CALS Table Processing with XSLT and Schematron

Nigel Whitaker

DELTAXML

Conference Paper: XML London

ABSTRACT

CALS tables are used in many technical documentation standards. There are OASIS specifications for CALS tables which include a number of semantic rules to ensure table validity. This paper reports on some of our experiences with CALS table processing and validation. We implemented the majority of our validation code in XSLT and have needed to carefully consider performance when handling large tables of several thousand rows.

We have experimented with a number of new XSLT features when addressing performance issues and will report on our experiences. In addition to processing tables we wished to improve the quality of CALS tables that we would meet and which our users/customers would produce (we wished to rid the world of bad tables!). For this we have used schematron to check and report the validity of tables in a user friendly way. We met a number of obstacles and will report on these and our solutions ('work-arounds') in this paper.

CONTENTS

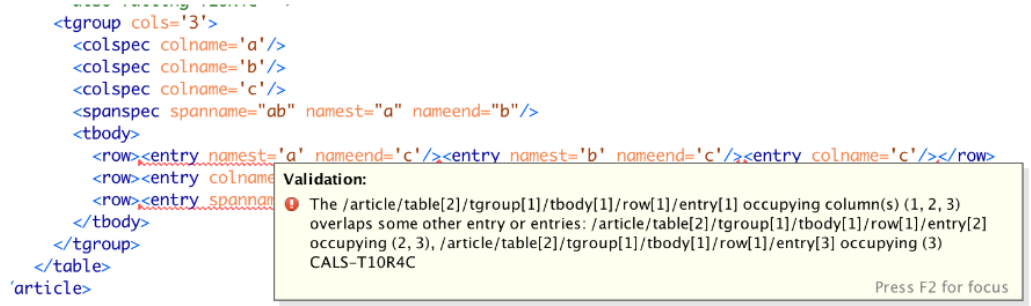
Background

CALS tables are used in many technical documentation standards such as DocBook, DITA and s1000d. Unfortunately these document standards refer to slightly different versions of the CALS specification (the 'exchange subset' and the full specification [1]) and they have slightly different messages about validity.

Different XML tools and processors (such as XSL:FO to PDF engines) enforce different subsets of the semantic rules. Our goal for comparison processing was to generate a valid result if the inputs were valid. In order to do this we needed a way of checking validity. We looked for something that we could reuse as part of our test processes and also internally as part of the processing of the tables. Table processing is a widely used example in schematron [2] tutorials and blog postings, however we did not find anything that was close to being a complete implementation of the CALS specification(s). We therefore decided to construct one.

Our CALS validity code is Open Source software (originally available on GoogleCode, now GitHub [3]). We provided it as open source as a number of collaborators were asking us about it. Additionally, customers were questioning our error/warning reporting and not fully understanding the nature of the validity issues. Using schematron allows us to provide good diagnostics to the user, typically in an XML editor or authoring tool, and we hope this would help reduce the number of bad tables, finding the structural and other table errors as they are created. Providing Open Source code would improve use and adoption. The oXygenXML editor supports schematron validation and **Figure 1, "Example of table validation error"** shows how an error is reported.

Figure 1. Example of table validation error



Introduction to table validity

CALS tables allow for 'wide' cells or entrys. Rather than a regular grid there are cells that 'span' a number of grid positions. Some simpler table models such as those for HTML tables use attributes with integer dimensions to describe the wide/tall cells. CALS tables use the *@morerows* attribute with an integer for vertical spanning, but use a more complex structure for horizontal spanning. A column specification section is declared and then the later cells reference these *colspec* and similar elements by name in order to configure their spanning.

The CALS rules cover a number of areas. Here are some examples.

Context constraints

The CALS specification include a number of constraints about the context in which an attribute can occur. For example, a wide (horizontally spanning) `entry` can be specified by referring to the start and ending `colspecs` (the `@namest` and `@nameend` attributes). It is also possible to define a `spanspec` element at the same time as the colspecs which itself refers to the start and end colspecs. It is permissible for an entry to refer the `spanspec` (using `@spanname`) in the main body of the table. However using a `spanname` attribute is not allowed in the context of the `thead` and `tfoot` elements used to define the headers and footers of a table.

Referential integrity constraints

References to `colspec` and `spanspec` elements from within a table have to be validated. These references have a per-table scope and therefore do not make use of the id or `xml:id` mechanisms.

Structural constraints

There are a number of structural constraints in the CALS spec. These tend to be orientated to making the table regular (so that for example each row has the correct number of columns occupied or that a `morerows` vertical span does not extend beyond the end of the table. Other constraints are used to ensure correct left-right ordering and to prevent overlap.

As discussed above attributes can be used to specify wide table entrys[1]. They can also be used simply position an entry in the table. Some editors and authoring tools will refer to a `colspec` in every entry in the table. However, its also possible to use an entry without specifying its position in which case its position has to be inferred and this style is used by other tools. The rules are quite complex and consider the position of the preceding entry element but also tall entrys that use `morerows` attributes in the rows above. The situation may be quite recursive in that an entry will use `morerows` but in order to work our which columns it occupies then it's necessary to look to all the entrys to its left to see if they define their position or infer their position by default rules and its also necessary to look at rows from above that 'straddle' or 'overlap' into the current row at positions before where the entry may be placed by the default rules.

XSLT processing of CALS tables

Row distance calculation

In calculating and checking for vertical spanning we originally wrote code that measured the distance between rows. The initial naive implementation is shown in **Figure 2, "Original row-distance code".**

Figure 2. Original row-distance code

```
<xsl:function name="cals:row-distance"
              as="xs:integer">
  <xsl:param name="r1" as="element()"/>
  <xsl:param name="r2" as="element()"/>
  <xsl:sequence
    select="abs(count($r1/preceding-
    sibling::*:row) - count($r2/
    preceding-sibling::*:row))"/>
</xsl:function>
```

This is code at least O(n) and the calling code, discussed later, made for $O(n^2)$ complexity. From our analysis we knew this function was called a lot, over 10^8 invocations in one of our tests.

Customer feedback reported performance concerns and the need to process larger tables. As it was a known 'hot-spot' we concentrated on optimizing this function. This was around the time of Saxon version 9.3 and maps were used[2]. This optimized code is shown in **Figure 3, "Optimized row-distance code".**

Figure 3. Optimized row-distance code

```
<xsl:variable name="rowtopos"
              as="map(xs:string,
              xs:integer)">
  <xsl:map>
    <xsl:for-each select="//*:row">
      <xsl:map-entry key="generate-id(.)"
      select="
        count(./preceding-sibling::*:row)
        + 1"/>
    </xsl:for-each>
  </xsl:map>
</xsl:variable>
<xsl:function name="cals:row-distance"
              as="xs:integer">
  <xsl:param name="r1" as="element()"/>
  <xsl:param name="r2" as="element()"/>
  <xsl:sequence select="
  abs(map:get($rowtopos, generate-
  id($r1)) - map:get($rowtopos, generate-
  id($r2)))"/>
</xsl:function>
```

This optimization reduced a 5 minute plus runtime to the 15-20 second range. For further details see **Section, "Performance summary"**.

Vertical column infringement processing

[1] As entry is an element name we have chosen to deliberately misspell this plural form.
[2] An accumulator would now be a better choice and would avoid the use of generate-id to form the map key.

The code shown in **Figure 4, "vertical infringement code"** was used to calculate which columns of a row are overlapped or infringed from above.

Figure 4. vertical infringement code

```
<xd:doc>
  <xd:desc>
    <xd:p>Describes how a table row is
    spanned from above.</xd:p>
    <xd:p>This result is a set of columns
    which are overlapped from above in
    the row specified as an argument. The
    'set' is really a sequence and may be
    out of order, eg: (3, 2).</xd:p>
  </xd:desc>
  <xd:param name="row">A table row</
  xd:param>
  <xd:return>A sequence of integers
  specifying which columns are spanned or
  'infringed' from above</xd:return>
</xd:doc>
<xsl:function name="cals:overlap2"
as="xs:integer*">
  <xsl:param name="row" as="element()"/>
  <xsl:sequence select="
  for $r in $row/preceding-sibling::*:row
  return
  let $row-distance :=
    cals:row-distance2($r, $row) return
    for $e in $r/*[@morerows] return
    if (xs:integer($e/@morerows) ge $row-
    distance
      then cals:entry-to-columns($e)
      else ()"/>
</xsl:function>
```

The above code reflects our XSLT 2.0 training and experience. For each row we look upwards and see if any of the rows above could infringe the current row. This process is O(n) and makes repeated use of the `row-distance` function above. It also uses the `entry-to-columns` function which for a table entry reports which columns are occupied.

There are several issues we knew about when writing this code that we were aware could cause performance issues:

- We will be using this function repeatedly and each time it is called it will look all the way back through the table.
- It looks all the way back to the top of the table since theoretically it is possible for the first row to have a morerows span to the end of the table. In common cases it's likely that spanning would be short, however doing an analysis of the maximum morerows value and the using this to optimize the code would have made some complex code even more complicated and difficult to maintain.

Forward looking morerows processing

The processing of morerows attributes could be attempted in forward looking manner. An `xsl:iterator` was used to make a morerows calculation using a sequence of integers. Each member of the sequence would store the current morerows value for its corresponding column as the iterator would allow it to be decremented as each subsequent row was processed, provided it did not also use morerows.

Figure 5. The morerows iterator

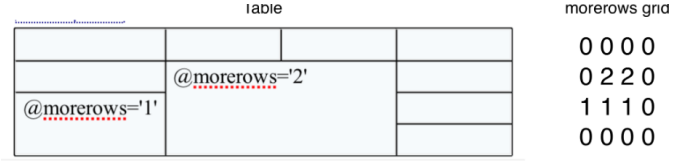
```
<xsl:iterate select="$tgroup/*:row">
  <xsl:param name="morerows"
  as="xs:integer*" select="for $i in 1 to
  $tgroup/@cols return 0"/>
  <xsl:param name="grid" select="map{}"
  as="map(xs:integer, xs:integer*)"/>
  <xsl:on-completion select="$grid"/>
  <xsl:variable name="rowmap"
                as="map(xs:integer,
                xs:integer)">
    <xsl:map>
      <xsl:for-each select="entry[@
      morerows]">
        <xsl:variable name="coveredCols"
        as="xs:integer+" select="
           cals:entry-to-columns(.,
           $morerows)"/>
        <xsl:sequence select="
          map:merge(for $i in
          $coveredCols return
          map:entry($i, xs:integer(@
          morerows)))"/>
```

```
      </xsl:for-each>
    </xsl:map>
  </xsl:variable>
  <xsl:next-iteration>
    <xsl:with-param name="morerows"
    select="for $i in 1 to
            count($morerows) return
            max(($morerows[$i]-1,
            $rowmap($i),0))"/>
    <xsl:with-param name="grid"
      select="map:merge(($grid,
      map:entry(
        count(preceding-
        sibling::*:row)+1,
        $morerows)))"/>
  </xsl:next-iteration>
</xsl:iterate>
```

In **Figure 5, "The morerows iterator"** the `morerows` param stores the sequence that is adjusted as each row is iterated over. Additionally, we wanted to store these values and the grid param records each of the `morerows` calculations using a map where the map key is an integer corresponding to the row number.

The `rowmap` calculates the `morerows` values declared in the current row (it maps from column number to the morerows value). In order to do this knowledge of the morerows spanning from above is needed to determine the column positions of any entries which do not define their columns by explicit reference to colspecs.

Figure 6. Example table and morerows grid



An example table and the corresponding morerows grid is shown in **Figure 6, "Example table and morerows grid"**.

The iterator that has been developed now needs to be used. If our intention was, for example, to produce a normalized form of the table then it should be possible to use the iterator in an `xsl:template` matching the `tgroup` or `table`.

However, we are keen to preserve the ability to have checking for individual entries in the table and error reporting specific to those entries. We could use the iterator in a function after finding the parent table, but then we would be iterating repeatedly over the same table. In order to use the same data we then looked at accumulators.

Caching the table data for use in schematron

After discounting using the iterator in a template we tried to find ways of keeping the data available. Thinking in terms of imperative programming you would write:

```
if (empty map) then {
  construct map; // done once
}
return lookup data in map;
```

However this doesn't fit well with either XSLT or schematron. We found a solution using the new `xsl:accumulator` mechanism (once we realized an accumulator doesn't actually need to accumulate anything!).

If we put our accumulator in a function, as indicated in **Figure 7, "Function with iterator"**, we can then use the iterator in an accumulator as shown in **Figure 8, "Storing data in an accumulator"**. Another accumulator is used to provide a mapping from the rows to the row numbers as this is what is used to index the map representing our grid data. The use of a sequence in this accumulator is designed to support the possibility of nested tables.

Figure 7. Function with iterator

```
<xsl:function name="cals:generate-
morerows-data"
     as="map(xs:integer, xs:integer*)">
  <xsl:param name="tgroup"
  as="element()"/>
  <xsl:iterate select="$tgroup/*:row">
    <xsl:param name="morerows" select="
    for $i in 1 to $tgroup/@cols return
    0" as="xs:integer*"/>
```

```
    <xsl:param name="grid"
               as="map(xs:integer,
               xs:integer*)"
               select="map{}"/>
    <xsl:on-completion select="$grid"/>
    ...
  </xsl:iterate>
</xsl:function>
```

Figure 8. Storing data in an accumulator

```
<xsl:accumulator name="table-spanning"
    as="map(xs:integer, xs:integer*)"
    initial-value="map{}">
  <xsl:accumulator-rule match="*:tgroup"
    phase="start"
    select="cals:generate-morerows-
    data(.)"/>
</xsl:accumulator>

<xsl:accumulator name="row-number"
as="xs:integer*" initial-value="()">
  <xsl:accumulator-rule match="*:tgroup"
  phase="start" select="(0, $value)"/>
  <xsl:accumulator-rule match="*:tgroup"
  phase="end" select="tail($value)"/>
  <xsl:accumulator-rule match="*:row"
  select="head($value)+1, tail($value)"/>
</xsl:accumulator>
```

Using the accumulator data in schematron

We can now use the data in various ways. One technique is to create functions that use the data by calling the `accumulator-after` or `accumulator-before` functions. It is also possible, when using the XSLT query binding and foreign element support, to use the data directly in schematron. In order to check CALS tables there are three assertions that check the table structure that operate on every table entry. We can use some foreign XSLT code at the the schematron rule level so that they are available to all of the assertions. The code is quite long so only the basic mechanism is shown in **Figure 9, "Accessing accumulator data from schematron**".

Figure 9. Accessing accumulator data from schematron

```
<pattern id="p-structure">
  <rule context="*:entry">
    <xsl:variable name="row"
    as="element()"
      select="ancestor::*:row[1]"/>
    <xsl:variable name="table-data"
      as="map(xs:integer, xs:integer*)"
      select="$row/ancestor::*:tgroup[1]/
      accumulator-after('table-
      spanning')"/>
    <xsl:variable name="row-number"
    as="xs:integer" select="$row/
        accumulator-after('row-
        number')"/>
    <xsl:variable name="morerows"
    as="xs:integer*" select="$table-
    data($row-number)"/>
    <assert
      test="... cals:entry-to-columns(.,
      $morerows)
           ... ">...</assert>
    <assert test="...">...</assert>
    <assert test="...">...</assert>
  </rule>
</pattern>
```

Every table entry will invoke the assertions above, but in each case we rely on the pre-calculated or accumulated data. This should change an $O(n^3)$ complexity problem into one closer to O(n).

Performance summary

Two customer support cases presented particular problems with table performance. The earlier row- distance optimization with maps gave performance improvements at the time, but there were still performance doubts and concerns. The optimization work with iterators and accumulators discussed here have provided dramatic performance improvements as seen in **Table 1, "Performance data"**.

The results discuss run time improvements. There was no obvious change in memory consumption.

Table 1. Performance data

	case 1	case 2
description	Standards documentation	Semiconductor data
file format	DocBook 5	DITA 1.1
file size	8.7MB	574KB
tree details	459262 nodes, 656993 characters, 208 attributes	31670 nodes, 41285 characters, 10384 attributes
table count	15	3
average row count	1083	1149
largest table (rows)	2032	1552
Saxon EE 9.7.0.4 performance: Apple iMac, 3.2 GHz Intel Core i5, 24GB, MacOS 10.11.4, Java 1.8.0_74 64 Bit Server VM		
original runtime	greater than 11 hours	348.49s
row-distance map optimized runtime	1557.52s	16.09 s
iterator/ accumulator optimized runtime	2.20s	0.152s

Schematron processing

We have two ways of using the schematron file. The first, standard, way is to use it as a file checker, perhaps inside an XML editor or authoring tool as discussed earlier. Additionally we are interested in using checking code as part of our comparison products which required some modifications to the to the standard schematron processing model and tool-chain. As schematron processing is based on XSLT (the schematron is 'compiled' to XSLT using XSLT) we've taken the approach of using further XSLT to modify the generated checker to satisfy our requirements. This approach works while the generated checker is stable, but could cause us problems if new version of the schematron code, such as the 'skeleton', is released. One reason for describing our requirements and changes here is to gauge if there is wider interest in them. If so, we could see if there is acceptance to have them incorporated into the official release.

Schematron phases

When developing the CALS validity constraints a misconception about schematron phases was not noticed until the final stages of testing. We followed a development model similar to that of traditional compilers where checking is performed in stages. Similar to a compiler constraints were grouped into categories such as: context, reference and structure. When developing the structural constraints referential integrity was assumed. We developed and tested those constraints first and assumed the implementation wouldn't run those constraints if the referential integrity constraints fail. Instead schematron phases seem to be something that is left to user control - the user is often presented with a list of phases and asked which to run.

Alternatives were considered. Adding conditional code to the later constraints to test the earlier one and therefore prevent failures in the execution of complex structural constraints was a possibility, but would add more complexity and make the code harder to maintain. We decided to modify the execution model of the generated schematron checker. The XSLT which is generated uses modes to (which are then template- applied to the root node '/' in turn) which correspond to the various phases. A very simple, but naive, initial modification used saxon:assign to record any failure in a global variable which was then tested between the apply- templates for the various stages.

While the saxon:assign approach worked it was not elegant. Advice was sought from the schematron-users email list and David Carlisle suggested an alternative algorithm using a nested structure of xsl:if tests, variables and apply-template instructions.

The new xsl:try/xsl:catch mechanism may provide a better mechanism for our preferred phasing and may be investigated.

Issues with schematron granularity

Current Schematron tools focus on validating an entire XML file. This is what is needed in an editor or authoring tool. For use in our comparator products we needed something slightly different. When comparing and aligning a table we make decisions on how the result will be represented partly based on whether the input tables were valid. The requires a knowledge of the validity of a table (we add a valid attribute during input processing) rather than the whole file. Schematron defines an XML format, the Schematron Validation and Reporting Language (SVRL), to describe the validation results. Unfortunately this format is a flat list of validation results. While it does contain XPaths which point to the location of failures it is a hard process to group these and associate them with source tables. We took a different approach and again adjusted (by post processing the generated XSLT) the control flow so that rather than rely on template matching form the root or / element downwards a set of phases are applied to each table and then processing moves to the next table.

If other schematron users have similar requirement it may be appropriate to share code, approaches or discuss alternatives further.

Acknowledgements

The author has been assisted by colleagues at DeltaXML Ltd who have also contributed to the code and associated tests. This is why the first-person form ‘we’ is used in this paper.

We would also like to thank Dr John Lumley who has been assisting DeltaXML with performance profiling and optimization approaches with our XSLT code. He suggested that an iterator would improve the performance of the CALS checking code and the results presented above confirms this.

David Carlisle and other members of the schematron-users email list have helped with the schematron issues we have discussed above and we are very gratefully for their advice.

References

[1] Harvey Bingham (ed.) CALS Table Model Document Type Definition. OASIS Technical Memorandum TM 9502:1995. . OASIS Inc.. 1995. https://www.oasis-open.org/specs/tm9502.html

[2] ISO/IEC 19757-3:2006 Document Schema Definition Languages (DSDL) — Part 3: Rule-based validation — Schematron. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=40833

[3] CALS Table Schematron. Github repository. https://github.com/nigelwhitaker/cals-table-schematron

Head Office (Sales and Support)

DeltaXML Ltd
Malvern Hills Science Park
Malvern, Worcestershire
WR14 3SZ UK

W: www.deltaxml.com
E: info@deltaxml.com
T: +44 1684 532 130