Constraint Specification Languages: comparing XCSL, Schematron and XML-Schemas

Marta Henriques **Jacinto** <marta.jacinto@itij.mj.pt>
Giovani Rubert **Librelotto** <grl@di.uminho.pt>
José Carlos Leite **Ramalho** <jcr@di.uminho.pt>
Pedro Rangel **Henriques** <prh@di.uminho.pt>

Abstract

After being able to mark-up text and validate its structure according to a grammar, we may start thinking it would be natural to be able to validate some non-structural issues in XML documents like relationships between elements belonging to different contexts, invariants over data models, constraints over attribute values and relationships between attributes.

XML Schemas are a big step in that direction. However, they only allow users to specify primitive constraints like data typing and data format.

Currently, we can find two approaches that represent a complement to DTDs or XML Schemas - XCSL and Schematron - and allow us to specify constraints and to validate the instances of a family of documents against that set of rules. Both are implemented on top of XSL. Both use a kind of an XML envelope to hide XSL specification. XSLT pattern language is the core language of both systems. With all these resemblances it is easy to conclude that they are quite similar. However they differ in some fundamental concepts.

These two constraint specification languages together with XML Schemas were hardly tested and benchmarked with an huge test suite. The most significant results will be discussed in this paper.

We will try to answer questions like: Do they do the same job? Are there some kind of constraints that are easier to specify with one of them? Do you need different background to use the tools? Is it possible to use them in similar situations (the same DTD, the same XML instances)? May we use them to produce an equal result? How do XCSL and Schematron relate to XML Schemas? What is the intersection area of these three? What kind of constraints each one of these three is able to specify? What kind of constraints each one of these three can not specify?

In this article, we will use that test suite and show, step-by-step, the way we handled several kinds of constraints in many different instances.

Table of Contents

1. Introduction 2
2. Schema Languages addressed in this paper 3
    2.1. XML Constraint Specification Language (XCSL) 3
    2.2. Schematron 5
    2.3. W3C XML Schemas 5
3. Case Study 1: Fiscal Certificate 5
    3.1. Constraining with XCSL 7
    3.2. Constraining with Schematron 8
    3.3. Summary 10
4. Case Study 2: 2nd Conference for a Divorce 11

4.1. Constraining with XCSL 12
4.2. Constraining with Schematron 13
4.3. Summary 14
5. Case Study 3: Poem 14
5.1. Constraining with XCSL 15
5.2. Constraining with Schematron 18
5.3. Summary 20
6. Case Study 4: Database 20
6.1. Constraining with XCSL 23
6.2. Constraining with Schematron 25
6.3. Summary 28
7. Conclusion 28
Bibliography 29
Glossary 29

1. Introduction

As a descendant of SGML, XML allows the specification of the documents' structure. This way, the documents will be syntactically validated. However, being sure documents are correctly written from a syntactic point of view, does not assure the static semantics correctness. From the publisher's point of view, it is desirable not to produce invalid documents. Having this problem in mind, we have been working on a solution. That solution is based on the approach followed to specify programming languages: we use a specification language to define a set of contextual conditions over the textual contents of elements and the attribute values that should be satisfied by an XML instance. Those conditions restrict the set of syntactically correct documents to the set of semantically valid ones.

Concerning Content Constraining, we can classify constraints as belonging to one of the four categories defined below:

Domain range checking	This is the most common constraint. We need this type of constraint when we want a certain content/value to be between a pair of values (inside a certain domain). Normally, this kind of constraint is used when data is of type date or numeric.
Dependencies between two elements or attributes	We have cases where an attribute value depends on the value of another element or attribute located in a different branch of the document tree. These are clearly context dependent constraints.
Pattern matching against a Regular Expression	Sometimes we need to guarantee that content follows a certain format (as in the case of dates: there are more than 100 formats, or telephone numbers).
Complex constraints	We call this last kind of constraints complex because we group here all the remaining constraints, usually weird: they require a nested loop behaviour or complex nested calculations. Most of the examples covered in this paper belong to this family of constraints.

This clear separation and the experience gained working with XML documents allowed the consolidation of our approach to deal with the semantic specification problem above referred: the XCSL - XML Constraint Specification Language.

Besides, another solution now exists - Schematron.

Moreover, the W3C consortium proposed the use of XML-Schemas in order to validate documents, instead of the former option: DTDs. XML-Schemas improve the validation degree, still, not all semantic problems can be solved.

In this context, we started studying each of them: XCSL, Schematron and XML-Schemas in order to compare the achieved results.

We use section Section 2 to briefly introduce those technologies.

In the sections Section 3, Section 4, Section 5 and Section 6 we show four complete case-studies. Each section begins with the description of the problem itself: a class of documents we want to create for a special purpose with a set of particular contextual constraints; we introduce the DTD designed to define the structure of that family of documents, and one XML instance. Afterwards we show, with the first tool we are evaluating, XCSL, the way we handled all the problems the particular case-study raises. Finally, we do exactly the same study with the second tool we are evaluating, Schematron. Shortly, the results obtained while using XML-Schemas will be reported.

In section Section 7 we present the paper revision, its purpose and the conclusions we can infer from the reported work in the form of answers to the questions enumerated in the abstract.

2. Schema Languages addressed in this paper

XML Constraint Specification Language (XCSL,), is a domain specific language conceived to allow XML designers to restrict the content of XML documents. It is a simple, and small language useful to write contextual constraints over the textual value of XML elements.

Schematron, on the other hand, has the same purpose.

Both are XML languages; so it becomes possible to add semantic restrictions to XML documents using an XML dialect. These approaches offer document designers a complete XML framework to couple with syntax and semantics. The benefits of such approaches are obvious.

Having so much in common, we ought to compare these two proposals through the use of case-studies, so that we can identify both similarities and main differences between them.

2.1. XML Constraint Specification Language (XCSL)

The Constraint Specification Language is formally defined in [RAM 2000] and in other published articles. This exercise of formalization helped us to find the core structure of the language.

A specification in XCSL is composed by one or more tuples. Each tuple has three parts [RAM 2001]:

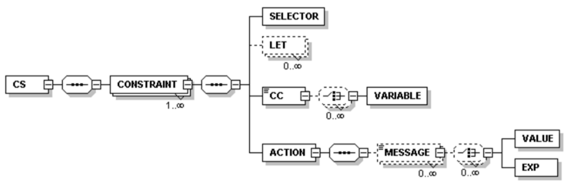
Context Selector	As the name suggests, this is the expression that selects the context where we want to enforce the constraint.
Context Condition	The condition we want to enforce.
Action	The action we want to trigger every time the condition does not hold.

In a more formal notation we can write:

```
ConstraintSpec = Constraint+
Constraint = (ContextSelector, ContextCondition, Action)
```

We could use a grammar to define the language, but as we stated before, we decided to use XSLT to specify the constraints; in order to be coherent, we needed an XML wrapper for XSLT expressions (like in XSL). So, each XCSL specification is defined as an XML instance and the XCSL language is defined by a DTD; the present version of the DTD that specifies XCSL (named xcsl.dtd) is shown below.

Figure 1. XCSL Schema Diagram



Nowadays, XML Schema has overcome the DTD approach to define classes of XML instances. We also made that upgrade; however, as the XML Schema is much more verbose than the correspondent DTD, we decided to include here the DTD and just a diagrammatic description of the XML schema. That diagram is shown in Figure Figure 1, as obtained with the XML Spy 4.1, from Altova.

```
<?xml version="1.0" encoding="UTF-8"?> <!-- XCSL: XML
Constraint Specification Language --> <!ELEMENT CS
(CONSTRAINT)+> <!ATTLIST CS
    DTD CDATA #IMPLIED
    DATE CDATA #IMPLIED
    VERSION CDATA #IMPLIED
>
<!ELEMENT CONSTRAINT (SELECTOR, LET*, CC, ACTION)> <!ELEMENT
SELECTOR EMPTY> <!ATTLIST SELECTOR
    SELEXP CDATA #REQUIRED
>
<!ELEMENT LET EMPTY> <!ATTLIST LET
    NAME CDATA #REQUIRED
    VALUE CDATA #REQUIRED
>
<!ELEMENT CC (#PCDATA | VARIABLE)*> <!ELEMENT VARIABLE
EMPTY> <!ATTLIST VARIABLE
    SELEXP CDATA #REQUIRED
>
<!ELEMENT ACTION (MESSAGE*)> <ELEMENT MESSAGE (#PCDATA |
VALUE)*> <!ELEMENT VALUE EMPTY> <!ATTLIST VALUE
    SELEXP CDATA #REQUIRED
>
```

Figure 2. XCSL Workflow

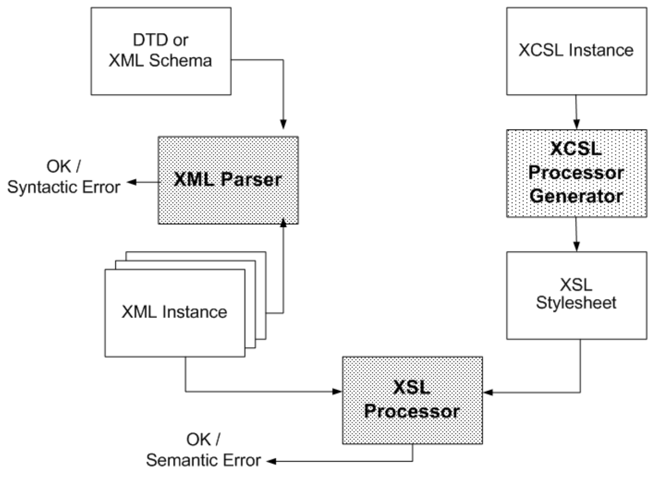


Figure Figure 2 shows the XCSL Workflow. This is the process to validate documents' semantics driven by XCSL Constraint Specifications.

2.2. Schematron

Schematron is an XML schema language which was designed and implemented by Rick Jelliffe at the Academia Sinica Computing Centre, Taiwan. It combines powerful validation capabilities with a simple syntax and implementation framework.

At Schematron's design and specification time, there were several aims, from which we highlight: to promote natural language descriptions of validation failures; to allow a more human-readable answer as the validation result; aim for a short learning curve by layering on existing tools (XPath and XSLT); support workflow by providing a system which understands the phases through which a document passes in its lifecycle.

Schematron is described in many papers, for instance in [DOD 2001]. Its DTD and XML-Schema can be found in *http://www.ascc.net/xml/schematron/*.

Validating XML instances using Schematron involves two steps. The first one is to compile the constraints' document into a run-time validator. The second one is running each instance against that specific validator.

Two options are available: *Topologi Schematron Validator* - the two steps process is transparent to the user, and all that he/she needs is to provide the constraints' document and the XML instances; and *schematron-report* - the user explicitly has to engage both the steps.

While *schematron-report* is processed in the command-line invoking a tool like Saxon, (as we do with XCSL); *Topologi Schematron Validator* is a windows-based interactive environment.

2.3. W3C XML Schemas

XML Schema was created once the syntax of DTDs fell short of the requirements of the XML users. The aims of the W3C XML Schema Working Group were to create a language that would be more expressive than DTDs and written in XML Syntax. In addition, it would also allow authors to place restrictions on the elements' content and attribute values in terms of primitive datatypes found in most languages.

While using DTDs, to specify constraints, even if very simple, we need to use a constraining language (such as XCSL or Schematron) and, consequently, need two documents to completely validate an XML instance. Constraining with XML-Schemas, on the other hand, means, for a particular set of constraints, using only one document to validate XML instances instead of using both a DTD and a constraint document. Unfortunately, the range of constraints we can validate with XML-Schemas is far from the set we specified above.

To validate XML instances against a XML-Schema, several parsers are available. Those parsers are similar to the traditional XML validators; but now, instead of a DTD they are driven by an XML-Schema.

3. Case Study 1: Fiscal Certificate

Let us suppose that someone in Portugal asks for a fiscal certificate, which is a certificate of the goods declared by someone's relatives by the time of his or her death (this is compulsory so that the goods can be inherited). That document should include the identification of the dead one and both the dates: the one of that person's death and the certificate requirement's; these are required fields (obviously, the current date shall be the most recent). Moreover, people must ask for this kind of certificate in the finance department of the last residence area of the dead one, so we should enforce that congruence too. The correspondent DTD for such a document, could be:

```
<!ELEMENT fcert (header, body, ending)> <!ELEMENT header
(#PCDATA | department)*> <!ELEMENT department (#PCDATA)>
<!ATTLIST department
    place CDATA "0101"
>
<!ELEMENT body (requester, request)> <!ELEMENT requester
(#PCDATA | name | CF | address)*> <!ELEMENT name (#PCDATA)>
```

```
<!ELEMENT CF (#PCDATA)> <!ELEMENT address (#PCDATA)>
<!ELEMENT request (#PCDATA | affinity | name | date | village |
parish | municipality)*> <!ELEMENT affinity (#PCDATA)>
<!ELEMENT date (#PCDATA)> <!ATTLIST date
    value CDATA "19000101"
>
<!ELEMENT village (#PCDATA)> <!ELEMENT parish (#PCDATA)>
<!ATTLIST parish
    place CDATA "010101"
>
<!ELEMENT municipality (#PCDATA)> <!ATTLIST municipality
    place CDATA "0101"
>
<!ELEMENT ending (#PCDATA | place | date)*> <!ELEMENT place
(#PCDATA)>
```

One XML instance is as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?> <!DOCTYPE fcert
SYSTEM "fcert_cm.dtd"> <fcert>
    <header>
        Dear Sir, Chief of the Finance Department of
        <department place="110504">Lisbon's 4th Fiscal Parish</department>
    </header>
    <body>
        <requester>
            <name>Rita Santos </name>
            taxpayer Ner.
            <CF>31988455</CF>
            with the address
            <address>Pedras tortas Street, Ner 7 - 5423 Ranholas
          </address>
        </requester>
        <request>
            requests your Excellency to certify if, on behalf of the death
            ...
            <name>Francelestina Pereira e Santos</name>
            who died on the
            <date value="19990913">13th of September 1999</date>
            ...
            parish of
            <parish place="100611">Salir de Matos</parish>
            municipality of
            <municipality place="1006">Caldas da Rainha</municipality>
            and maried she was with
            ...
        </request>
    </body>
    <ending>
        Ask that her request be granted
        <place>Caldas da Rainha</place>
        <date value="19991020">20th of October 1999</date>
        The requester
    </ending>
</fcert>
```

This particular XML is valid from a static and semantic point of view. Any XML parser is able to check its structure against the given DTD. However if the current date was before the death's date, the document was invalid and the DTD doesn't provide the means to verify that.

3.1. Constraining with XCSL

In order to be able to specify the first constraint, we added an attribute *value* to the *date* element which will keep the date in a standard format "yyyymmdd". The semantic constraint is the following (we only show the *CONSTRAINT* element as the rest is trivial)

```
                    <CONSTRAINT>
    <SELECTOR SELEXP="//request/date"/>
    <CC>
    @value < /fcert/ending/date/@value
</CC>
    <ACTION>
        <MESSAGE>
        The date of the death pointed out:
          <VALUE SELEXP="/fcert/body/request/date"/>,
        is posterior to the request date:
          <VALUE SELEXP="/fcert/ending/date"/>
        </MESSAGE>
    </ACTION>
</CONSTRAINT>
```

We compare the value of those two attributes: the first belonging to the *body* sub-tree (date of death) and the other one belonging to the *ending* sub-tree (date of the request). As the date 19990913 occurred before 19991020, the XML instance would be correctly validated. If this was not the case, an error message would be emitted; for instance, if the first attribute value was 20010803 and the second one was 20010607, the following error message would be triggered:

```
   <err-message>
            The date of the death pointed out: 3rd of August 2001,
            is posterior to the request date:  7th of June 2001
      </err-message>
```

In order to be sure that the request is delivered in the appropriate Finance Department, we specified the following constraint.

```
<CONSTRAINT>
    <SELECTOR SELEXP="//fcert/body/request"/>
    <CC>
    parish/@place = /fcert/header/department/@place
    or
    municipality/@place = /fcert/header/department/@place
</CC>
    <ACTION>
        <MESSAGE>
        The request for this certificate shall not be
        delivered in this department
        <VALUE SELEXP="/fcert/header/department"/>,
        but in the department in charge of the
        <VALUE SELEXP="parish"/>'s parish,
        <VALUE SELEXP="municipality"/>'s municipality.
    </MESSAGE>
    </ACTION>
</CONSTRAINT>
```

Where we want to ensure that the *place* attribute of the *department* element is equal to the same *place* attribute of one of the elements *parish* or *municipality*.

To define such a constraint, we compare the value of each *place* attribute belonging to the *body* sub-tree (simplified by writing just the *request* element, descendant of the *body* element, once the first one is unique in the whole document) - with the *place* attribute belonging to the *header* sub-tree (place of the Department). As in this case the person is willing to deliver the request in a department which *local* attribute is not equal to the *parish*'s one nor to the *municipality*'s one, the following error message would be displayed:

```
 <err-message>
        The request for this certificate shall not be
        delivered in this department
        Lisbon's 4th Fiscal Parish,
        but in the department in charge of the
        Salir de Matos's parish, Caldas da Rainha's municipality.
    </err-message>
```

Moreover, when using DTDs we can not enforce order nor cardinality of elements allowed to occur in mixed content. Therefore, we will need a set of constraints that enforce both the order and cardinality inside the mixed content of the elements *header*, *requester*, *request* and *ending*.

Concerning the *requester* element, we want to force each sub-element to appear only once and the order of appearance to be: *name*, *CF* (tax payer number) and *address*. The respective constraint is:

```
<CONSTRAINT>
    <SELECTOR SELEXP="//fcert/body/requester"/>
    <CC>
    (count(name) = 1) and
    (count(CF) = 1) and
    (count(address) = 1) and
    name(name[1]/following::*) ='CF' and
    name(CF[1]/following::*) ='address'
</CC>
    <ACTION>
        <MESSAGE>
        Either -requester- sub-elements occur in a wrong order,
        either they occur a wrong number of times.
    </MESSAGE>
    </ACTION>
</CONSTRAINT>
```

The number of occurrences of each of the elements *name*, *CF* and *address*, sub-elements of the *requester* element is counted and compared to 1. Besides, we verify that the *name* element is followed by a *CF* element, followed by an *address* element. The instance we presented would produce no errors, given that every element occurs only once and in the expected order.

The constraints for the other three elements are similar.

3.2. Constraining with Schematron

Remember that the particular XML instance shown in Section 3 would be validated correctly for its structure and is, concerning the dates, semantically correct as well, but if the current date was not correct, the document would be validated the same way and wouldn't be correct.

The first semantic constraint is the following (we only show the *diagnostics* and *pattern* elements as the rest is trivial).

```
    <diagnostics>
        <diagnostic id="00">
            Correct!
        </diagnostic>
        <diagnostic id="01">
            The indicated date of the death:
            <value-of select="/fcert/body/request/date"/>,
            is posterior to the request date:
              <value-of select="/fcert/ending/date"/>
        </diagnostic>
    </diagnostics>
    <pattern name="dates">
        <rule context="//request/date">
            <assert test="@value < /fcert/ending/date/@value"
              diagnostics="01"/>
            <report test="@value < /fcert/ending/date/@value"
              diagnostics="00"/>
        </rule>
    </pattern>
```

We compare the value of those two attributes: the first belonging to the *body* sub-tree (date of death) and the other one belonging to the *ending* sub-tree (date of the request). As the date 19990913 occurred before 19991020, the XML instance would be correctly validated and the return would be:

```
Schematron Report

Validation of the Fiscal Certificate
   dates
   Correct!
```

If it wasn't so, we would receive an error, for instance if the first *value* attribute's value was 20010803 and the second one was 20010607, the following error message would be triggered (from now on, we will only show the piece of return that relates to what we are explaining, avoiding unnecessary repetition):

```
   dates
   The indicated date of the death: 3rd of August 2001 ,
   is posterior to the request date:
   7th of June de 2001
```

In order to be sure that the request is delivered in the appropriate department, we specified the following constraint, where we want to ensure that the *place* attribute of the *department* element is equal to the same *place* attribute of one of the elements *parish* or *municipality*

```
    <diagnostics>
        <diagnostic id="02">
            The request for this certificate shall not be
            delivered in this department
            <value-of select="/fcert/header/department"/>,
            but in the department in charge of the
            <value-of select="parish"/>'s parish,
            <value-of select="municipality"/>'s municipality.
        </diagnostic>
    </diagnostics>
    <pattern name="Finance department">
        <rule context="//fcert/body/request">
            <assert test="parish/@place =
               /fcert/header/department/@place
        or
        municipality/@place = /fcert/header/department/@place"
          diagnostics="02"/>
        </rule>
    </pattern>
```

We compare the value of each *place* attribute belonging to the *body* sub-tree - with the *place* attribute belonging to the *header* sub-tree (place of the department). As in this case the person is willing to deliver the request in a department which *local* attribute is not equal to the *parish*'s one nor to the *municipality*'s one, the following message would be displayed:

```
  Finance department
  The request for this certificate shall not
  be delivered in this department
  4º Bairro Fiscal de Lisboa ,
  but in the department in charge of the Salir de Matos 's parish,
  Caldas da Rainha 's municipality.
```

Next, we present the constraint that enforces both the order and cardinality of the *requester* element. For the *header*, *request* and *ending* elements, the constraints are alike.

For the *requester* element, we want to force each sub-element to appear only once and the order of appearance to be *name*, *CF* (tax payer number) and *address*, so the constraint is:

```
  <diagnostics>
      <diagnostic id="04">
          Either -requester- sub-elements occur in a wrong order,
          either they occur a wrong number of times.
      </diagnostic>
  </diagnostics>
  <pattern name="requester element">
      <rule context="//fcert/body/requester">
          <assert test="(count(name) = 1) and
                        (count(CF) = 1) and
                        (count(address) = 1) and
                        name(name[1]/following::*) ='CF' and
                        name(CF[1]/following::*) ='address'"
                          diagnostics="04"/>
          </rule>
  </pattern>
```

The number of occurrences of each of the elements *name*, *CF* and *address*, sub-elements of the *requester* element is counted and compared to 1. Besides, we verify that the *name* element is followed by a *CF* element, by its turn followed by an *address* element. The instance we presented would produce no errors once every element occurs once and by the expected order.

3.3. Summary

For this case-study, no major diferences were found in the specification. Nevertheless, we may state that with Schematron we have a linear way of providing the information of correctness of the document - *report* element - , whereas with XCSL we would have to write the negation ourselves.

4. Case Study 2: 2nd Conference for a Divorce

In Portugal, to get divorced, a couple has to deliver two documents in court. The first one is called the *First Request for Divorce*, time when they state their will to get divorced, assuring that in the presence of the judge. The second one, which we are focusing on in this example, is called the *Second Conference Requirement* and the earlier it can be submitted is 90 days after the first one. This last one is to ask for a new meeting where the couple will state they still wish to get divorced, after which the couple will be officially divorced. At the moment of designing the DTD for this family of documents, we have two options: the number of days passed since the first conference will be stated; or just the date of the first conference will be written. It makes much more sense adopting the second one. Therefore, the DTD will be:

```
    <!ELEMENT div_2c (header, body, ending)>
    <!ELEMENT header (sender, addressee)>
    <!ELEMENT sender (#PCDATA | cdepart)*>
    <!ELEMENT cdepart (#PCDATA)>
    <!ELEMENT addressee (#PCDATA | court)*>
    <!ELEMENT court (#PCDATA)>
    <!ELEMENT body (requesters, request)>
    <!ELEMENT requesters (#PCDATA | name)*>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT request (#PCDATA | date | article)*>
    <!ELEMENT date (#PCDATA)>
    <!ATTLIST date
        value CDATA "19000101"
    >
    <!ELEMENT article (#PCDATA)>
    <!ELEMENT ending (text, place, date, signature, signature)>
    <!ELEMENT place (#PCDATA)>
    <!ELEMENT signature (#PCDATA)>
    <!ELEMENT text (#PCDATA)>
```

One possible XML instance is as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?> <!DOCTYPE div_2c
SYSTEM "div_2c02.dtd"> <div_2c>
  <header>
    ...
  </header>
  <body>
    ...
    <request>
      identified in the referred Action of Divorce
      official papers, having accomplished the first
      conference in the
      <date value="20010406">6th of April of 2001</date>
      and both maintaining their will to divorce, come,
      by this means to require to be convoked for the
      second conference, according to the
      <article>1423th article of the Code of Civil Law
      </article>
      in order to the definite divorce be decreed.
    </request>
  </body>
  <ending>
    <text> Ask that their request be granted </text>
    <place>Caldas da Rainha</place>
    <date value="20010506">6th of May of 2001</date>
    <signature>The first requester's Lawyer</signature>
    <signature>The second requester</signature>
  </ending>
</div_2c>
```

Structurally, this XML instance would be validated by any XML tool, but what about the date comparison? It is essencial that the requirement for the Second Conference occurs at least 90 days after the first conference. For that, we will need to compare both dates. Out of this comparison, we shall get the number of days in which they differ. To achieve this, we may use a function [TON 2000]that receives a gregorian date and returns the Julian day for each of the dates. Afterwards, it is enough to subtract them to get the number of days. Finally, comparing that result with the number 90, we will know exactly whether the document is semantically correct or not.

4.1. Constraining with XCSL

With XCSL, we will specify this semantic constraint as follows (again, we only show the *CONSTRAINT* element):

```
<CONSTRAINT>
 <SELECTOR SELEXP="//div_2c"/>
 <LET NAME="a" VALUE="(floor((14-substring(ending/date/
                       @value,5,2)) div 12))"/>
 <LET NAME="y" VALUE="(substring(ending/date/@value,1,4)
                      + 4800 - $a)"/>
 <LET NAME="m" VALUE="(substring(ending/date/@value,5,2)
                      + 12 * $a - 3)"/>
 <LET NAME="t" VALUE="(substring(ending/date/@value,7,2)
                      + floor((153 * $m + 2) div 5) +
                      (365 * $y) + floor($y div 4) -
                      floor($y div 100) +
                      floor($y div 400) - 32045)"/>
   ...
 <CC>
    ($t - $t2) >= 90
 </CC>
  <ACTION>
     <MESSAGE>
        Only <VALUE SELEXP="($t - $t2)"/> days undergone
        since the first conference...
        You will have to wait a little longer!!
     </MESSAGE>
  </ACTION>
</CONSTRAINT>
```

We use 8 elements *LET* as they will provide the modularity needed to reduce the amount of lines needed to specify this constraint. The context is the *root* element (*div_2c*). The first four *LET* elements are applied to the *ending* branch of the document's tree (where we can find the date of the requirement itself) --- we use the names *a*, *y* and *m* for the intermediary calculations and, finally, *t* to keep the Julian day of that date. The second set of *LET* elements is applied to the *body* branch of the document's tree (where we can find the date of the first petition) --- we now use the names *a2*, *y2*, *m2* and *t2*, with the same meaning.

After defining all this variables, the *CC* element itself will be as simple as subtracting *t2* out of *t* and comparing the result with 90.

To provide a personalized result, we can simply use a *VALUE* element inside *ACTION* element, where we use the variables *t* and *t2* we defined before, avoiding the duplication of all the code.

The XML instance shown above is structurally correct, but semantically incorrect since the first *date* is the 6th of April and the second one the 6th of May, both of the year 2001. Therefore, while validating this document against the constraint document specified above, we would get the following error message:

```
<?xml version="1.0" encoding="iso-8859-1"?> <doc-status>
    <err-message>
        Only 30 days undergone
        since the first conference...
        You will have to wait a little longer!!
    </err-message>
</doc-status>
```

where 30 is the number of days between the two dates, generated automatically according to the *VALUE* element specified in the constraint.

4.2. Constraining with Schematron

At this time, the only possibility of specifying the constraint we specified in the previous subsection, now with Schematron, is writing the whole equation each time we need to use it. This is due to the fact that, in Schematron language, only XPath functions are available and the one used with XCSL, that allows variables to be instantiated, is an XSL function. The equivalent constraint in Schematron would, therefore, be,

```
<title>Request for the 2nd conference of divorce</title>
<diagnostics>
 <diagnostic id="01">
   Less than 90 days undergone since the first
   conference...
   You will have to wait a little longer!!
 </diagnostic>
</diagnostics> <pattern name="Days since the First
Conference">
 <rule context="//div_2c">
  <assert test="
   (((substring(ending/date/@value,7,2)+
   floor((153*(substring(ending/date/@value,5,2)+12*
     (floor((14-(substring(ending/date/@value,5,2)))
     div 12))-3)+2) div 5)+
   (365 * (substring(ending/date/@value,1,4)+4800-
     (floor((14-(substring(ending/date/@value,5,2)))
     div 12)))))+
   floor((substring(ending/date/@value,1,4)+4800-
     (floor((14-(substring(ending/date/@value,5,2)))
     div 12))) div 4)-
   floor((substring(ending/date/@value,1,4)+4800-
     (floor((14-(substring(ending/date/@value,5,2)))
     div 12))) div 100)+
   floor((substring(ending/date/@value,1,4)+4800-
     (floor((14-(substring(ending/date/@value,5,2)))
     div 12))) div 400)-32045)
   - ...)
   >= 90" diagnostics="01">
  </assert>
 </rule>
</pattern>
```

Where we put "...", all the code for the calculation of the Julian day is repeated, now for the *body* branch. We don't repeat it to keep the example as understandable as possible. Notice that in this constraint we are not providing the personalized output we did with XCSL. To do this, it would be necessary to repeat all the code listed above for the calculation of both of the dates. This is clearly a disadvantage once the number of lines we need to specify the Schematron 's constraint is huge when compared with XCSL's one.

As we pointed out before, the semantic validation of the XML instance we presented before triggers an error message, which in this case will be, :

```

Request for the 2nd conference of divorce

Days since the First Conference
    Less than 90 days undergone since the first conference...
    You will have to wait a little longer!!
```

This does not mean we can not use a key functionality in Schematron, but that it can not be used for variables, and can only be used to keep a set of values in a list against which we will be able to compare other values - cases like the uniqueness problem in databases we show in the last case-study.

4.3. Summary

This is clearly an example in which the XCSL specification is much easier than the Schematron's one. With the last one, it is much easier to make mistakes. Moreover, the XCSL specification is more clear and any change of the code will be simpler.

5. Case Study 3: Poem

The poem family of documents is generically made up of a set of stanzas. Traditionally, it is composed by distichs, tercets, quatrains, quintains, sestets, septets, octaves, nine verses stanzas and ten verses stanzas, in any number and order. Free poems may have stanzas with virtually any number of verses, reason why we must define a general stanza (gstanza).

To make it possible to identify the kind of poem we are writing, we shall have a style attribute. This is important to identify, for instance, sonnets, a sub-family of the poems family. Portuguese sonnets must be made of four stanzas, the first two are quatrains and the last two are tercets, written exactly by this order.

Bellow, we show the DTD we have designed for this family of documents, allowing the style of the poem to be stated and also the specification of general stanzas, beyond the nine tradicional stanzas.

```

    <!ELEMENT poem (title, author, body, date)>
    <!ATTLIST poem
        style CDATA #IMPLIED
    >
    <!ELEMENT title (#PCDATA)>
    <!ELEMENT author (#PCDATA)>
    <!ELEMENT body (distich | tercet | quatrain | quintain | sestet |
      septet | octave |
    ninev | tenv | gstanza)+>
    <!ELEMENT distich (verse, verse)>
    <!ELEMENT tercet (verse, verse, verse)>
    <!ELEMENT quatrain (verse, verse, verse, verse)>
    <!ELEMENT quintain (verse, verse, verse, verse, verse)>
    <!ELEMENT sestet (verse, verse, verse, verse, verse, verse)>
    <!ELEMENT septet (verse, verse, verse, verse, verse, verse,
      verse)>
    <!ELEMENT octave (verse, verse, verse, verse, verse, verse,
      verse, verse)>
    <!ELEMENT ninev (verse, verse, verse, verse, verse, verse,
      verse, verse, verse)>
    <!ELEMENT tenv (verse, verse, verse, verse, verse, verse,
      verse, verse, verse, verse)>
    <!ELEMENT gstanza (verse)+>
    <!ELEMENT verse (#PCDATA | name | place)*>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT place (#PCDATA)>
    <!ELEMENT date (#PCDATA)>
```

Gstanza represents all the stanzas which have just one *verse* or more than 10 *verses*. As we are using a DTD, we can only state that *gstanza* is made of *verse* elements (one or more). Therefore, we will need a constraint that returns an error when this element is misused.

The XML instance bellow is structurally correct and represents a *poem* which has sonnet *style*.

```
    <?xml version="1.0" encoding="ISO-8859-1"?>
    <!DOCTYPE poem SYSTEM "poema_n.dtd">
    <poem style="sonnet">
        <title>Sonnet antique</title>
        <author>Álvaro de Campos</author>
        <body>
            <quatrain>
                <verse>Olha, <name>Daisy</name>: quando eu
                  morrer tu hás-de</verse>
                <verse>dizer aos meus amigos aí de
                  <place>Londres</place>,</verse>
                <verse>embora não o sintas, que tu escondes</verse>
                <verse>a grande dor da minha morte. Irás de</verse>
            </quatrain>
            <quatrain>
                <verse>
                    <place>Londres</place> p'ra <place>Iorque</place>,
                    onde nasceste
            (dizes ...</verse>
                <verse>que eu nada que tu digas acredito),</verse>
                <verse>contar àquele pobre rapazito</verse>
                <verse>que me deu tantas horas felizes,</verse>
            </quatrain>
            <tercet>
                <verse>embora não o saibas, que morri ...</verse>
                <verse>Mesmo ele, a quem eu tanto julguei amar,</verse>
                <verse>nada se importará... Depois vai dar</verse>
            </tercet>
            <tercet>
                <verse>a notícia a essa estranha <name>Cecily</name>
                </verse>
                <verse>que acreditava que eu seria grande...</verse>
                <verse>Raios partam a vida e quem lá ande!</verse>
            </tercet>
        </body>
        <date>1922</date>
    </poem>
```

5.1. Constraining with XCSL

According to all that was said, we will need two constraints to verify if, having *sonnet style*, poems only have two *quatrains* and two *tercets*. We decided to implement two constraints for each one of these in order to be able to produce a clearer information to the user. This way the user will know exactly what is wrong: less/more than 2 *quatrains*, less/more than 2 *tercets* were used.

For the first element, *quatrain*, these constraints will be:

```
  <CONSTRAINT>
    <SELECTOR SELEXP="//poem"/>
    <CC>
    not(@style = 'sonnet') or
    count(body/quatrain) >= 2</CC>
    <ACTION>
      <MESSAGE>WARNING:
        A sonnet must have 2 quatrains
        </MESSAGE>
    </ACTION>
  </CONSTRAINT>
  <CONSTRAINT>
    <SELECTOR SELEXP="//poem"/>
    <CC>
    not(@style = 'sonnet') or
    count(body/quatrain) <= 2</CC>
    <ACTION>
      <MESSAGE>WARNING:
        A sonnet only may have 2 quatrains</MESSAGE>
    </ACTION>
  </CONSTRAINT>
```

This way, when the *quatrain* element occurs less than twice, the user is informed a *sonnet* must have two *quatrains* and, when it occurs more than twice, that the *sonnet style* allows only two *quatrains*.

For these two constraints, we say that unless *style* is different from sonnet, the *quatrain* element must occur twice. In the first constraint we evaluate if the number of occurrences is less than two, writing, therefore, the complementary expression, greater or equal, triggering the error message every time the negation occurs. In the second, we evaluate if the number of occurrences is greater than two, writing, therefore, the complementary expression, less or equal, triggering the error message every time the negation occurs.

The particular instance we've shown would produce no errors, but with an instance of a *sonnet* with only one *quatrain*, we would receive the following message:

```
   <err-message>WARNING:
        A sonnet must have 2 quatrains
   </err-message>
```

If we did use 3 or more *quatrain* elements, the following error would be displayed:

```
    <err-message>WARNING:
        A sonnet only may have 2 quatrains
    </err-message>
```

For the *tercet* element, the constraints are similar, reason why we won't show them.

With the constraints specified above, we already obligate a *sonnet* to have two *quatrains* and two *tercets*, but we still lack the validation of the order in which they occur. We specify the following constraint to verify if the first element *quatrain* is followed by another *quatrain* element, which is followed by a *tercet* element which is, by its turn, followed by another *tercet* element. We also verify that the global occurrences of this two elements equals 4, once it is enough to have more than 4 *quatrains* and *tercets* for the structure to be incorrect.

```
<CONSTRAINT>
    <SELECTOR SELEXP="//poem/body"/>
    <CC>
    not(../@style = 'sonnet') or
    ((count(quatrain) + count(tercet) = 4) and
    name(quatrain[1]/following::*) ='quatrain' and
    name(quatrain[2]/following::*) ='tercet' and
    name(tercet[1]/following::*) ='tercet')
    </CC>
    <ACTION>
      <MESSAGE>WARNING:
        The only structure allowed for a sonnet is:
            Quatrain
            Quatrain
            Tercet
            Tercet
        </MESSAGE>
    </ACTION>
  </CONSTRAINT>
```

The presented instance would produce no errors but if we swapped the order of the first *quatrain* and the first *tercet*, we would receive:

```
    <err-message>WARNING:
        The only structure allowed for a sonnet is:
            Quatrain
            Quatrain
            Tercet
            Tercet
    </err-message>
```

The sonnets are now completely validated. However, we still need one constraint that guarantees that general stanzas are used only when and if the poem's style is *freestyle*:

```
  <CONSTRAINT>
    <SELECTOR SELEXP="//poem/body"/>
    <CC>
    ../@style = 'freestyle' or
    count(gstanza) = 0</CC>
    <ACTION>
      <MESSAGE>WARNING:
        The style of your poem is <VALUE SELEXP="../@style"/>,
        therefore you can't use stanzas
        with only one verse or more than ten verses,
        allowed only in the case of free poems.
      </MESSAGE>
    </ACTION>
  </CONSTRAINT>
```

Unless the poem has *freestyle*, the element *gstanza* must not be used (it shall never occur).

Again, the instance we've shown would produce no errors once we did not use any *gstanza* elements. If we did use a *gstanza* element, we would receive the following error message:

```
    <err-message>WARNING:
        The style of your poem is sonnet,
        therefore you can't use stanzas
        with only one verse or more than ten verses,
        allowed only in the case of free poems.
      </err-message>
```

Finally, and once we can't specify any cardinality we want with a DTD, we need a constraint to prohibit the use of the *gstanza* element for the specification of *stanzas* with 2, 3, ..., 10 verses:

```
<CONSTRAINT>
  <SELECTOR SELEXP="//poem/body/gstanza"/>
  <CC>
      count(verse) = 1 or count(verse) > 10 </CC>
  <ACTION>
    <MESSAGE>WARNING:
      The element -gstanza- was used to specify a stanza with
      <VALUE SELEXP="count(verse)"/>
      verses. Instead, the appropriate element must be used.
    </MESSAGE>
  </ACTION>
</CONSTRAINT>
```

Again, the XML instance would produce no errors, but it would be enough to use the *gstanza* element instead of the *quatrain* element for the first quatrain to get the following error:

```
  <err-message>WARNING:
      The element -gstanza- was used to specify a stanza with 4
      verses. Instead, the appropriate element must be used.
  </err-message>
```

5.2. Constraining with Schematron

As stated previously, we will need two constraints to verify if, having sonnet *style*, poems only have two *quatrains* and two *tercets*, i.e., both elements occur twice each. Again, we decided to implement two constraints - two *assert* elements - for each one of these in order to be able to produce a clearer information to the user. Once with schematron the restriction's title is always shown, we also specified two *report* elements: the first one returns "correct!" every time the poem has sonnet style and the restriction is respected; the second one returns "Not applicable." when the poem's *style* is not sonnet.

For the first element, *quatrain*, these constraints will be:

```
  <title>Poem's validation</title>
  <diagnostics>
      <diagnostic id="00">
          Correct!
      </diagnostic>
      <diagnostic id="00a">
          Not applicable.
      </diagnostic>
      <diagnostic id="01a">
          A sonnet only may have 2 quatrains
      </diagnostic>
      <diagnostic id="01b">
          A sonnet must have 2 quatrains
      </diagnostic>
  </diagnostics>
  <pattern name="Sonnets: Quatrains">
      <rule context="/*">
          <assert test="not(@style = 'sonnet') or
              count(body/quatrain) <= 2" diagnostics="01a">
          </assert>
          <assert test="not(@style = 'sonnet') or
              count(body/quatrain) >= 2" diagnostics="01b">
          </assert>
          <report test="@style = 'sonnet' and
              count(body/quatrain) = 2" diagnostics="00"/>
```

```
            <report test="@style != 'sonnet'" diagnostics="00a"/>
        </rule>
    </pattern>
```

If our XML instance was of a *sonnet* with only one *quatrain*, we would receive the following message:

```
    Poem's validation
    Sonnets: Quatrains
    A sonnet must have 2 quatrains
```

If we did use 3 or more *quatrain* elements, the following error would be displayed:

```
    Sonnets: Quatrains
    A sonnet only may have 2 quatrains
```

For the *tercet* element, the constraints are similar, reason why we won't show them.

We will also need a constraint which, provided the *style* attribute equals to *sonnet*, analises the global structure of the poem and verifies it is: *quatrain*, *quatrain*, *tercet*, *tercet*. It verifies the first element *quatrain* is followed by another *quatrain* element, which is followed by a *tercet* element which is, by its turn, followed by another *tercet* element and also that the global occurrences of this two elements equals to 4.

```
    <diagnostics>
        <diagnostic id="03">
            The only structure allowed for a sonnet is:
            Quatrain Quatrain Tercet Tercet
        </diagnostic>
    </diagnostics>
    <pattern name="Sonnets: Structure">
        <rule context="//poem/body">
            <assert test="not(../@style = 'sonnet') or
    ((count(quatrain) + count(tercet) = 4) and
    name(quatrain[1]/following::*) ='quatrain' and
    name(quatrain[2]/following::*) ='tercet' and
    name(tercet[1]/following::*) ='tercet')" diagnostics="03">
            </assert>
            <report test="../@style != 'sonnet'" diagnostics="00a"/>
        </rule>
    </pattern>
```

If we swapped the order of the first *quatrain* and the first *tercet* in the instance we've shown, we would receive:

```
    Sonnets: Structure
    The only structure allowed for a sonnet is:
    Quatrain Quatrain Tercet Tercet
```

The sonnets are now completely validated. However, we still need one constraint that guarantees that general stanzas are used only when and if the poem's *style* is freestyle:

```
    <diagnostics>
        <diagnostic id="04">
        The style of your poem is <value-of select="../@style"/>,
        therefore you can't use stanzas
        with only one verse or more than ten verses,
        allowed only in the case of free poems.
        </diagnostic>
    </diagnostics>
    <pattern name="gstanza occurrences">
        <rule context="//poem/body">
            <assert test="../@style = 'freestyle' or
                count(gstanza) = 0" diagnostics="04">
            </assert>
            <report test="../@style = 'freestyle'" diagnostics="00a"/>
        </rule>
    </pattern>
```

If we used a *gstanza* element in the instance we've shown, we would receive the following error message:

```

    gstanza occurrences
    The style of your poem is sonnet,
    therefore you can't use stanzas
    with only one verse or more than ten verses,
    allowed only in the case of free poems.
```

Finally, and once DTDs don't allow this kind of validation, we need a constraint to prohibit the use of the *gstanza* element for the specification of *stanzas* with 2, 3, ..., 10 verses:

```

    <diagnostics>
        <diagnostic id="05">
        The element -gstanza- was used to specify a stanza with
        <value-of select="count(verse)"/>
        verses. Instead, the appropriate element must be used.
        </diagnostic>
    </diagnostics>
    <pattern name="gstanza's use">
        <rule context="//poem/body/gstanza">
            <assert test="count(verse) = 1 or count(verse) > 10" diagnostics="05">
            </assert>
        </rule>
    </pattern>
```

If we had used the *gstanza* element instead of the *quatrain* element for the first *quatrain*, we would get the following error:

```

    gstanza's use
    The element -gstanza- was used to specify a stanza with 4
    verses. Instead, the appropriate element must be used.
```

5.3. Summary

From this case-study we may infer that, again, no major differences can be found with both the specifications. After some study and trials, all the restrictions were easily specified with both the languages.

6. Case Study 4: Database

One database is made up of various tables, each one having several registries consisting of fields. Each registry shall have a key, i.e., a field with an unique value among all the values in that table. Assuming XML is a good technology to represent DBs, we will need to assure the uniqueness of some field only in the table in which that is the key field. As the type *ID* is available for attributes, we could use it, but that attribute would have to be unique in the whole document, and that is not what we need. Therefore, we will specify a DTD for this kind of documents without the use of *ID* type of attributes and afterwards we will specify one constraint to deal with the uniqueness problem. The generic DTD, usable for any number of tables with any number of fields each, will be:

```
    <!ELEMENT DB (STRUCTURE, DATA)>
    <!ELEMENT STRUCTURE (TABLE)+>
    <!ELEMENT TABLE (COLUMNS, KEYS)>
    <!ATTLIST TABLE
        NAME CDATA #REQUIRED
    >
    <!ELEMENT COLUMNS (COLUMN)+>
    <!ELEMENT COLUMN EMPTY>
    <!ATTLIST COLUMN
        NAME CDATA #REQUIRED
        TYPE CDATA #REQUIRED
        SIZE CDATA #REQUIRED
        NULL (yes | no) #REQUIRED
    >
    <!ELEMENT KEYS (PKEYS)>
    <!ELEMENT PKEYS (PKEY)+>
    <!ATTLIST PKEYS
        TYPE (simple | complex) #REQUIRED
    >
    <!ELEMENT PKEY EMPTY>
    <!ATTLIST PKEY
        NAME CDATA #REQUIRED
    >
    <!ELEMENT DATA (items)+>
    <!ELEMENT items (items-REG+)>
    <!ATTLIST items
        NAME CDATA #REQUIRED
    >
    <!ELEMENT items-REG (FIELD)+>
    <!ELEMENT FIELD (#PCDATA)>
    <!ATTLIST FIELD
        name CDATA #REQUIRED
    >
```

Where: *DB* - root element; *STRUCTURE* - structure of the database ; *DATA* - the data itself; *items* - data of each table; *items-REG* - one registry; *FIELD*- one field.

The next document is an extract of an XML instance for a database with four tables (*stocks*, *suppliers*, *clients* and *orders*):

```
<?xml version="1.0"?> <!DOCTYPE DB SYSTEM "dbml_g.dtd">
<DB>
    <STRUCTURE>
        <TABLE NAME="stocks">
            <COLUMNS>
                <COLUMN NAME="cprod" TYPE="nvarchar" SIZE="10" NULL="no"/>
           ...
            </COLUMNS>
            <KEYS>
                <PKEYS TYPE="simple">
                    <PKEY NAME="cprod"/>
                </PKEYS>
            </KEYS>
        </TABLE>
        <TABLE NAME="suppliers">
        ...
        </TABLE>
        <TABLE NAME="clients">
            <COLUMNS>
                <COLUMN NAME="cclient" TYPE="nvarchar" SIZE="10" NULL="no"/>
                <COLUMN NAME="name" TYPE="nvarchar" SIZE="50" NULL="no"/>
                <COLUMN NAME="contact" TYPE="nvarchar" SIZE="10" NULL="no"/>
                <COLUMN NAME="account" TYPE="nvarchar" SIZE="10" NULL="no"/>
            </COLUMNS>
            <KEYS>
                <PKEYS TYPE="simple">
                    <PKEY NAME="cclient"/>
                </PKEYS>
            </KEYS>
        </TABLE>
        <TABLE NAME="orders">
            <COLUMNS>
                <COLUMN NAME="corder" TYPE="nvarchar" SIZE="10" NULL="no"/>
                <COLUMN NAME="cprod" TYPE="nvarchar" SIZE="10" NULL="no"/>
                <COLUMN NAME="quant" TYPE="nvarchar" SIZE="10" NULL="no"/>
                <COLUMN NAME="cclient" TYPE="nvarchar" SIZE="10" NULL="no"/>
            </COLUMNS>
            <KEYS>
                <PKEYS TYPE="simple">
                    <PKEY NAME="corder"/>
                </PKEYS>
            </KEYS>
        </TABLE>
    </STRUCTURE>
    <DATA>
        <items NAME="stocks">
            <items-REG>
                <FIELD name="cprod">a111</FIELD>
                <FIELD name="description">agros meio-gordo milk</FIELD>
                <FIELD name="quant">150</FIELD>
                <FIELD name="csup">f019</FIELD>
            </items-REG>
            <items-REG>
                <FIELD name="cprod">a111</FIELD>
                <FIELD name="description">ucal meio-gordo milk</FIELD>
                <FIELD name="quant">230</FIELD>
                <FIELD name="csup">f231</FIELD>
            </items-REG>
            <items-REG>
                <FIELD name="cprod">b112</FIELD>
                <FIELD name="description">ucal meio-gordo milk</FIELD>
                <FIELD name="quant">204</FIELD>
                <FIELD name="csup">f231</FIELD>
            </items-REG>
            ...
        </items>
        <items NAME="suppliers">
            <items-REG>
                <FIELD name="csup">f019</FIELD>
                <FIELD name="name">Agros, S.A.</FIELD>
                <FIELD name="address">Porto</FIELD>
            </items-REG>
         ...
        </items>
        <items NAME="clients">
            <items-REG>
                <FIELD name="cclient">c001</FIELD>
                <FIELD name="name">Corner's Cafe</FIELD>
                <FIELD name="contact">123456324</FIELD>
                <FIELD name="account">123456789012345678901</FIELD>
            </items-REG>
            <items-REG>
                <FIELD name="cclient">c002</FIELD>
                <FIELD name="name">Supermimo Supermarket</FIELD>
                <FIELD name="account">09876543210987654321O</FIELD>
            </items-REG>
            ...
        </items>
        <items NAME="orders">
            <items-REG>
                <FIELD name="corder">o012001</FIELD>
                <FIELD name="cprod">a111</FIELD>
                <FIELD name="quantity">10</FIELD>
                <FIELD name="cclient">c001</FIELD>
            </items-REG>
            <items-REG>
                <FIELD name="corder">o072001</FIELD>
                <FIELD name="cprod">b112</FIELD>
                <FIELD name="quant">20</FIELD>
                <FIELD name="cclient">c002</FIELD>
            </items-REG>
            ...
        </items>
    </DATA>
</DB>
```

This is a valid XML instance, even if several *FIELD* elements have the values *a111* and *b112* (*a111* is repeated in the *stocks* TABLE); the *contact* FIELD was forgotten in one of the *clients* TABLE's records; and the *quantity* FIELD was used instead of the correct one - *quant* - in one record of the *orders* TABLE. This means that the document is valid, however it does not reflect a valid database. Therefore, we will need to enforce several constraints in order to have usable XML documents.

6.1. Constraining with XCSL

To ensure the uniqueness of the key of each table, i.e. *cprod* attribute is unique in the branch that refers to the *stocks* TABLE, *csup* to the *suppliers* TABLE, *cclient* to the *clients* TABLE and *corder* to the *orders* TABLE, we will need four constraints, one for each sub-tree. The following constraint is to validate the first table's key uniqueness:

```
  <CONSTRAINT>
    <SELECTOR SELEXP="//items[@NAME='stocks']/items-REG
      [FIELD[@name='cprod']]"/>
    <LET NAME="keycprod" VALUE="FIELD"/>
    <CC>count(//items[@NAME='stocks']/items-REG
      [FIELD[@name='cprod'] = $keycprod]) = 1</CC>
    <ACTION>
      <MESSAGE>WARNING:
        cprod: <VALUE SELEXP="FIELD"/> is not unique!</MESSAGE>
    </ACTION>
  </CONSTRAINT>
```

We use a *LET* element to keep in *keycprod* all the values that the element *FIELD* takes in the context *//items[@NAME='stocks']/items-REG[FIELD[@name='cprod']]* (i.e. for the branch *items* with *name* attribute equal to 'stocks', every occurrence of *FIELD* for which the *name* attribute is equal to 'cprod'). After that, we count how many times each instance of *FIELD* occurs in*keycprod* list.

As we pointed out in the begining of the example, *a111* is repeated in the *stocks* TABLE. The other repetitions won't be pointed out as they occur outside the *items[@NAME='stocks']* sub-tree. Therefore, the following errors will be displayed:

```
<err-message>WARNING:
     cprod: a111 is not unique!</err-message>
<err-message>WARNING:
     cprod: a111 is not unique!</err-message>
```

The constraints for the other three tables are similar. Provided we substitute *stocks* by the name of each of the other tables and *cprod* by the name of each of the other tables' key name, the construction of the new constraints is trivial.

These constraints are not enough to completely validate our documents. We also need to validate, for each table:

- that each and every field defined in the *STRUCTURE* sub-tree is used to instantiate the records in the *DATA* sub-tree - there are no fields unused nor used more than once;
- that each and every record in the *DATA* sub-tree uses *FIELD*s identifiers defined in the *STRUCTURE* sub-tree.

For the third TABLE, *clients*, the first restriction is:

```
<CONSTRAINT>
  <SELECTOR SELEXP="TABLE[@NAME='clients']/COLUMNS/COLUMN"/>
  <LET NAME="tableclients" VALUE="@NAME"/>
  <CC>
      (count(//items[@NAME='clients']/items-REG/FIELD
        [@name = $tableclients]) =
      count(//items[@NAME='clients']/items-REG))
  </CC>
  <ACTION>
    <MESSAGE>WARNING:
      The field <VALUE SELEXP="$tableclients"/> was not used
      in every record of the
      "clients" table (or was used more than once in some record).
    </MESSAGE>
  </ACTION>
</CONSTRAINT>
```

With the *LET* element, we place in *tableclients* all the values the *NAME* attribute assumes in the context *TABLE[@NAME='clients']/COLUMNS/COLUMN* (values the *NAME* attribute assumes every time the *COLUMN* element occurs in the sub-tree *TABLE[@NAME='clients']*). After this, we count, for each *FIELD* element belonging to the sub-tree *items[@NAME='clients']*, how many times the *name* attribute occurs in the previously defined list *tableclients*. We also count the number of records (*items-REG*) of the sub-tree *items[@NAME='clients']*, to assure that each field defined for the *clients* TABLE was used as many times as the number of records described for that TABLE. The action is triggered when these two values are not the same.

In the example we are using, one of the records of the *clients* TABLE does not have the *contact* FIELD, therefore, the following error would be displayed:

```
<err-message>WARNING:
     The field contact was not used in every record of the
     "clients" table (or was used more than once in some record).
   </err-message>
```

The constraints for the other three tables are similar. Provided we substitute *clients* by the name of each of the other tables, the construction of the new constraints is trivial.

For the fourth TABLE, *orders*, the second restriction is:

```
<CONSTRAINT>
  <SELECTOR SELEXP="items[@NAME='orders']/items-REG/FIELD"/>
  <LET NAME="tableorders2" VALUE="@name"/>
  <CC>
      (count(//TABLE[@NAME='orders']/COLUMNS/COLUMN[@NAME =
        $tableorders2]) > 0)
  </CC>
  <ACTION>
    <MESSAGE>WARNING:
      The field <VALUE SELEXP="$tableorders2"/> doesn't exist
      for this table: "orders".
    </MESSAGE>
  </ACTION>
</CONSTRAINT>
```

With the *LET* element, we place in *tableorders2* the list of values the *name* attribute assumes in the context *items[@NAME='orders']/items-REG/FIELD* (values the *name* attribute assumes every time the *FIELD* element occurs in the sub-tree *items[@NAME='orders']*). After this, we count, for each *COLUMN* element belonging to the sub-tree *TABLE[@NAME='orders']*, the number of times the *NAME* attribute occurs in the previously defined list *tableorders2*. The action is triggered when this value is greater than zero, this is, the *FIELD* used in the sub-tree *items[@NAME='orders']*) was defined in the sub-tree *TABLE[@NAME='orders']*.

In the example we are using, one of the records of the *stocks* TABLE has a *quantity* FIELD instead of the *quant* FIELD that has been defined in the *STRUCTURE* sub-tree, therefore, the following errors would be displayed (notice the first error is generated by the first kind of constraint, only the last one is produced by the last constraint shown):

```
 <err-message>WARNING:
      The field quant was not used in every record of the
      "orders" table (or was used more than once in some record).
   </err-message>
 <err-message>WARNING:
      The field quantity doesn't exist for this table: "orders".
   </err-message>
```

The constraints for the other three tables are similar. Provided we substitute *orders* by the name of each of the other tables, the construction of the new constraints is trivial.

6.2. Constraining with Schematron

At this point we will need to remember that, for this family of documents, we needed three constraints for each of the four tables.

To ensure the uniqueness of the key of each table, we will need four constraints, one for each sub-tree. The following constraint is to validate the *stocks* TABLE key uniqueness (we only show diagnostics and pattern elements as the rest is trivial):

```
  <diagnostics>
      <diagnostic id="01">
          cprod: <value-of select="FIELD[@name='cprod']"/> is not unique!
      </diagnostic>
  </diagnostics>
  <pattern abstract="true" id="ch">
      <rule context="//items[@NAME='stocks']/items-REG[
        FIELD[@name='cprod']]">
          <key name="keycprod" path="FIELD"/>
      </rule>
  </pattern>
  <pattern name="Stocks table key">
      <rule context="//items[@NAME='stocks']/items-REG">
          <assert test="count(key('keycprod',
            FIELD[@name='cprod'])) = 1" diagnostics="01">
          </assert>
      </rule>
  </pattern>
```

We use two *pattern* elements. The first one stands for an abstract rule wich places in *keycprod* the list of values that the element *FIELD* takes in the context *//items[@NAME='stocks']/items-REG[FIELD[@name='cprod']]*. The other one counts how many times each instance of *FIELD* in the context *//items[@NAME='stocks']/items-REG* occurs in *keycprod* list.

The errors reported are the same they were with XCSL, but look slightly differet. We were unable to use *Schematron-report* as it returns an error because of the use of the *key* element, using therefore an alternative: *Topologi Schematron Validator*.

```
Stocks table key /DB[1]/DATA[1]/items[1]/items-REG[1]
<items-REG>...</> cprod: a111 is not unique!

/DB[1]/DATA[1]/items[1]/items-REG[3] <items-REG>...</>
cprod: a111 is not unique!
```

Just like what happened while working with XCSL, the constraints for the other three tables are similar. Provided we substitute *stocks* by the name of each of the other tables and *cprod* by the name of each of the other tables' key name, the construction of the new constraints is trivial.

We still need two more constraints for each table, as we explained before.

For the third TABLE, *clients*, the restriction to assure that each and every field defined in the *STRUCTURE* sub-tree is used to instantiate the records in the *DATA* sub-tree - there are no fields unused, nor used more than once, is:

```
    <diagnostics>
        <diagnostic id="03a">
            The field <value-of select="@NAME"/>
            was not used in every record of the
            "clients" table (or was used more than once in some record).
        </diagnostic>
    </diagnostics>
    <pattern abstract="true" id="ch">
        <rule context="items[@NAME='clients']/items-REG/FIELD">
            <key name="tableclients" path="@name"/>
        </rule>
    </pattern>
    <pattern name="Clients table (use of all the defined fields)">
        <rule context="TABLE[@NAME='clients']/COLUMNS/COLUMN">
            <assert test="(count(key('tableclients',@NAME))) =
            count(//items[@NAME='clients']/items-REG)" diagnostics="03a">
            </assert>
        </rule>
    </pattern>
```

We use two *pattern* elements. The first one stands for an abstract rule with which we place in *tableclients* the list of values the *name* attribute assumes in the context *items[@NAME='clients']/items-REG/FIELD* (values that the *name* attribute assumes every time the *FIELD* element occurs in the sub-tree *items[@NAME='clients']*). The other one counts, for each *COLUMN* element belonging to the sub-tree *TABLE[@NAME='clients']*, how many times the *NAME* attribute occurs in the list *tableclients*. We also count the number of records (*items-REG*) of the sub-tree *items[@NAME='clients']*, to assure that each field defined for the *clients* TABLE was used as many times as the number of records described for that TABLE. The action is triggered when these two values are not the same.

In the example we are using, one of the records of the *clients* TABLE does not have the *contact* FIELD, therefore, the following error would be displayed:

```
Clients table (use of all the defined fields)
/DB[1]/STRUCTURE[1]/TABLE[3]/COLUMNS[1]/COLUMN[3]
<COLUMN NAME="contact" TYPE="nvarchar" SIZE="10" NULL="no">...</>
The field contact was not used in every record of the "clients"
table (or was used more than once in some record).
```

The constraints for the other three tables are similar. It is enough to substitute *clients* by the name of each of the other tables.

For the fourth TABLE, *orders*, the restriction needed to assure that each and every record in the *DATA* sub-tree uses *FIELD*s identifiers defined in the *STRUCTURE* sub-tree, is:

```
<diagnostics>
    <diagnostic id="04b">
        The field <value-of select="@name"/>
        doesn't exist for this table: "orders".
    </diagnostic>
</diagnostics>
<pattern abstract="true" id="ch">
    <rule context="TABLE[@NAME='orders']/COLUMNS/COLUMN">
        <key name="tableorders2" path="@NAME"/>
    </rule>
</pattern>
<pattern name="Orders table (inexistent fields)">
    <rule context="items[@NAME='orders']/items-REG/FIELD">
        <assert test="count(key('tableorders2',@name)) = 1"
        diagnostics="04b">
        </assert>
    </rule>
</pattern>
```

We use two *pattern* elements. The first one stands for an abstract rule with which we place in *tableorders2* the list of values the *NAME* attribute assumes in the context *TABLE[@NAME='orders']/COLUMNS/COLUMN* (values that the *NAME* attribute assumes every time the *COLUMN* element occurs in the sub-tree *TABLE[@NAME='orders']*). The other one counts, for each *FIELD* element belonging to the sub-tree *items[@NAME='orders']*, how many times the *name* attribute occurs in the list *tableorders2*. Each *name* attribute shall belong to the list of fields defined in the sub-tree *TABLE[@NAME='orders']* (and once).

In the example we are using, one of the records of the *stocks* TABLE has a *quantity* FIELD instead of the *quant* FIELD that has been defined in the *STRUCTURE* sub-tree, therefore, the following errors would be displayed (notice the first error is generated by the first kind of constraint, only the last one is produced by the last constraint shown):

```
Orders table (use of all the defined fields)
/DB[1]/STRUCTURE[1]/TABLE[4]/COLUMNS[1]/COLUMN[3]
<COLUMN NAME="quant" TYPE="nvarchar" SIZE="10" NULL="no">...</>
The field quant was not used in every record of the "orders" table
(or was used more than once in
some record).
```

```
    Orders table (inexistent fields)
    /DB[1]/DATA[1]/items[4]/items-REG[1]/FIELD[3]
    <FIELD name="quantity">...</>
    The field quantity doesn't exist for this table: "orders".
```

The constraints for the other three tables are similar. We just need to substitute *orders* by the name of each of the other tables.

6.3. Summary

For this case-study, transforming XCSL constraints in Schematron constraints or the other way round is not linear. The perspective in which we have to think, in order to write this kind of constraints, differs from one to another. With XCSL we grab in the list all the values pertaining to the "origin" branch, whereas with Schematron we grab in the list all the values pertaining to the "destiny" branch, i.e, if we want to check if all the values defined in the X sub-tree were used in the Y sub-tree, we create the XCSL's list with the X branch values and the Schematron's list with the Y branch values.

We set up the Schematron lists of values in independent pattern elements, what means that, apart from the inconvenience of having to define two pattern elements, makes it possible to define global variables. XCSL on the other hand, doesn't allow this, meaning that we may use exactly the same names for every constraint (i.e, each *CONSTRAINT* element) whereas with Schematron we can not.

7. Conclusion

In this paper we briefly introduced three constraint languages - XCSL, Schematron and XML-Schemas -, and used four real life case-studies to compare the two first approaches.

Those examples were chosen from a much larger set of documents we have been working with. The complete report is undergoing and will be published shortly.

This study allowed us to find the answers to the questions we faced in the beginning:

Do they do the same job?

According to all that we've shown, yes, XCSL and Schematron can do the same job.

Are there some kind of constraints that are easier to specify withone of them?

The kind of constraints for which it is useful to use variables are clearly simpler to specify with XCSL once, at the present moment, it has the advantage of allowing the use of variables what significantly shortens the number of lines that we need to write and, consequently, errors become less likely to occur. The rest of the constraints differ on the mental structure of each person: with XCSL you start writing a constraint and do it all at once using the CONSTRAINT element whereas with Schematron (unless you don't need to report the value some element or attribute assumes in a particular instance), you need to use two elements - pattern and diagnostics. Even so, all XCSL constraints are less verbose than Schematron's ones.

Globally, XCSL looks easier to understand, learn and use than Schematron.

Do you need different background to use the tools?

The only thing that differs from one to another is the particular XML language each one has. To validate documents using one of the tools, all that you have got to learn is its DTD (or Schema).

Is it possible to use them in similar situations (the same DTD, the same XML instances)?

Unless for the huge amount of lines we need to use when constraining the referred cases with Schematron, it looks like it is possible to specify every constraint with both the tools.

May we use them to produce an equal result?

Although Schematron is much more verbose, we were able to produce the same kind of results with both the tools.

Schematron is clearly more complex than XCSL and even if it is true that the first one has some possibilities inexistent in the last one (like documentation or the ability of entitling the whole restrictions' document), it is also true that maybe the over effort to learn all those facilities overweighs the advantage of using them.

How do XCSL and Schematron relate to XML Schemas?

Some constraints we specified with them may be specified by XML Schemas, however, there are some constraints that we can't specify with XML Schemas. Therefore, we may still need to use both an XML Schema and a constraint document (written in XCSL or Schematronlanguages).

What is the intersection area of these three?

Domain Range, mixed content and cardinality constraints.

What kind of constraints each one of these three is able to specify?

Supposing we covered all possible kinds of constraints, and we believe we did, we may constrain Domain Range, mixed content and cardinality constraints with XML Schemas and all kinds of constraints both with XCSL and Schematron.

Bibliography

[RAM 2000] José Carlos Leite Ramalho, 2000. *Anotação Estrutural de Documentos e sua Semântica*, Universidade do Minho - Portugal.

[RAM 2001] Ramalho, José C. and Henriques, Pedro R., 1998. *Constraining Content: Specification and Processing*, XML Europe'2001, Internationales Congress Centrum (ICC), Berlin, Germany.

[DOD 2001] Dodds L., 2001. *Schematron: Validating XML Using XSLT*, XSLT UK Conference, Keble College, Oxford, England.

[TON 2000] Claus Tondering, 2000. *Frequently Asked Questions about Calendars - Version 2.3*, in http://www.tondering.dk/claus/calendar.html.

Glossary

XCSL,	XML Constraint Specification Language

Biography

Marta Henriques **Jacinto**
University of Minho
Braga
Portugal
Email: marta.jacinto@itij.mj.pt

With a degree on Applied Mathematics and Computation, Marta Jacinto is currently working for ITIJ - the Computer Department of the Ministry of Justice as Systems Engineer.

As a researcher, she is working on her Master thesis under the subject "Semantic Validation of XML Documents".

Giovani Rubert **Librelotto**
University of Minho
Braga
Portugal
Email: grl@di.uminho.pt

Giovani Librelotto, has a degree and a Master on Computer Science. He is currently researching on XML and Topic Maps and is preparing his Ph.D. thesis.

José Carlos Leite **Ramalho**
University of Minho
Braga
Portugal
Email: jcr@di.uminho.pt

J. C. Ramalho is an Auxiliary Professor at the Computer Science Department of the University of Minho.

He has a Masters on "Compiler Construction" and a Ph.D. on the subject "Document Semantics and Processing". He has been managing several XML projects and consulting.

Pedro Rangel **Henriques**
University of Minho
Braga
Portugal
Email: prh@di.uminho.pt

Pedro Henriques is an Associated Professor of Computer Science at University of Minho.

His research and teaching activity has been concerned with programming in general - paradigms, specification formalisms and languages; in particular, his main interest is the development of language processors.

He completed, some years ago, his Ph.D. at University of Minho in the area of Attribute Grammars; he is, now, the leader of the "Language Specification and Processing" group. The application of the "grammatical approach to problem solving" and the use of "parsing and semantic analysis technologies" in various problem domains (namely, document processing, information retrieval and data/text mining, and geographical information systems) are the present concerns of his academic work.