequent siblings from the children list.

Step 5: Get the *correct sequence* list from the hash table.

Step 6: Get a list of the correct previous siblings.

Step 7: Get a list of the correct subsequent siblings.

Step 8: Group the correct sequence list from Step 5 into priority classes.

Step 9: Scramble the list of Step 3 and Step 5 into a single linear vector. The vector contains elements from both the list arranged in an alternative order.

```
Step 10: For each priority class
            For each element from the scrambled vector
               If element belongs to current priority class
                  If element belongs to the list generated in Step 6
                     Return "after element-name"
                  End If
                  If element belongs to list generated in Step 7
                     Return "before element-name"
                  End If
               End If
            End For
         End For
```

Algorithm: Determining the correct position to insert an element.

Example 1:

The main objective behind the algorithm is to find the closet neighbor for the element based on a true sequence. Consider the following scenario, with element A being the context element and whose correct position needs to be determined.

True Sequence: A B C D E F

Occurred Sequence: B D C E A F

Following the algorithm from Step 3 onwards:

- Ø 3. List of previous siblings B D C E
- Ø 4. List of subsequent siblings F
- Ø 6. List of correct sequence previous siblings: NULL
- Ø 7. List of correct sequence subsequent siblings: B C D E F
- Ø 8. The priority is based on the distance of the elements from the context element. In the example since B is the closest in the correct sequence it holds the highest priority followed by C and so on. Thus the priority sequence is {B} {C} {D} {E} {F}
- Ø 9. Scrambling is the process of arranging one element each from the list of previous siblings (B D C E) and subsequent siblings (F) in alternative order. Thus the scrambled vector is E F C D B
- Ø 10. Iterating through the '*For*' loop, a positive match for B is found. Since, B occurs in the list of *correct sequence subsequent siblings* the statement *'before B'* is returned.

Example 2:

With the same right and occurred sequence, a run through the algorithm with C being the context element and whose correct position needs to be determined.

- Ø 3. List of previous siblings B D
- Ø 4. List of subsequent siblings E A F
- Ø 6. List of correct previous siblings A B
- Ø 7. List of correct subsequent siblings D E F
- Ø 8. The priority class sequence is: {B,D} {A, E} {F}
- Ø 9. The scrambled sequence is: D E B A F
- Ø 10. Match found for D, Since D belongs to *correct subsequent list*, *'before D'* is returned.

4.4: Modify the Learning Object

Script Name: process_control_file.PL

4.4.1: Overview:

The '*validate learning object*' program sets a Boolean when an inconsistency is encountered and invokes the '*modify learning object*'. Thus, modify learning object is called only if any change needs to be performed on the learning object. The program assumes the control file as input and performs the changes as directed by the control file. The schematic diagram is shown below:

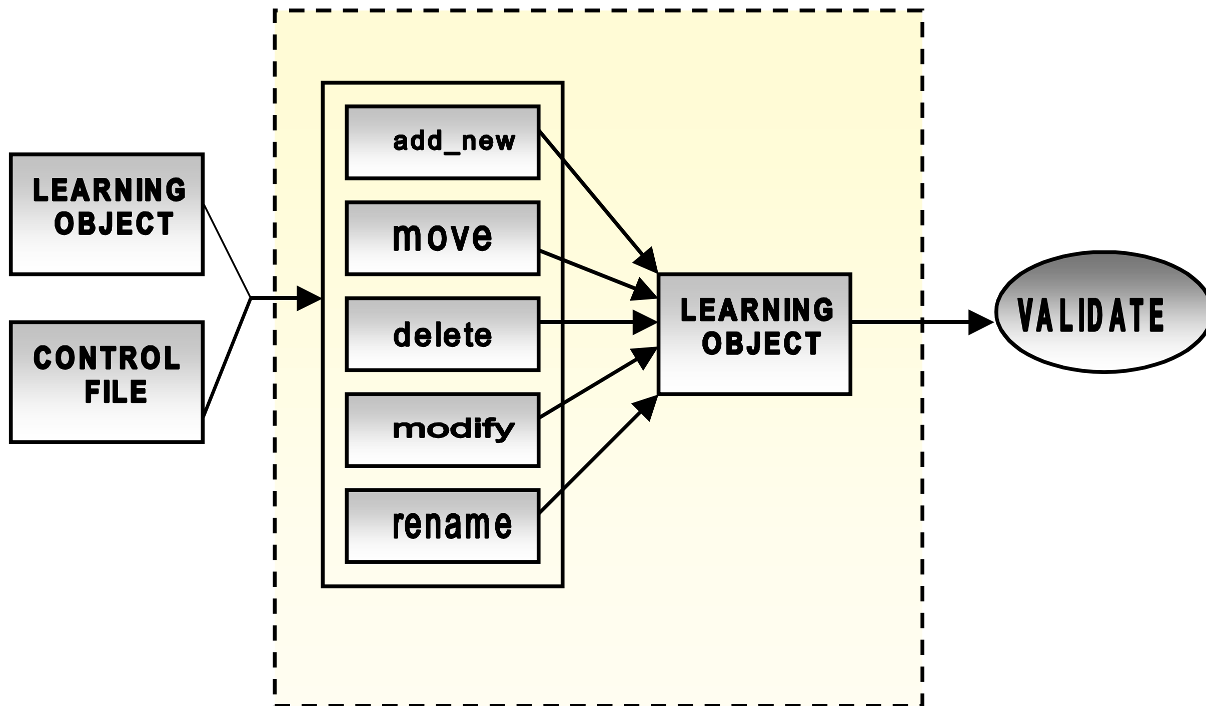


***FIGURE 4*: Modify Learning Object design**

As shown in the block diagram, this program has a very simple logic and blindly follows what it is directed to do by the control file. The program assumes that the control file is in a specific format. The command, assumed to be the first field should be among the operations discussed in section 3.1. The source should be the second field and the optional third field must be destination delimited by commas. A sample control file is shown below.

```
rename,/object/tracking/status,/object/tracking/action_status
delete,/object/tracking/action/docversion
move,/object/action/actiondate,before /object/action/actor
delete,/object/tracking/action/status
move,/object/tracking/actiondesc,after /object/tracking/actor
move,/object/content/block/list,after /object/content/block/para[2]
delete-att,/object,xmlns:xsi
add-new,/object/metadata/object_type,after /object/metadata/proponent
```

***FIGURE 5*: A Control File**

4.4.2 Module Description:

Each primitive is implemented in a simple module. The modules are

- Ø add-new
- Ø delete
- Ø move
- Ø rename
- Ø modify

The *add, delete, modify* and *rename* modules are fairly straightforward. The *move* module is more complex. The move statements can fall into potentially four categories. They are

a. 1: 1 Category:

This situation arises when a source element needs to be moved *'before'* or *'after'* a destination element. This is the most common occurrence and is always the output of the *find position algorithm.*

b. 1: N Category:

This situation arises when a source element needs to be moved *'before'* or *'after'* multiple destination elements. This is possible because the XPath of the destination

satisfies more than one element. In this case the program selects the first destination element and moves the source '*before'* or *'after'* it.

c. N: 1 Category:

This situation arises when a multiple source elements need to be moved 'before' or 'after' a destination element. This is possible because the XPath of the source satisfies more than one element. In this case all the sources are moved '*before*' or *'after'* the destination.

d. N: N Category:

This situation arises when multiple source elements need to be 'before' or 'after' multiple destination elements. This implemented just the way as N: 1 is implemented with the chosen destination being the first among the possible destinations.

The scenarios are summarized in Table 9.

SCENARIO	DESCRIPTION	EXAMPLE	IMPLEMENTED
1:1	A single source element to be moved *'before'* or '*after*' a single destination element	move,/object/tracking/actiondesc,after /object/tracking/actor[1]	Yes
1:N	A single source element to be moved *'before'* or *'after'* multiple destination elements.	move,/object/tracking/actiondesc,after /object/tracking/actor. The source tag is moved *'before'* or *'after'* the first destination tag.	Yes
N:1	Multiple sources to be moved *'before'* or *'after'* a single destination element.	move,/object/tracking/actiondesc,after /object/tracking/actor[1]	Yes
N:N	Multiple sources to be moved *'before'* or *'after'* multiple destination elements	move,/object/tracking/actiondesc,after /object/tracking/actor. All the source tags are moved '*before'* or '*after'* the first destination tag.	Yes

Table 9: The four scenarios possible for a move statement

5. EVALUATION

The software has the ability to identify changes in the schema and is also capable of performing modifications to the learning objects. The changes that the software is capable of identifying are presented in the Table 10 below. The second column of the table lists the primitives that are generated when such changes are recognized.

SNO	TYPE OF CHANGE	PRIMITIVES
1	Adding a new tag	add-new
2	Deleting a tag	delete
3	Change the sequence of a tag	move
4	Modifying the contents of a tag	modify
5	Modifying the contents of an attribute	modify-att
6	Deleting an attribute	delete-att

Table 10: Types of changes that the software can identify and their respective primitives

There are certain kinds of changes in the schema that the software cannot identify. These kinds of changes arise when a tag is moved from within its local parent to a different parent. These are changes that are generally more than one level deep. The software also does not have the ability to recognize tag renames. However, the primitives for add and remove can effect a ‘rename’ and get generated as a result of renaming. Thus, apart from the table generated automatically, the user can create a table of changes they want, that is input to the system. Table 11 lists the changes that software cannot recognize. The second column of the column states the primitives designed for handling these changes.

SNO	TYPE OF CHANGE	PRIMITIVES
1	Tag Renames	rename
2	Tag moves from one parent to another	move

Table 11: Types of changes that the software cannot identify and their respective primitives

The demo is viewable at http://onlineacademy.org/ikme/demoV1.0/demo_menu.html. The Import link on the demo page brings up the form for Import learning objects.

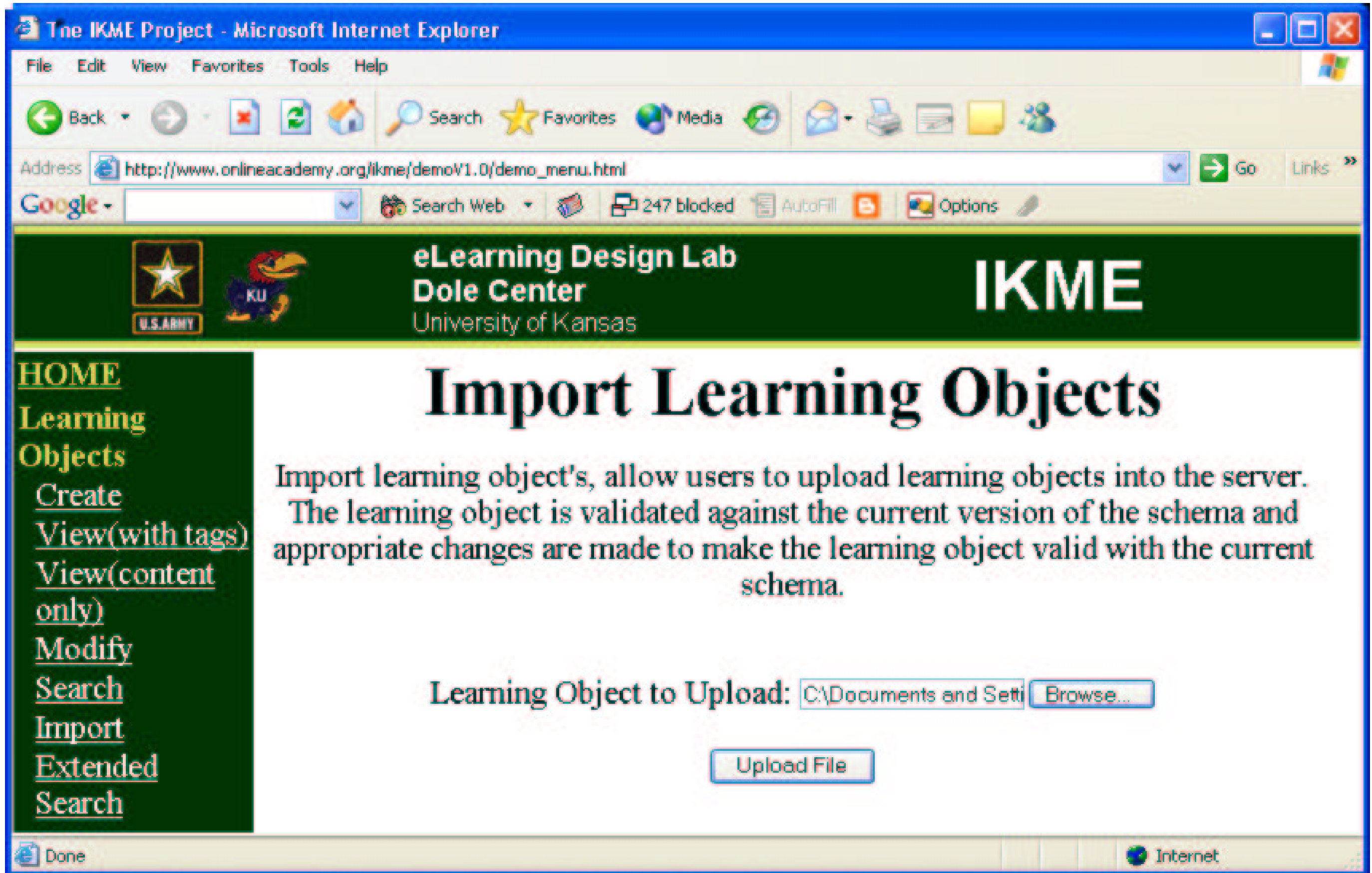


***FIGURE 6*: Homepage for Import learning Objects**

Test 1: A small XML file

Test one is conducted with a small XML file. The XML file has *object* tag as the root element followed by empty elements, *metadata* and *content*. This XML file (*Figure 7)* is used for pure test purpose and does not contain any content to qualify as a useful learning object.

```
<?xml version="1.0" encoding="UTF-8"?>
<object>
      <metadata/>
      <content/>
</object>
```

***FIGURE 7*: A XML test document.**

This learning object has the following problems.

The following are the suggested changes in the Learning Object you wish to upload

	COMMAND	SOURCE	DESTINATION
☑	move	/object/metadata/description/topic/object_c	after /object/metadata/description/service
☑	add-new	/object/tracking	after /object/metadata
☑	add-new	/object/views	after /object/metadata
☑	add-new	/object/reconstructions	before /object/content
☑	add-new	/object/metadata/security	before /object/metadata/source
☑	add-new	/object/metadata/source	after /object/metadata/security
☑	add-new	/object/metadata/description	after /object/metadata/source
☑	add-new	/object/metadata/applicability	after /object/metadata/description
☑	add-new	/object/metadata/file	after /object/metadata/applicability

Correct them!

Internet

***FIGURE 8*: Suggested changes for the XML document.**

These commands are shown to the user for approval (see Figure 8). The user may use the check boxes to turn on or off any or all the suggested changes.

The first statement belongs to the list of unidentified changes (Table 11) and can be ignored in this case. The statements thereafter specify the position the new tags that must be added and their positions so that the document will validate against the learning object schema. Statements 2, 3 and 4 specify the required tags that need to be added as children of the root element. The rest specify the sub elements that need to be added to element, *metadata*.

The *correct them!* Button calls *modify learning objects*. When it completes it produces the output shown in Figure 9.

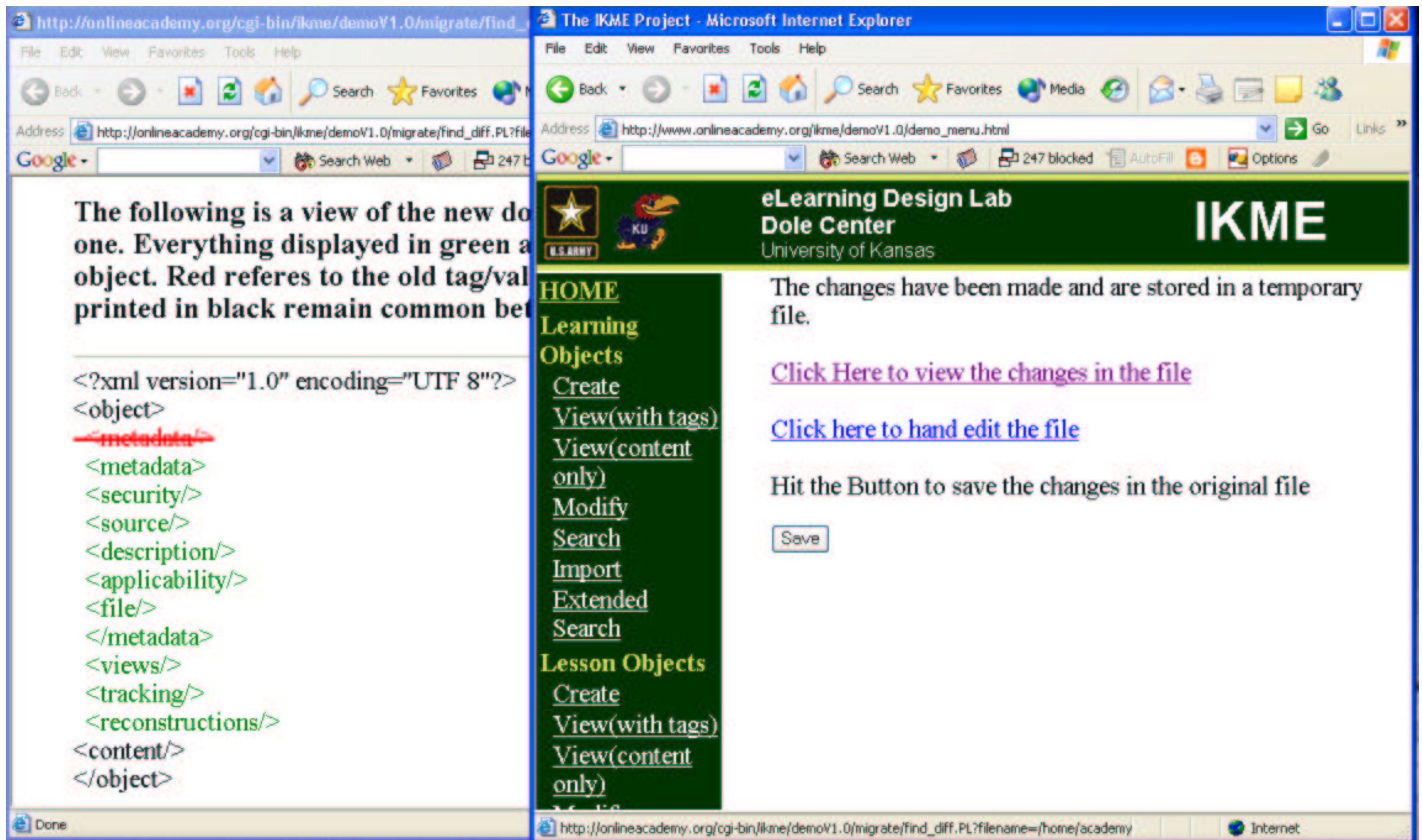


***FIGURE 9*: Viewing the changes done on the document**

The window on the right is the actual output. But by clicking the first link, the window to the left appears. The left window shows in colored font the changes that have taken place. Removals appear in red and additions in green. We can see that *metadata* has gone from an empty tag, <metadata/> to a pair of tags with other required tags *view* and *tracking* in between.

The save button invokes *validate learning objects* again that attempts to validate the modified XML file. The output produced is shown in Figure 10.

The following are the suggested changes in the Learning Object you wish to upload

	COMMAND	SOURCE	DESTINATION
☑	move	/object/metadata/description/topic/object_c	after /object/metadata/description/service
☑	add-new	/object/metadata/security/classification	before /object/metadata/security/portionma
☑	add-new	/object/metadata/security/portionmark	after /object/metadata/security/classificatior
☑	add-new	/object/metadata/description/object_categc	before /object/metadata/description/propor
☑	add-new	/object/metadata/description/proponent	after /object/metadata/description/object_c
☑	add-new	/object/metadata/description/object_type	after /object/metadata/description/proponei
☑	add-new	/object/metadata/description/vital_record	after /object/metadata/description/object_ty
☑	add-new	/object/metadata/description/topic	after /object/metadata/description/vital_rec
☑	add-new	/object/metadata/applicability/branch	before /object/metadata/applicability/envirc
☑	add-new	/object/metadata/applicability/environment	after /object/metadata/applicability/branch
☑	add-new	/object/metadata/file/documentscheme	before /object/metadata/file/filename
☑	add-new	/object/metadata/file/filename	after /object/metadata/file/documentscheme
☑	add-new	/object/metadata/file/fileformat	after /object/metadata/file/filename
☑	add-new	/object/metadata/file/filelocation	after /object/metadata/file/fileformat
☑	move	/object/views	after /object/tracking
☑	add-new	/object/tracking/action	after as_first_child
☑	move	/object/tracking	before /object/views

Correct them!

FIGURE 10: **More changes**

Thus, the software continues to revalidate the document and suggest changes, until the document is valid. In this case, a valid but empty learning object is produced.

Test 2: Complete Learning Object

Test two is conducted with a learning object (*urban_triad-00-001.xml*) provided by the US Army in the month of June 2003. However, in July 2003 the schema was revised making the learning object invalid. The software now identifies some of the changes and

corrects them automatically to reflect the modifications made to the schema. The learning object is shown in Figure 11.


```xml
<?xml version="1.0" encoding="UTF-8"?>
<object type="Content - Doctrine" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <metadata>
        <security>
            <portionmark>U</portionmark>
            <classification type="US Security">Unclassified</classification>
            <restriction type=""/>
        </security>
        <security>
            <portionmark>U</portionmark>
            <classification type="US Security">Unclassified</classification>
            <restriction type=""/>
        </security>
        <header level="object">
            <header/>
            <title>Overview of the Urban Triad</title>
            <subtitle/>
        </header>
        <source type="">
            <source_document>
                <header>This information was extracted from:</header>
                <title/>
                <publication>JP 3-06</publication>
                <edition/>
                <publication_date>May 2002</publication_date>
                <source_comment/>
            </source_document>
            <actual_event>
                <event/>
                <event_date/>
                <location/>
                <about_whom/>
                <event_comment/>
            </actual_event>
        </source>
        <description>
            <obj_description/>
            <service/>
            <proponent>J7</proponent>
            <doc_version/>
            <vital_record/>
            <rev_date/>
            <topic>
                <topic_node>Urban Triad</topic_node>
                <topic_acronym/>
                <object_category>Characteristics</object_category>
                <taxonpath ref_taxonomy="Environment" level1="Urban Environment" level2="Urban Triad" level3="" level4="" level5="" level6=""/>
            </topic>
            <cross_ref>
                <crossref_list>JDEIS</crossref_list>
                <crossref_number/>
                <crossref_title/>
            </cross_ref>
        </description>
        <applicability>
            <indicator type=""/>
            <interoperability level="Joint"/>
            <branch/>
            <echelon/>
            <unittype/>
            <executor/>
            <environment>Urban Environment</environment>
        </applicability>
        <file>
            <documentscheme>CALL IO</documentscheme>
            <filename>urban_triad-00-001.xml</filename>
            <fileformat>text/xml</fileformat>
            <filelocation/>
        </file>
          </metadata>
```


***FIGURE 11*: The *metadata* tag of the learning object urban_triad-00-001.xml**

The following are the suggested changes in the Learning Object you wish to upload

	COMMAND	SOURCE	DESTINATION
☑	rename	/object/metadata/description/doc_version	/object/metadata/description/object_versio
☑	move	/object/metadata/description/topic/object_c	after /object/metadata/description/service
☑	rename	/object/tracking/action/docversion	/object/tracking/action/object_version
☑	rename	/object/tracking/action/status	/object/tracking/action/action_status
☑	delete-att	/object	xmlns:xsi
☑	move	/object/metadata/security[1]/portionmark	after /object/metadata/security[1]/classificat
☑	move	/object/metadata/security[1]/classification	before /object/metadata/security[1]/portionr
☑	modify-att-value	/object/metadata/security[1]/classification	
☑	delete	/object/metadata/security[2]	
☑	move	/object/metadata/security[2]/portionmark	after /object/metadata/security[2]/classificat
☑	move	/object/metadata/security[2]/classification	before /object/metadata/security[2]/portionr
☑	modify-att-value	/object/metadata/security[2]/classification	
☑	add-new	/object/metadata/description/object_type	after /object/metadata/description/propone
☑	move	/object/metadata/description/proponent	after /object/metadata/description/service
☑	delete	/object/metadata/description/doc_version	
☑	move	/object/metadata/description/vital_record	before /object/metadata/description/rev_da
☑	move	/object/metadata/description/topic/topic_ac	after /object/metadata/description/topic/top
☑	delete	/object/metadata/description/topic/object_c	
☑	move	/object/metadata/description/topic/taxonpa	after /object/metadata/description/topic/top
☑	delete	/object/tracking/action/docversion	
☑	move	/object/tracking/action/actiondate	before /object/tracking/action/actor
☑	move	/object/tracking/action/actor	after /object/tracking/action/actiondate
☑	delete	/object/tracking/action/status	
☑	move	/object/tracking/action/actiondesc	after /object/tracking/action/actor
☑	move	/object/content/block/list	after /object/content/block/para[2]
☑	move	/object/content/block/para[2]	after /object/content/block/title

Correct them!

FIGURE 12: **Changes suggested by the software**

The first four commands display the tags to be renamed and moved from one parent to another. These four commands belong to the list of unidentified changes (Table 11). The commands after the first four are intelligently generated. The *security* element under *metadata* has the elements '*portionmark' and 'classification'* in the incorrect order. Hence, the second and the third move statements are generated. In the same vein, the '*classification'* tag has an unidentified value *US-Security* as the attribute . The statement *'modify-att-value /object/metadata/security[1]/classification,'* would alter the attribute value of *classification* to an empty string. The other interesting statement is *delete /object/metadata/security[2]/*. The schema was revised such that *security* occurs only once under *metadata*. Hence, the program allowed the first *security* element to pass though unscathed and raised a violation for the second *security* element.

The statement *delete '/object/tracking/action/doc_version'* is generated because the element '*doc_version'* has been renamed to '*object_version'* in the modified schema and the validator did not identify the tag '*doc_version'*. Since the execution the control file takes place sequentially the program would initially rename the tag '*doc_version'* to '*object_version'* (statement 1) and then try to delete '*doc_verision'* when it encounters the delete command. The delete command would obviously fail and hence would be ignored. The figure also displays several move statements showing that the order of many elements have altered in the schema. Users can select the statements they wish to execute by checking the check boxes.

The *correct them* button invokes *modify learning object* and the statements that have been approved by the user will be executed. The next screen gives the user the option to view changes by clicking the *view changes* button or hand edit the file using by clicking the *hand edit link.* The hand edit link throws up a text box with the entire learning object in it. This option gives the user to delete/modify the element as he/she chooses to do. Thus by un-checking the *delete security[2]* command and by deleting the first *security* tag, the user can retain the contents of *security[2]* instead of *security[1].* The screen shot of hand edit is shown below.

eLearning Design Lab
Dole Center
University of Kansas

The changes have been made and are stored in a temporary file.

Click Here to view the changes in the file

Click here to hand edit the file

Hit the Button to save the changes in the original file

Save


```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v5 rel. 3 U (http://www.xmlspy.com) by
(dole) --><object type="Content - Doctrine">
  <metadata>
    <security>
      <classification type="">Unclassified</classification>
      <portionmark>U</portionmark>
      <restriction type=""/>
    </security>
    <header level="object">
      <header/>
      <title>Overview of the Urban Triad</title>
      <subtitle/>
    </header>
    <source type="">
      <source_document>
        <header>This information was extracted from:</header>
        <title/>
        <publication>JP 3-06</publication>
        <edition/>
        <publication_date>May 2002</publication_date>
        <source_comment/>
      </source_document>
      <actual_event>
        <event/>
        <event_date/>
        <location/>
        <about_whom/>
        <event_comment/>
      </actual_event>
    </source>
```


***FIGURE 13*: The Edit box that allows hand editing.**

By clicking the view changes link, the modifications that have been made on the learning object can be viewed. The changes are in green and the statements that have been changed are displayed in red and have been struck out.

The Perl script that is invoked when the *view changes link* is clicked calls the Linux, *'diff'* command. The *'diff'* command takes in two files and produces the difference between the two files as output. The Perl script parses the output and displays the older version of the file in red and newer version of the file in green. A screen shot of clicking on the view changes link is presented in Figure 14.

```
<?xml version="1.0" encoding="UTF 8"?>
<! edited with XMLSPY v5 rel. 3 U (http://www.xmlspy.com) by sbatheja (dole) ><object
type="Content Doctrine" xmlns:xsi="http://www.w3.org/2001/XMLSchema instance">
<! edited with XMLSPY v5 rel. 3 U (http://www.xmlspy.com) by sbatheja (dole) ><object
type="Content Doctrine">
<metadata>
<security>
 <classification type="">Unclassified</classification>
<portionmark>U</portionmark>
<classification type="US Security">Unclassified</classification>
<restriction type=""/>
</security>
<security>
<portionmark>U</portionmark>
<classification type="US Security">Unclassified</classification>
<restriction type=""/>
</security>
<header level="object">
<obj_description/>
<service/>
<proponent>J7</proponent>
<doc_version/>
 <object_category>Characteristics</object_category>
 <object_type/>
 <object_version/>
<vital_record/>
<rev_date/>
<topic>
<topic_node>Urban Triad</topic_node>
<topic_acronym/>
<object_category>Characteristics</object_category>
<taxonpath level1="Urban Environment" level2="Urban Triad" level3="" level4="" level5="" level6=""
ref_taxonomy="Environment"/>
</topic>
<cross_ref>
</metadata>
```

The changes have been made and are stored in a temporary file.

Click Here to view the changes in the file

Click here to hand edit the file

Hit the Button to save the changes in the original file

Save

***FIGURE 14*: Viewing the changes made to the learning object.**

The above picture shows the modifications that have been made. Some of the interesting changes are:

1. The attribute for classification is no longer “US – Security”.
2. The order of ‘*classification’* and ‘*portionmark’* has been changed.

3. The tag '*doc_verision'* has been renamed to '*object_version*'.
4. The tag '*object_category'* has moved to a new spot.

By continuing the process a valid '*urban_triad-00-001.xml'* will be produced. This file can later be moved to a central location and hence the change(s) will be immediately reflected upon any lesson/manual object using the particular learning object that has been modified.

6. <u>FUTURE WORK</u>

The software successfully meets all the goals initially set. However, this is only a beginning and several modifications to this tool are possible. Some of them are

6.1 n-gram analysis:

The current software assumes that an empty string is a valid enumeration value. This may not be true in many scenarios and hence will result in invalid documents being produced. What is required is an n-gram analysis on the enumerated values. For example, the tag '*restriction'* had '*classification'* as a valid enumerated type, until it was changed to '*classifications*' in the revised schema. This resulted in making several learning objects invalid. The current tool identifies '*classification'* as an invalid entry and suggests an empty string as the replacement instead of *'classifications'*, resulting in loss of data. The software cannot identify tag renames. An n-gram analysis can help identifying renamed tags.

6.2 Schema comparisons:

The current software is intelligent in identifying changes that are local to particular element and does not take into consideration the complete environment of the element. That is sequence changes only within 1 level deep can be identified. For example:The software needs external intervention to realize that the tag '*object_category'*, initially under '*topic'* has moved one level up to be under *description.*

move ../description/topic/object_category after ../description/service

These kinds of changes can be identified only by comparing two schemas.

6.3 User History:

Some intelligent delete statements can be produced by keeping track of user's history. Quoting from the previous example, a delete statement was issued for '*security [2]'*. By keeping track of user's history, an analysis function can be applied to both

‘*security [1]’* and ‘*security [2]’* to decide between the more relevant one and a delete statement can be issued for the less relevant tag.

6.4 Batch Processing:

When a schema changes, all previously created learning objects need the same or similar updates. Creating an update script that updates all objects similarly would greatly aid the end user by requiring them to approve the changes once for the whole collection.

REFERENCES

[1] Hong Su, Diane Kramer, Li Chen, Kajal Claypool, Elke A. Rundensteiner, *"XEM: Managing the Evolution of XML Documents,"* Worcester, MA 01609–2280

[2] Knowledge Management Definition,
http://www.commerce-database.com/knowledge-management.htm1

[3] Roy Goldman, Jason McHugh, Jennifer Widom, *"Semistructured Data to XML: Migrating the Lore Data Model and Query Language,"* Stanford University

[4] XML Schema RequirementsW3C Note 15 February 1999,
http://www.w3.org/TR/1999/NOTE-xml-schema-req-19990215

[5] XML Path Language (XPath) Version 1.0, W3C Recommendation 16 November 1999 http://www.w3.org/TR/1999/REC-xpath-19991116

[6] XML, The Perl Way, http://www.xmltwig.com

[7] Larry Wall, Tom Christiansen, Jon Orwant, Programming Perl – Third Edition, O'Reilly and Associates, July 2000