Schematron Validation for INSPIRE Air Quality Data

Katharina Schleidt, IT Analyst, Austrian Environment Agency, Austria

Abstract

For the validation of XML documents such as the GML documents provided by INSPIRE download services, basic XML validation based on the XSD Schemata does not fully guarantee valid documents. In addition to syntactical problems not identified by schema validation, semantic constraints from the underlying UML data specifications cannot be taken into account. In this paper, we show how through the use of Schematron validation (based on ISO 19757-3) these deficits can be mitigated.

Introduction

As INSPIRE is progressing from the more abstract definition phase to the concrete implementation phase, the provision of correctly GML encoded data files via download services is becoming increasingly relevant. Unfortunately, many of the GML files provided, while being syntactically valid in accordance with the INSPIRE schemata derived from the INSPIRE data specifications, still do not fully conform to the underlying UML models that the INSPIRE data specifications are based on.

This problem becomes more urgent as more complex extensions to the Thematic INSPIRE Data Specifications are being defined, requiring more complex constraints to be applied to the data provided. In the data specification for European Air Quality Directive (AQD) e-Reporting, in addition to the extensions required for the provision of further air quality reporting relevant properties, various complex constraints have been defined.

While XML Schema validation checks the basic syntax of XML files there are many deficits; thus, while the validated files are formally valid, they do not contain complete and valid data. More semantic validation steps such as required for AQD e-Reporting are not supported at all.

In this paper we describe the benefit of complementing basic XSD based XML validation with Schematron validation. This allows us to identify and flag both syntactic errors not covered by XSD based schema validation as well as more complex semantic constraints in a standardised manner. Thus, we can assure that the data provided via the INSPIRE infrastructures truly conforms to the underlying data specifications, and fulfils the requirements of its users.

Deficits of XML Schema validation

While XML Schema validation provides syntactic checks against the XSD, there are many ways of providing valid XML files with serious errors. For example, the following XML element provides a valid INSPIRE namespace:

```
<base:namespace/>
```

XML Schema validation also does not assure that the targets of code-lists or xlinks are correct and valid. Additional semantic checks, i.e. based on constraints in the underlying UML data model, cannot be performed on the XML Schema level.

Schematron

Schematron is a rule-based validation language for making assertions about the presence or absence of patterns in XML trees. Schematron was developed by Rick Jelliffe; he described Schematron as "a feather duster to reach the parts other schema languages cannot reach". Schematron rules can require that the content of an element is controlled by one of its siblings or that an element must have specific attributes; one can also specify required relationships between multiple XML files.Plaintext error messages can be associated with the Schematron rules for easy interpretation by users.

Schematron rules are defined in XML in accordance with the Schematron Schema, defined in ISO/IEC FDIS 19757-3. For the definition of Schematron rules, the functionality of XPath, XSLT and XQuery are utilized. For execution, Schematron rules are usually first transformed to XSLT before being applied to the XML documents to be validated. This allows for easy integration with the other aspects of the data provision infrastructure.

In the following sections, we will describe the basic elements of Schematron rules. For further information, see the Schematron website at http://www.schematron.com.

Schematron Namespaces

All namespaces from the XML file to be checked must be declared using the Schematron namespace element <sch:ns>. Each namespace entry must include the prefix used in the XML file as well as the correct URI for this namespace as follows:

```
<sch:ns prefix="sch" uri="http://purl.oclc.org/dsdl/schematron"/>
<sch:ns prefix="gml" uri="http://www.opengis.net/gml/3.2"/>
…
```

Note: if a namespace is missing, the Schematron file will not work. This is a common error.

Basic Schematron Pattern Syntax

The main element contained within Schematron files is the <pattern> element. This is used to group one or more rules together, as well as to set priorities in rules. The first rule within a pattern where a node matching the context is found is applied; following rules are ignored.

The **<title>** element is used for documentation purposes, and has no influence on the processing.

The central element of Schematron is the **<rule>** element. This provides the context of what is to be checked where in the XML file via the context attribute. The value of the context attribute provides the starting point for the rule; this can be either absolute (so starting from the document root) or relative (so an XPath located anywhere within the XML file). All statements within the <rule> are then referenced relative to the rule context.

Within the <rule> element we find the **<assert>** element; this is the statement that defines the actual check being performed. One must keep in mind that the <assert> statement functions in a negative manner - the Free Text message contained within the <assert> statement is only displayed if the XPath Expression contained in the test attribute is evaluated as false.

For diagnostic purposes, it is often useful to also introduce a **<report>** element; this works parallel to the **<assert>** element but based on the XPath Expression contained in the test attribute being evaluated as true. This can be useful to assure that the rule defined is actually being run and can be removed after development.

```
<sch:pattern>
   <sch:title>Zone Element Check </sch:title>
   <sch:rule context="XPath Expression">
      <sch:assert test="XPath Expression">
         Free Text What's Wrong
      </sch:assert>
      <sch:report test="XPath Expression">
         Free Text What's Right
      </sch:report>
   </sch:rule>
</sch:pattern>
```

Formulating Schematron Rules

Initially, users often find it difficult to define Schematron rules. This is due to the focus on the assert; the assert is triggered if the assert condition is false.

One good trick for getting ones brain around the negative condition required by the assert statement is to first define the report condition. The problem with directly defining the assert is that one needs to define the correct test condition; the error message will be displayed if this condition is NOT met. While one can formulate this directly, it is often easier to first think about the report condition (so what's the good case we wish to report on). The same condition then applies to the assert statement, providing an error message if this condition is not met.

Abstract Schematron Pattern Syntax

As various validation steps are repeated across multiple elements of a schema, it can be useful to define abstract patterns containing the validation logic and then instantiating these patterns with specific parameters for each element where it is required. The syntax is the same as described above, with the addition of the following two attributes being provided within the sch:pattern tag:

- abstract: for abstract patterns, this value is set to "true"
- id: a unique identifier for this abtract pattern. Required for later instantiation.

XPath Functions

For most Schematron rules, some basic functions will be required. Some of the simplest cases are using the ***string-length*** function to ascertain if an element is filled with content or the ***compare*** function to check if a specific value has been provided in an element.

Due to the functional approach used in Schematron, it is sometimes difficult to understand how to nest a series of functions to gain the desired result. For example, in order to heuristically assure that a date is probably in accordance with ISO 8601, we can first remove all Alpha characters pertaining to the standard with the function ***translate***, convert the result to a number with the function ***number***, then convert this result back to a **string** using the function *string* as follows:

```
<sch:let                                                    name="date"
value="string(number(translate(./gml:timePosition,          '-:+TZ',
''))"/>
```

We can then compare the result of this process with the predefined string 'NaN' as follows:

```
<sch:assert test="((string-length($date)>0) and ($date!= 'NaN'))">
```

Using XQuery

For some Schematron rules, we cannot rely only on static XPaths to identify the individual pieces of data to check. In such cases we must utilize XQuery to first identify the explicit instance of data to inspect. Examples where this is necessary are:

- Checking if one instance of a multiple field is provided with a specific value
- Checking if a codelist entry is consistent with an external (XML encoded) codelist
- Checking if the target of an xlink exists either within the same file, or within an external file

In order to formulate an XQuery, one must first define the XPath pointing to the location of the object to be queried. This can be either an element or an attribute.

The element to be queried should be surrounded by square brackets; the opening bracket replaces the last slash (/) within the XPath.

Within the brackets the query condition must be stated. This can be a simple comparison using basic operators or an XPath function may be used.

Simple Operator Query:

```
<sch:let                                                name="sampleFound"
value="/gml:FeatureCollection//aqd:AQD_Sample/aqd:inspireId/base:I
dentifier[base:localId = $sampleRef]" />
```

Query using XPath function contains:

```
<sch:let                                                name="inDictionary"
value="$codelist/rdf:RDF/skos:Concept[contains(@rdf:about,
$language)] " />
```

Design Patterns

In order to facilitate the Schematron rule creation process, in the course of the AQD e-Reporting work we have defined various design patterns for rules, that can be adapted to different contexts and reused. These basic patterns will be discussed in the following sections. For brevity, only the assignment statements (let) and assertion statements (assert) will be shown in these patterns; these statements must be embedded in full Schematron rules as shown above.

Check existence

One of the most basic checks in Schematron rule development is to check if an element has actually been provided. This is necessary in the case that an optional

element must be provided due to a specific profile. In the following example, the variable zoneContent is filled with the content of the XML file available under the XPATH /gml:FeatureCollection/gml:featureMember/aqd:AQD_Zone. In the assert statement, we define the test simply as the variable zoneContent (if there is content, this will evaluate as true):

```
<sch:let name="zoneContent"
value="/gml:FeatureCollection/gml:featureMember/aqd:AQD_Zone"/>
<sch:assert test="($zoneContent)">
```

While the previous rule shows us if the defined element has been provided, it does not yet guarantee that there is actual content available (see the examples in the section on XML Schema validation deficits). In order to assure that content has been provided, we can check the length of the content; to do this, we can modify the test condition within the assert statement to include the length of the field provided as follows:

```
<sch:assert test="string-length($zoneContent)>0">
```

The same logic can be used to check if several elements of a subtype are provided, for this purpose, the content of each element to be checked is assigned to a variable; the test in the assert statement then checks to be sure that all of these elements provide content:

```
<sch:let name="individual"
value="base2:RelatedParty/base2:individualName/gmd:LocalisedCharac
terString"/>
<sch:let name="organisation"
value="base2:RelatedParty/base2:organisationName/gmd:LocalisedChar
acterString"/>
<sch:assert test="(($individual and (string-
length($individual)>0)) and ($organisation and (string-
length($organisation)>0)))">
```

Note: in this case we're doing a double check, first of the existence of the element, and then of the length.

Using the same logic, we can also check if one of several possible elements have been provided. In this case, we simply replace the AND combinations in the test clause with OR statements as follows:

```
<sch:let name="geometry" value="am:geometry"/>
<sch:let name="lau" value="aqd:LAU"/>
```

```
<sch:assert test="((string-length($geometry)>0) or (string-length($lau)>0)">
```

Check for Correct Values

In some contexts, the value of a specific field must always have a predefined value. This can be checked using the *compare* function as shown:

```
<sch:let name="media" value="ef:mediaMonitored"/>
<sch:assert test="($media and (compare($media, 'air') = 0))">
```

In cases where multiple instances of an element may be provided but one needs to assure that one is present with a specific value, XQuery can be used to locate the desired element. In the example below, we wish to assure that one resultQuality element provided pertains to time coverage. To do this, we perform an XQuery on the XPath where the string 'Time Coverage' should be made available; if we have content in the resulting variable, we know that the necessary element is available in the XML file:

```
<sch:let name="timeCoverage"
value="./om:resultQuality/gmd:DQ_DomainConsistency/gmd:result/gmd:DQ_ConformanceResult/gmd:explanation[gco:CharacterString = 'Time Coverage']"/>
<sch:assert test="string-length($timeCoverage)>0" >
```

Similarly, we can also check to see if one of several values has been provided within a specific element. In the example below, we verify that a valid SRS name has been provided. As the relevant part of the SRS name is at the end of the string provided, we use the function *ends-with*:

```
<sch:let name="srsName" value="ef:geometry/gml:Point/@srsName"/>
<sch:assert test="(string-length($srsName)>0) and (ends-with($srsName, '4258') or ends-with($srsName, '4326') or ends-with($srsName, '4937') or ends-with($srsName, '27700'))">
```

We can also check if INSPIRE ID is provided with unique values within the XML file. To perform this test, we must first determine the localId and namespace of the feature being checked; these are assigned to the variables localId and namespace. We then perform an XQuery across the document, looking for other instances of this localId and namespace pair and storing the result in the variable uniqueId. In the assert statement, we formulate the test to first check if the localId and namespace being used for the check are correctly filled, and then count how many occurrences

of this pair have in the variable uniqueId; if this pair is truly unique, there should be only one element contained.

```
<sch:let name="localId" value="./base:Identifier/base:localId"/>
<sch:let name="namespace"
value="./base:Identifier/base:namespace"/>
<sch:let name="uniqueId"
value="count(/gml:FeatureCollection//base:Identifier[base:namespac
e = $namespace][base:localId = $localId]/..)"/>
<sch:assert test="( (string-length($localId)>0) and (string-
length($namespace)>0) and ($uniqueId =1) )">
```

In order to assure that a date interval is correctly provided, we must perform two checks. The first check is to assure that the beginTime is valid. The second check is to assure that if an endTime is provided, this is after the beginTime.

Some difficulties were encountered in creating this rule as the XSLT *datetime*functions return an error if no valid date is provided in the function *call* (i.e. the end date is defined as indeterminatePosition="unknown"). By careful ordering of the conditions in the assert and report statements, checks on the validity of the date are performed before other functions are called. Also, no date comparison operators are available at present, so we had to use a work-around by removing all non-numerical characters from the ISO dataTime string and interpreting the result as a number. Ideally one would use a regular expression to first check the date syntax but this was not done in this case.

```
<sch:let name="beginPosition"
value="./am:designationPeriod/gml:TimePeriod/gml:beginPosition"/>
<sch:let name="endPosition"
value="./am:designationPeriod/gml:TimePeriod/gml:endPosition"/>
<sch:let name="beginTime" value="number(translate($beginPosition,
'-:+TZ', ''))"/>
<sch:let name="endTime" value="number(translate($endPosition, '-
:+TZ', ''))"/>
<sch:assert test="((string($beginTime) != 'NaN'))">…
<sch:assert test="((string($endTime) = 'NaN') or
(string($beginTime) = 'NaN') or ($endTime>= $beginTime))">…
```

Checking with dependencies

There are various cases in which the provision of a specific element depends on the value provided for a different element. The simplest case is when the element to be checked is a boolean - in this case one can do a simple "OR" statement between the

2 fields if the other values are to be provided when the boolean is "FALSE". If the fields must be provided when the boolean value is "TRUE", the *not()* function must be used as in the example below.

Here we check to see if the exceedanceDescription data type is fully filled in case that the value exceedance is set to "TRUE"

```
<sch:let name="exceedance" value="aqd:exceedance"/>
<sch:let name="numericalExceedance"
value="aqd:numericalExceedance"/>
<sch:assert test="((not($exceedance)) or (string-
length($numericalExceedance)>0))">
```

Through extension of the XPaths provided in the variable assignment statements, as well as inclusion of XQuery, this pattern can be extended to check against values stemming from a different section of the XML tree.

If the codelists referred to within the XML document are available in XML form, it is possible to check if the codelist entries provided are valid. In the example below, the codelists for AQD e-Reporting used are made available in SKOS format, which is also XML based, and thus parsable.

In order to perform this check, we first assign the codelist entry to a variable, in this example zoneType. We then open the codelist document and assign it to the variable codelist. We then create a further variable inDictionary, which contains all occurrences of the codelist entry we found in our XML document available within the codelist file. If the variable inDictionary is empty, we know that the codelist entry provided is not available from the codelist specified:

```
<sch:let name="zoneType" value= "aqd:aqdZoneType/@xlink:href" />
<sch:let name="codelist"
value="document('http://dd.eionet.europa.eu/vocabulary/aq/zonetype
/rdf')" />
<sch:let name="inDictionary"
value="$codelist/rdf:RDF/skos:Concept[@rdf:about = $zoneType] " />
<sch:assert test=" $inDictionary" >
```

Checking Xlink Targets

In order to assure that the target of an xlink is also provided, we must assign the entire content of the XML to be checked to a variable, in this example the variable featureCollection. The content of the xlink is assigned to the variable content.We then query featureCollection to see if there is an entry with the value of the xlink in the localId.

Note: at present there is no standardized method for composing xlinks for INSPIRE Ids; thus we're making the assumption that the namespace relevant part ends in AQD/.

```
<sch:let     name="content"     value="substring-after(@xlink:href,
'AQD/')"/>
<sch:let name="featureCollection" value="/gml:FeatureCollection"/>
<sch:let                                        name="valueProvided"
value="$featureCollection/gml:featureMember/aqd:AQD_Attainment/aqd
:inspireId/base:Identifier[base:localId = $content]"/>
<sch:assert test="$valueProvided">
```

The same approach can be used to check if the target of an xlink is available from a different file. However, for this purpose, we must provide information on where the other file is located. This can be done either through modification of the Schematron rule file or through provision of this information in an external configuration file. The external file is opened using the *document* function as described for validating external codelists.

Conclusions

In this paper, we have clearly shown both the limitations of basic XML validation based on XSD schema files, as well as the strengths of Schematron as a way of identifying both more complex syntactical errors as well as semantic errors not formalized within the XSD schema. We have provided an overview of Schematron rule syntax, as well as examples of how to specify rules for common validation requirements. We very much hope that this will help in supporting the community in providing better validated data via INSPIRE services, assuring better accessibility and interoperability for all users.