Novi Sad J. Math.
Vol. 33, No. 2, 2003, 111–118

VALIDATION OF XML BIBLIOGRAPHIC RECORDS IN JAVA ENVIRONMENT

Mirjana Zeremski[1], **Dušan Surla**[1,2]

Abstract. One of the main library network function is so–called shared cataloguing, i.e. the exchange of bibliographic records on national and international level. In this paper we present a system for creating valid bibliographic records based on XML technology, as a basis for exchange of these records within default library network. To perform the exchange of XML bibliographic records it is necessary to ensure these records be valid against the adopted document type definition. The validation will be done against a schema written in XML Schema Language, based on the international standard for recording the bibliographic data in machine-readable cataloguing format – UNIMARC. The system for creating valid XML bibliographic records will be implemented in Java environment.

AMS Mathematics Subject Classification (2000): 68P15, 68P05

Key words and phrases: XML Schema Language, UNIMARC, validation of XML bibliographic records

1. Introduction

UNIMARC (UNIversal Machine Readable Cataloguing) is international standard for recording the bibliographic units in machine–readable form. XML (Extensible Markup Language) technology consists of services and tools which provide maintenance for structuring, presenting and exchanging of electronic documents.

The conversion process of bibliographic UNIMARC records into XML documents, presented in paper [4], provides the records whose form is suitable for electronic interchange. A single bibliographic record (i.e. the one corresponding to the single bibliographic unit) is converted into XML document stored in a single file. However, to use a bibliographic record in a system of shared cataloguing it is necessary to provide its compliance with a format adopted among the institutions which participate in such a system. In this paper the UNIMARC format is implied format of bibliographic records.

[1]University of Novi Sad, Faculty of Science and Mathematics, Department of Mathematics and Informatics, Trg D. Obradovića 4, 21000 Novi Sad, Serbia and Montenegro, e-mail: mirjanaz@im.ns.ac.yu (M. Zeremski), surla@uns.ns.ac.yu (D. Surla)

[2]The research is supported by the Ministry of Science, Technologies and Development of the Republic of Serbia (Project No. 1875)

In [3], the XML schema of bibliographic records, built in accordance with UNIMARC format is presented. Any instance of this schema is an XML bibliographic record. So, to check the correctness of XML bibliographic records obtained in the conversion process, it is enough to validate them against the mentioned schema using one of the editors specialized for the work with XML documents.

However, if it is needed to validate a large number of XML documents (maybe thousands of them), as is the case in a library network, the mentioned editors would be very slow and non–efficient and also there would not be the possibility for further automatic processing of information on the obtained errors. Another disadvantage of existing editors, related to further processing of error information and correcting of errors consists of the following: after the validation process there is information about exactly one error – the first one detected. Furthermore, the usage of existing editors requires the good knowledge of XML, which is not so simple for a librarian whose job is processing and verifying records. On the basis of the above mentioned, it would be desirable to develop a specialized editor for XML bibliographic records which would enable the validation of these records and its usage would not require any knowledge of the XML technology.

Validation of XML bibliographic records against the schema described in [3], defining their structure and content in such a way that they correspond to UNIMARC format, should comprise the following checks related to the structure and content:

- Whether all mandatory fields are stated in the record, and within the stated fields all mandatory subfields, as well as indicators in fields for which they are defined;
- Whether the fields, or subfields that repeat in a record, can really be repeated according to the definition of UNIMARC format;
- Whether the fields, or subfields stated in a record really exist according to the definition of UNIMARC format;
- If fields **421**, **423** or **469** appear in a record and within these the subfield 1 with secondary fields and whether they are according to the definition of the format really secondary fields;
- If in a record, and within a field, one or both indicators appear with their values, are they according to UNIMARC format really defined as indicators for that field and the content;
- Whether the values written in the indicators and subfields are correct according to the format definition.

2. Implementations environment

The system for validation of XML bibliographic records is implemented in Java environment. It is done by the ready–made package *Sun Multi–Schema*

XML Validator – MSV, developed by the company ***Sun Microsystems*** [1]. Package MSV implements interface JARV (Java API for RELAX Verifiers). JARV consists of three components: *VerifierFactory*, *Schema* and *Verifier*.

The *VerifierFactory* interface is the main interface between the implementation and users application. It contains a method to compile a schema into a *Schema* object. The *Schema* interface is the internal representation of the schema. Also, this interface has a method to create a new *Verifier* object. The *Verifier* interface represents a so–called "validator"; it has a *Schema* object in it and it validates documents by using that schema.

The usage of JARV interface usually implies the following steps:

- Step 1: create VerifierFactory
- Step 2: compile a schema
- Step 3: create a Verifier
- Step 4: perform validation

It is possible to perform validation in two ways. One way is to call the *verify* method of the *Verifier* class. This method accepts a DOM (Document Object Model) tree, file, URL, etc., and returns the yes/no answer, depending on whether the checked XML document is valid or not. In the case that XML document is not valid, it is important to get detail information on detected errors, which is possible by using the *setErrorHandler* method of the *Verifier* class. It is important to emphasize that by the validation with *verify* method of *Verifier* class, all existing errors in the processed document will be found and reported immediately.

The validation via SAX (Simple API for XML) interface is supported in two ways. The first possibility is to implement the validator as a *ContentHandler*, which can be obtained by calling the *getVerifierHandler* method, of the *Verifier* object. In this case the validation is performed by calling the *isValid* method of the *VerifierHandler* object. This method returns logical value true/false depending on whether the validation process ended successfully or not. The other possibility is to implement the validator as *XMLFilter*, which can be done by calling the *getVerifierFilter* method, of the *Verifier* object. In this case the validation process is performed by calling *isValid* method, of the *VerifierFilter* object. For getting more information on errors detected, the *setErrorHandler* method should be used.

3. Validating systems class diagram

In Figure 1 is presented the class diagram of the application that performs validation of XML bibliographic records by using MSV package. A class diagram is used for presenting the static model of a system and is shown in notation of Unified Modeling Language – UML [2].

The diagram in Figure 1 shows the following classes:

- *PrintStream*, *File* and *FilenameFilter*, from standard Java input–output package java.io

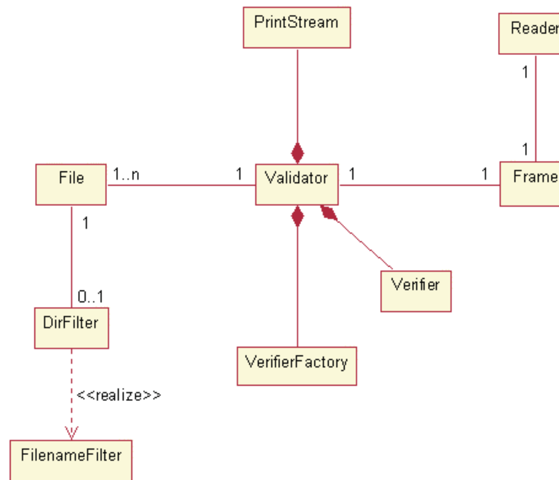

Figure 1: Validating systems class diagram

- *Verifier* and *VerifierFactory*, from the package org.iso–relax.verifier (included in MSV package)
- *Validator*, *Frame* and *Reader* implemented in this paper.

The central class of this application, in which the validation is performed, is *Validator* class. This class contains three instance variables:

- *out*, of type *PrintStream*,
- *factory*, of type *VerifierFactory* and
- *verifier*, of type *Verifier*.

The relationship between classes when the instance of the first class is the instance variable of the second class, is presented on the diagram as composition. The *Validator* class has got two methods: its constructor method and *validate* method.

The creation of *out* object is performed in constructor method, so it represents the output flow redirected to auxiliary file (this file serves to print the outgoing messages of validation process) whose name is passed in as an argument of this method. Then, the redirection of standard output to the *out* flow is done. In the constructor method the instance variable *factory* is also created, there where the specification of the XML Schema language is quoted, because the schema presented in [3] is written in this language.

The *validate* method performs validation. It accepts two arguments of a *String* type: the name of the schema against which the validation will be performed, and a filter which will be used during the file uploading. In the *listF*

array (of *File* type), from the folder which contains XML records, all the records or only those whose name suites to set filter are uploaded. For uploading files according to a filter, the auxiliary class *DirFilter* is used, which implements the standard Java class *FilenameFilter* from java.io package. For every uploaded XML record in the *listF* list, at first is called the *setErrorHandler* method of the *Verifier* class, which enables error handling without throwing any exceptions, and further is called the *verify* method, of the same class, to perform the validation.

The *Reader* is an auxiliary class which performs reading from the auxiliary file (as the one presented in Figure 3) and parsing of its content using *StringTokenizer* class from java.util – standard Java package.

Class *Frame* provides the user's interface for validation of XML records and also provides interaction between the classes *Validator* and *Reader*. For application user, within *Frame* class, is provided to set: a name for the schema and also a filter for choosing documents. Every time the application is started, the user can request the validation to be performed many times for different sets of documents.

4. Validating system's sequence diagram

In Figure 2 is presented the sequence diagram which describes the scenario of validation procedure of XML bibliographic records. The sequence diagram is being used for describing the dynamical characteristics of a system and is shown in notation of UML. This kind of diagram describes the messages that the objects send to each other with a goal to perform certain operation, where the emphasis is on a temporal component. A short description of the scenario shown in Figure 2 is given below.

In this diagram, there are the following objects: *myFrame* of the *Frame* class, *myVal* of the *Validator* class, *factory* of the *VerifierFactory* class, *out* of the *PrintStream* class, *verifier* of the *Verifier* class, *path* of the *File* class and *myReader* of the *Reader* class.

The *User* object sends the message *startVal(schema, filter)* to *myFrame*. It is the request for validation start. Then two arguments of a *String* type are passed in: *schema* – names the file which contains schema and *filter* – names the filter for choosing files. Further, within *myFrame* object the passed arguments from the user interfaces form are taken over using the message *getData()*. Moreover, *myFrame* initiates the creation of *myVal* by sending the message *new(tempFile)*, where *tempFile* represents the name of the file where the system output will be redirected during the validation. The object *myVal* sends the message *new(tempFile)*, and in this way is created the object out. Object *myVal* also sends the message *newInstance("http://www.w3.org/2001/ XMLSchema")* to create the *factory* object. The passed argument indicates the specification of schema language used for writing schema, against which validation will be performed.

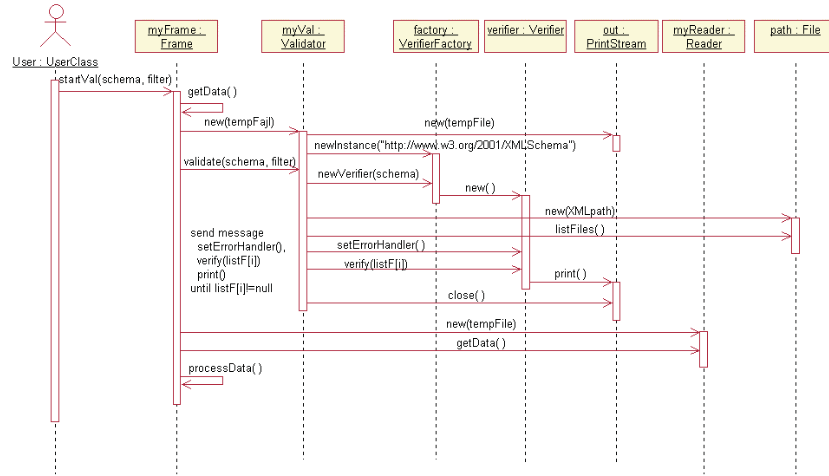

Figure 2: Sequence diagram of the validation system

After that, the control flow is returned to *myFrame*. It sends the message *validate(schema, filter)* to *myVal*. Further, *myVal* sends the message *newVerifier(schema)* to *factory* object, which initiates the creation of *verifier* object. After that, the control flow is returned to *myVal*. Then, *myVal* initiates the creation of *path* object, where *XMLpath* string is passed as argument. This string represents the absolute path in a file system to the folder containing the XML documents to be validated. Furthermore, *myVal* sends the message *listFiles()* to the *path*, to upload files into the array of *File* objects (named *listF*). The array *listF* will be returned as the result of the operation. It will contain all files from the stated path, or if the filter is set by the initiation of validation, only these that suite it.

The set of actions that follow is performed within a loop and is done for every file from the *listF* array. First, *myVal* sends the message *setErrorHandler()* to *verifier*, to enable printing of error messages. Further, *myVal* sends the message *verify(listF[i])* to the same object and then the argument representing the current element of *listF* array. After the validation is complete, the *print()* message is sent to *out*, and the validation results are to be written in the *tempFile*.

When the validation of all files from the array *listF* is completed, *myVal* sends the message *close()* to the *out* object, and destroys it, and the control flow is returned to *myFrame*. Further, *myFrame* initiates the creation of *myReader* object, and the *tempFile* is passed as argument. Moreover, by sending the message *getData()*, *myVal* requests the results of validation and puts them in

the variable *tempData*, an array of four vectors (class *Vector*). At the end, the processing of the obtained data is done within *myFrame* object and the control flow is returned to the *User*.

5. Validation output results

In Figure 3 is presented a section from the auxiliary file in which the standard output during the validation procedure was redirected. The *Reader* class enables reading from that file. To separate messages concerning different documents, the string *"$$$"* is used. If the validated document is valid, then its path and appropriate message are written, e.g. *"D:\XMLzapisi\rec29.xml is valid"*, which means that the document from the file rec29.xml is valid. The documents from the files rec1.xml, rec12.xml, rec13.xml are also valid.

```
$$$
D:\XMLzapisi\rec29.xml is valid
$$$
D:\XMLzapisi\rec1.xml is valid
$$$
Error at line:51, column:10 of file:D:/XMLzapisi/rec10.xml
 tag name "sfa" is not allowed. Possible tag names are: <ind1>

Error at line:62, column:10 of file:D:/XMLzapisi/rec10.xml
 tag name "sf4" is not allowed. Possible tag names are: <ind1>

Error at line:67, column:10 of file:D:/XMLzapisi/rec10.xml
 tag name "sfa" is not allowed. Possible tag names are: <ind1>

$$$
Error at line:21, column:19 of file:D:/XMLzapisi/rec11.xml
 "19??" does not satisfy the "decimal" type

Error at line:39, column:20 of file:D:/XMLzapisi/rec11.xml
 "b. g." does not satisfy the "decimal" type

Error at line:51, column:8 of file:D:/XMLzapisi/rec11.xml
 tag name "sfa" is not allowed. Possible tag names are: <ind1>

$$$
D:\XMLzapisi\rec12.xml is valid
$$$
D:\XMLzapisi\rec13.xml is valid
```

Figure 3: Validation output results

If errors are located in the validation process, for every error is written its exact location in a record (the numbers of line and column), followed by the name with the absolute path of the file and, of course, by the message which precisely describes the error. For example, in Figure 3 is seen that the results of validation for the fourth document (behind the fourth string *"$$$"*), stored in the file rec11.xml, contain three errors. The first one is in the 21st line and 19th column of the document, where, instead of the found data "19?? " the value of the decimal type was expected (*"19??" does not satisfy the "decimal" type*). The second one is in the 39th line and 20th column of the document, where decimal value was expected instead of the found data "b. g." (*"b. g." does not satisfy the "decimal" type*). The third one is in the 51st line and 8th column, where instead of the mandatory element <ind1>, the element <sfa> was found. (*tag name "sfa" is not allowed. Possible tag names are: <ind1>*).

6. Conclusion

Based on the described modelling, the implementation of a validation system for XML bibliographic records is carried out. The output results of validation are stored in a file. The presentation of the obtained validation results can be implemented in different ways (e. g. in the form of a table).

It is also possible to carry out statistical processing of validation results, i.e. to find out how many times appear a message on the missing indicator (on the position where it is defined), on the wrong data type of subfields and indicators content, on the missing mandatory field, etc.

The described validation system could be of significant help in developing a server for shared cataloguing, because it provides the efficiency validation control of a great number of XML bibliographic records. It could be also integrated in such a server as its integral part.

References

[1] DEVELOPER CONNECTION, Sun Multi–Schema Validator http://www.sun.com/software/xml/developers/multischema/

[2] Rumbaugh, J., Jacobson, I., Booch, G., The Unified Modeling Language, Reference Manual. Addison Wesley, 1999.

[3] Zeremski, M., Surla, D., Modelling of Bibliographic Records With XML Schema Language. In: Proceedings of XXVIII Yugoslav Symposium on Operational Research SYM-OP-IS 2001. pp. 265–268. Belgrade, 2001 (in Serbian).

[4] Zeremski, M., Surla, D., The Validation of Bibliographic Records Using XML Schema Language. In: Proceedings of Symposium on Computer Sciences and Informatics YU INFO 2002 (CD–ROM), Kopaonik, 2002 (in Serbian).

Received by the editors December 15, 2002