Journal Article Tag Suite subset and Schematron: achieving the right balance

Alexander B. Schwarzman

The Optical Society, Washington, DC, USA

Abstract

Ensuring that published content adheres to the publisher's business and style rules requires the implementation of quality-control solutions that encompass the entire enterprise, including vendors and in-house staff. The solutions must span the entire life cycle of the manuscript, from extensible markup language conversion to production to post-publication enhancements. Two techniques that may help in achieving this goal are developing Schematron and making a Journal Article Tag Suite subset. Both come with costs: Schematron change management requires development and maintenance of an extensive test-base; making a subset requires comprehensive content analysis and the knowledge of the publishing program's direction. Achieving the right balance between the two techniques may reduce the costs associated with them.

Keywords

Journal Article Tag Suite subset; Journal publishing tag; Schematron; XML

Introduction

In this paper, we revisit the notion of "appropriate layer validation" at the current state of technology. We share the experience of running a successful large-scale quality-control operation that has been accomplished by using a combination of Journal Article Tag Suite (JATS) subset and Schematron. After demonstrating what Schematron change management entails, analyzing the advantages and costs associated with building Schematron and with creating a subset, and considering several validation scenarios, we conclude with the suggestion that the two techniques, when used in tandem, may complement one another and help control software development costs.

Schematron Change Management: An Introductory Example

Let us take a real example from a production Schematron. Consider the XML markup below:

Received: December 20, 2017
Accepted: January 31, 2018

Correspondence to
Alexander B. Schwarzman
aschwarzman@osa.org

ORCID
Alexander B. Schwarzman
<https://orcid.org/0000-0002-6485-4158>

It is the secondary publication originally presented in Journal Article Tag Suite Conference (JATS-Con) Proceedings 2017, available from: <https://www.ncbi.nlm.nih.gov/books/NBK425543/>.

```

<fig id="g005">
  <label>Fig. 5.</label>
  <caption><p>Diffraction efficiency of gratings.</p></caption>
  <graphic xlink:href="ao-55-18-4952-g005a"/>
</fig>

```

We need to implement the following requirement: “The last segment of an image filename (i.e., of the value of the `xlink:href` attribute on the element `graphic`) must correspond to the sequence number of the parent figure. For example, for the fifth figure in the body, the numeric part of the last segment of the image filename; i.e., ‘-g005a’, must be 5. We will make an exception for correction article type, and we will only check this requirement at the final production stage.”

This is how we will implement the requirement:

```

1 <rule id="FIGT140" context="graphic[parent::fig][not($module-code = 'TJI')]">
2
3   <d:req>The last segment of the graphic name (i.e., of the value of its xlink:href
4     attribute) must correspond to the sequence number of its parent figure within the
5     article's body. E.g., for the fifth figure in the body, the numeric part of the
6     last segment for its graphic '-g005a' must be 5.</d:req>
7
8   <d:note>We only check, via context, the graphic elements whose parent is fig.</d:note>
9
10  <d:note>Exception is made, via context, for Traditional Journals Initial (TJI)
11    validation.</d:note>
12
13  <d:func>We get the sequence number of the preceding fig element.</d:func>
14
15  <let name="fig-sequence" value="count(preceding::fig) + 1"/>
16
17  <d:func>We obtain a numeric part of the last segment for the graphic name.</d:func>
18
19  <let name="graphic-number"
20    value="number(replace(tokenize(@xlink:href, '-')[last()], '^[0-9]', ''))"/>
21
22  <d:note>Exception is made for Corrections.</d:note>
23
24  <assert
25    test="$graphic-number = $fig-sequence or /article/@article-type = 'correction'"
26    expecting '<value-of select="tokenize(@xlink:href, '-')[last()]" />' in graphic
27    name '<value-of select="@xlink:href" />' to contain number '<value-of
28      select="$fig-sequence" />'</assert>
29 </rule>

```

How can we make sure that this rule works as it is supposed to? We need to write two kinds of tests: ‘Go’ tests, which contain markup that should pass; and ‘NoGo’ tests, which contain markup that should trigger the error message specified in the assert statement. In this case, we have several Go tests, including: an article that contains one figure with a correctly named graphic; an article that contains multiple figures, each with correctly named graphics; a correction article that contains a figure whose graphic name does not have to follow the rule; an example in which a graphic is a child of a biography or of a boxed text rather than of a figure; and an article in which image filenames do not conform to the rule, but the rule doesn’t fire because the article is checked at the initial (as opposed to the final) production stage. We also have several NoGo tests, including: multiple examples, each with differently misnamed graphics (you may have seen such examples in the XML supplied by your conversion vendors), to make sure our regular expression catches the most frequently occurring variations of incorrect image names; and articles of types other than correction, i.e., review or introduction, with misnamed graphics.

Obviously, developing and maintaining such tests requires time and effort. But can we assume that once we have developed

and run the tests, and made sure our rule works as expected, we can forget about the tests? To answer this question, let us consider a couple of examples of what happens when Schematron code changes. First, in Schematron, each rule sets a context in which it operates. If two rules have the same context and a test in the first rule fires, then the tests in the second rule will not operate. Now, suppose your Schematron has dozens of rules, and in the course of development you accidentally added another rule with the same context. You will not discover the problem unless you run all your NoGo tests and see that some of them no longer generate an expected error. Second, the Schematron snippet above is self-documented: it contains an English-language description of requirements, exceptions, and functionalities. But this is not always the case. Suppose the Schematron code is not self-documented, and someone doing code review and cleanup decides that the predicate [parent::fig] in the rule's context is not necessary and removes it. You will not discover that this is a problem unless you run all your Go tests, including the one in which a graphic element is a child of a non-figure element, such as a biography bio. Then you will see that the test that used to pass now fails and throws an error, even though it should not.

What these examples make clear is that every time you make a change to the Schematron, you have to run all your tests to

ensure that all Go tests pass and all NoGo tests generate the errors they are supposed to. If a Go test fails or if a NoGo test does not throw the expected error then you need to investigate the problem in your Schematron code and correct it.

	EJI	EJF	TJI	TJF
figures.sch [FIGT]	✓	✓	✓	✓
funding.sch [FUND]	✓	✓	✓	✓
mathml.sch [MATH]	✓	✓	✓	✓
...				
meta-expr.sch [MTEX]	✓	✓		
meta-trad.sch [MTTR]			✓	✓
meta.sch [META]	✓	✓	✓	✓
...				
ref-meta.sch [RFMT]	✓	✓	✓	✓
sup-mat.sch [SPMT]	✓	✓	✓	✓
tables.sch [TBLE]	✓	✓	✓	✓

Fig. 1. Schematron: modular architecture.

Fig. 2. Individual module structure.

Schematron Change Management: Large-scale Schematron

Modular Schematron architecture and configuration

Let us now consider a reduced example of a large-scale production Schematron that consists of four top-level Schematron instances and a modular library of 25 files. Each top-level Schematron instance is designed to check a certain kind of document, e.g., Express journal (EJ) or Traditional journal (TJ) article, at either the initial (I) or the final (F) stage of production. Modules in the library reflect an article structure (metadata, references), article components (figures, tables), areas of special concern (funding, MathML), journal type (Traditional, Express) or some other organizing principle. A top-level Schematron instance invokes only those library modules that are appropriate to validate the respective kind of doc-

ument. Fig. 1 shows a grid illustrating a modular Schematron architecture. Each Schematron module in the library comprises a number of rules, where a rule defines a context and contains assert and (or) report statements testing certain conditions within the rule's context. The library also includes a number of supporting XSLT modules with variable, function, and key declarations used in the Schematron modules.

Fig. 2 shows an example of one rule in the sup-mat.sch module, which contains the requirements dealing with validation of supplementary materials. The rule sets a context and contains an assert statement and a report statement that test certain conditions.

Schematron testbase: Go and NoGo tests

As demonstrated in the introductory example, to verify that assert and report statements work properly we need to devel-

Fig. 3. Go test.


```

1 <article xmlns:xlink="http://www.w3.org/1999/xlink">
2   <front>
3     <journal-meta>
4       <journal-id journal-id-type="publisher-id">OPTICA</journal-id>
5     </journal-meta>
6     <article-meta>
7       <article-id pub-id-type="doi">10.1364/OPTICA.1.000001</article-id>
8       <volume>1</volume>
9       <issue>1</issue>
10      <fpage>1</fpage>
11      <!-- SUPPLEMENTARY MATERIAL METADATA -->
12      <supplementary-material id="s001" content-type="supplemental-document">
13        <object-id pub-id-type="doi">10.1364/OPTICA.1.000001.s001</object-id>
14        <label>Supplement 1</label>
15        <long-desc>Supplementary document with expanded equations.</long-desc>
16        <media xlink:href="optica-1-1-1-s001.PDF"/>
17      </supplementary-material>
18      <supplementary-material id="s002" content-type="supplemental-document">
19        <object-id pub-id-type="doi">10.1364/OPTICA.1.000001.s001</object-id>
20        <label>Supplement 2</label>
21        <long-desc>Supplementary document with expanded equations.</long-desc>
22        <media xlink:href="optica-1-1-1-s002.PDF"/>
23      </supplementary-material>
24    </article-meta>
25  </front>
26  <body>
27    <sec id="sec1">
28      <p>See <!-- SUPPLEMENTARY MATERIAL IN-TEXT CITATION WITH LINK -->
29        <inline-supplementary-material content-type="supplemental-document"
  
```

E [ISO Schematron] ERROR [spmt-test:SPMT130]: DOI '10.1364/OPTICA.1.000001.s001' is listed more than once

Fig. 4. NoGo test.

Go	NoGo
BACK-sequence.xml	BACK-fund-ack-sec-order.xml
FIGT-split-fig.xml	BMST-reflist-precedes-ack.xml
FUND-funding-group.xml	CNTR.xml
FUND-meta-back-discrepancy.xml	FUND-funder-sources-number-mismatch.xml
FUND-multi-awards-per-funder.xml	FUND-Program-973-alone.xml
FUND-Program-973-of-NSFC.xml	FUND-R1-Guangdong.xml
FUND-R2-punctuation.xml	FUND-R1-renamed-funder.xml
FUND-R4-exception-DOE-subagencies.xml	FUND-R1.xml
FUND-top-agency-and-its-subagency.xml	FUND-R2-renamed-funder.xml
FUND-untagged-head.xml	FUND-R2.xml
FUND-untagged-tail-bilingual.xml	FUND-R3.xml
FUND-untagged-tail.xml	FUND-R4-a_TRAD.xml
FUND_NSFC-tagged-alone.xml	FUND-R4-b_EXPR.xml
META-article-categories-JOT.xml	FUND-R4-c_TRAD.xml
META-copyright-non-OSA-access-open-generic.xml	FUND-R4-d_EXPR.xml
META-expr-aff-deceased.xml	FUND-R4-e_TRAD.xml
META-intro-JOT.xml	FUND-R4-f_EXPR.xml
META-mixed-citation-in-abstract.xml	FUND-R4-g_TRAD.xml
META-ORCID-ID.xml	FUND-R4-i_EXPR.xml
META-permissions-absent.xml	GLOB-statement.xml
META-sig-block.xml	META-ORCID-content-type-invalid.xml
MTTX-license-p-author-open.xml	META-ORCID-ID-invalid.xml
MTNN-copyright-year-matches-epub-year.xml	OCIS-more-than-six.xml
MTTR-license-p-author-open.xml	OSAH-sqf_CHORUS-AM-missing.xml
MTTR-license-p-closed.xml	OSAH-sqf_CHORUS-VOR-missing-a.xml
MTTR-license-p-generic-open.xml	OSAH-sqf_CHORUS-VOR-missing-b.xml
OCIS-more-than-six-JOT.xml	OSAH-sqf_rel-art-missing_comment.xml
OCIS-single-JOT.xml	OSAH-sqf_rel-art-missing_ecorr-a.xml
OSAH-CHORUS-AM.xml	OSAH-sqf_rel-art-missing_ecorr-b.xml
OSAH-CHORUS-VOR.xml	OSAH-sqf_rel-art-missing_errat.xml
OSAH-ecorrection.xml	OSAH-sqf_rel-art-missing_pub-note.xml
OSAH-publishers-note.xml	OSAH-sqf_rel-art-missing_reply.xml
RFOX-article-ID-article-type.xml	SPMT-0_duplicate-DOI.xml
RFOX-book-and-chapter-JOCN.xml	SPMT-0_suppl-doc-back-meta-mismatch.xml

Fig. 5. Go and NoGo tests.

Fig. 6. Full-text Go tests.

op Go and NoGo tests for each statement. Go tests contain examples of valid markup or content; they should always pass, regardless of future changes. NoGo tests contain typical examples of invalid markup or content; regardless of future changes, they should always fail and generate expected error messages. To verify this, it is necessary to compare the validation report a NoGo test produces with the reference (“gold”) standard; i.e., the report that contains expected error messages.

Go test failure or No Go test success indicates that your modifications had an unexpected effect and that you need to review and fix your Schematron code.

Fig. 3 shows a Go test for the sup-mat.sch module. If we invoke only this module and run it over the test we will get no errors. Fig. 4 shows a NoGo test for the requirement *Duplicate sup. mat. DOIs are not allowed*. The test contains two identical DOIs. Running the Schematron that invokes the sup-mat.sch module results in the expected error message “ERROR [spmt-test:SPMT130]: DOI ‘10.1364/OPTICA.1.000001.s001’ is listed more than once”. Fig. 5 shows a small sample of Go and NoGo tests for a few Schematron modules.

In addition to Go tests for the individual assert and report statements, it is also useful to have full-text Go samples for the entire articles of various types, such as research article, correction, review or editorial; and at the various stages of production process; e.g., at the initial and final stages. These tests should pass when they are run against top-level Schematron instances. Fig. 6 shows a directory with full-text Go tests.

Since it is not unusual for a large-scale Schematron to have thousands of assert and report statements, and because multiple Go and NoGo tests may be needed to check each statement, ideally the test base should include many thousands of Go and NoGo tests, as well as a script that runs them every time a change is introduced.

Schematron change management: summary

When Schematron is used as QA/QC tool, the Schematron change management requires the following steps: (1) create Go and NoGo tests for the individual assert and report statements, (2) create Go tests for full-text articles of various types, (3) debug Schematron code when Go tests fail and NoGo tests pass, and (4) develop and test a script to run the entire test-base every time Schematron code is modified.

Schematron Maintenance Costs

To ensure that published content adheres to the publisher’s business and style rules we try to choose technologies and devise solutions that provide quality control consistency across (1) the entire enterprise, including external (vendors) and internal (in-house) production processes and (2) the entire production cycle of a manuscript, from XML conversion to initial validation to final check to post-publication corrections/enhancements.

One of the advantages of Schematron as a technology is that it can serve as “glue” that binds together various production workflow segments. However, as we saw earlier, Schematron maintenance does come with costs. The question then becomes, how can we reduce them?

Let us recall our introductory example: “The last segment of an image filename (i.e., of the value of the `xlink:href` attribute on the element `graphic`) must correspond to the sequence number of the parent figure. For example, for the fifth figure in the body, the numeric part of the last segment of the image filename; i.e., ‘-g005a’, must be 5. We will make an exception for correction article type, and we will only check this requirement at the final production stage.”

Clearly, these kinds of constraints cannot be implemented by means of grammar-based languages, such as DTD, XSD Schema or RelaxNG. Schematron seems a natural choice here.

Now let us consider another requirement: “Element `fig` must have the following content model.”

```
<!ELEMENT fig ((label, caption, (alternatives | array | code | graphic | media |
preformat)+) >
<!ATTLIST fig
    id ID #REQUIRED
    xml:base CDATA #IMPLIED
    position (anchor | background | float | margin) 'float'
    orientation (portrait | landscape) 'portrait'
    specific-use CDATA #IMPLIED
    xml:lang NMTOKEN #IMPLIED
    fig-type (color-online | color-print | figure) #IMPLIED >
```

Recall that, in JATS, element `fig` has the following content model:

```
<!ELEMENT fig ((object-id*, label?, (caption)?, (abstract)*, (kwd-group)*, (alt-
text | long-desc | email | ext-link | uri)*, (disp-formula | disp-formula-group |
chem-struct-wrap | disp-quote | speech | statement | verse-group | table-wrap | p |
def-list | list | alternatives | array | code | graphic | media | preformat)*,
(attr | permissions)*) >
<!ATTLIST fig
    id ID #IMPLIED
    xml:base CDATA #IMPLIED
    position (anchor | background | float | margin) 'float'
    orientation (portrait | landscape) 'portrait'
    specific-use CDATA #IMPLIED
    xml:lang NMTOKEN #IMPLIED
    fig-type CDATA #IMPLIED >
```

Here, the turquoise highlights indicate the elements that have been eliminated, and the yellow highlights indicate the items that have been changed. Let us now suppose that we implement this requirement in Schematron.

Schematron code

Elements

To ensure the correct sequence of child elements in the element `fig`, we will have to write a series of relatively complex assert/report statements involving the preceding-sibling:: and following-sibling:: XPath axes. As Eden and Cleghorn [1] notes, “although XPath is capable of expressing a sequencing constraint, attempting to use it in this way is inevitably verbose and problematic for maintenance. For example, consider a DTD element model (title, p+) – a Schematron rule with the context title might trivially and accurately (albeit without prohibiting other elements) test following-sibling::pin an assert; however, now consider (rb, (rt | (rp, rt, rp))+) and the rapid increase in complexity is plain.”

Attributes

Because in our modified content model we have the `id` attribute whose presence is now required, whereas in the DTD it is simply implied, we would have to write a Schematron rule to check for its presence. Here is an example from Eden and Cleghorn’s study [1] where “\$required.attrib is a variable read from a static external document and containing a list of elements with attributes whose presence are required by CUP but which are merely permitted by the DTD.”

```

<rule context="*[some $r in $required.attribs/descendant::cup:* satisfies ($r/local-name() eq ./local-name())]"
  id="str.global.attr">
  <let name="att.spec"
    value="$required.attribs/descendant::cup:*[local-name() eq current()/local-name()]" />
  <let name="att.miss"
    value="for $a in $att.spec/@*
      return $a[not(
        some $b in current()/@* satisfies local-name($b) eq local-name($a)
      )]" />
  <report role="fatal"
    id="str.global.attr.01"
    test="count($att.miss) gt 0">The element <name/> is missing the following required
      attributes: <value-of select="string-join(for $x in $att.miss return name($x),'',
        ')" />.</report>
</rule>

```

Schematron code complexity

In other words, as Eden and Cleghorn [1] indicate, this approach “leads to a need to treat the DTD as a coarse whitelist of elements which may be present, and Schematron as a finer blacklist of structures which are not desired.” Because different error messages may require different responses, we may have to use Schematron role structure to indicate the severity level of a problem: one can differentiate between such roles as “fatal,” “error,” “warning,” and “info.” Then it is left to a user or to a processing system to decide how to react to each of these.

It is obvious that the complexity in and of itself may result in the Schematron code being brittle and error-prone. Yet, as we have seen, simply writing Schematron code, no matter how sophisticated, is not sufficient to ensure quality; one also has to build and run a testbase to future-proof the Schematron when further development occurs. What does this entail?

NoGo tests for eliminated elements

We would need to make sure that the eliminated elements are not present in the element *fig*’s content. The eliminated elements are abstract, alt-text, attrib, chem-struct-wrap, def-list, disp-formula, disp-formula-group, disp-quote, email, ext-link, kwd-group, list, long-desc, object-id, p, permissions, speech, and statement. To do this, we would need to write one or more NoGo tests to make sure an error message is emitted if *fig* contains any of the eliminated elements.

NoGo tests to check element occurrences

Since the occurrence indicators for the elements label and caption have changed from “optional” to “required,” and the occurrence indicator for the class of elements (alternatives | array | code | graphic | media | preformat) have changed from “optional, repeatable” to “required, repeatable,” we would have to write NoGo tests to make sure the Schematron will throw an error when these elements are not present.

NoGo tests for required attributes

Since the value of attribute id has changed from IMPLIED to REQUIRED, we would need a NoGo test to ensure an error is emitted when the attribute is not present.

Accounting for the specified attribute values

The value model for attribute fig-type has changed from CDATA to (color-online | color-print | figure). Thus we would have to build a number of NoGo tests to ensure that specifying non-allowed values would result in an error. Last but not least, we would have to run all of the above-mentioned NoGo tests every time a Schematron is modified.

Subsetting

The main advantage

We could, however, implement the same requirement simply by overriding the default JATS parameterized content models


```

<!ENTITY % fig-model      "((%id.class;)*, label?, (%caption.class;)?,
                           (%abstract.class;)*, (%kwd-group.class;)*,
                           (%access.class; | %address-link.class;)*,
                           (%block-math.class; |
                           %chem-struct-wrap.class; |
                           %intable-para.class; |
                           %just-table.class; | %just-para.class; |
                           %list.class; | %simple-display.class;)*,
                           (%display-back-matter.class;)* )"      >

<!ENTITY % fig-atts
        "%jats-common-atts;
        %display-atts;
        fig-type CDATA                                     #IMPLIED" >

```

with the customized ones:

```

<!ENTITY % fig-model      "((%label.class;),
                           (%caption.class;),
                           (%simple-display.class;)+)"            >

<!ENTITY % fig-atts
        "%jats-common-atts-id-required;
        %display-atts;
        fig-type (%fig-types;)                                #IMPLIED" >

<!ENTITY % fig-types      "color-online | color-print | figure"  >

```

Note that, in this case, not only would it be unnecessary to write any Schematron code whatsoever but, perhaps even more significantly, we would not have to build and run any NoGo tests. Instead, we would be guaranteed that only allowed elements (in the correct quantities and in the correct order), and only allowed attributes (with the correct attribute values) would be present in the XML to be checked by Schematron. The DTD-validating parser will ensure adherence to these content model rules when the XML instance is checked against the DTD subset prior to Schematron validation.

Although simplifying *fig* involves the modification of only one content model, in real life many models may need to be simplified (the examples of *ref-list* and *article-meta* elements in Lapeyre [2] are especially's work compelling). Therefore, using Schematron to verify markup structures that could otherwise be checked by a validating parser against a DTD subset can result in substantial increases in software development costs.

Additional advantages

As Lapeyre [2] points out, subsetting offers many additional advantages.

Conversion to XML

For example, consider conversion from an author-submitted or legacy format to your production XML with the complete, out-of-the-box JATS and Schematron. Obviously, Schematron can be applied only after the conversion to the tag set takes place. Therefore, the software and people who perform the conversion will have a choice as to which elements and attributes to use. If they make a poor choice, the mistake will not be revealed until later when the Schematron is run. Creating a subset prevents the possibility of choosing the wrong elements and attributes in the first place, which is much more efficient than discovering a problem downstream the workflow, correcting it upstream, and then running the process all over again.

```

3 <article-types xmlns="http://www.osa.org/Schematron/util">
4
5     <article type="article-commentary" related-type="commentary-article"/>
6     <article type="correction"          related-type="corrected-article"/>
7     <article type="reply"               related-type="reply-article"/>
8     <article type="research-article"/>
9 </article-types>

```

Fig. 7. Article type lookup authority (fragment).

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <article article-type="correction">
3     <front>
4         <journal-meta>
5             <journal-id journal-id-type="publisher-id">JOSAA</journal-id>
6         </journal-meta>
7         <article-meta>
8
9             <!--<related-article id="RA1"
10                             related-article-type="corrected-article"
11                             journal-id="JOSAA" vol="28" issue="7"
12                             page="1372"/>-->
13
14         <abstract/>
15
16         </article-meta>
17     </front>
18 </article>

```

ERROR [osah-test:OSAH120]: related-article is missing for @article-type 'correction'

+ add related article info

Fig. 8. QuickFix in oXygen: validation.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <article article-type="correction">
3     <front>
4         <journal-meta>
5             <journal-id journal-id-type="publisher-id">JOSAA</journal-id>
6         </journal-meta>
7         <article-meta>
8
9             <!-- <related-article id="RA1"
10                             related-article-type="corrected-article"
11                             journal-id="JOSAA" vol="28" issue="7"
12                             page="1372"/> -->
13
14             <related-article id="RA1" issue="" journal-id="JOSAA" page=""
15                             related-article-type="corrected-article" vol="" />
16         <abstract/>
17     </article-meta>
18 </front>
19 </article>
20

```

Fig. 9. QuickFix in oXygen: action.

Human editing and XML editing tools customization

Even more obvious is the impact of not doing subsetting on XML-editing software and the people who use it. XML visual editing tools typically display context-sensitive lists of elements and attributes allowed in a given context. Since such tools are typically used after the document is converted to XML but before it is Schematron-checked, using Schematron to simplify some content models, eliminate certain elements and attributes, or give specific values to other attributes will be of absolutely no help to human editors because they will not see these alterations reflected in their tool. For human editors, these modifications need to be made via subsetting. Otherwise, as with XML conversion, the tagging mistakes will not be revealed until Schematron is run downstream in the workflow, when correcting them will take more time and cost more.

Subset as a communication device

XML tools—such as XSLT stylesheets for HTML display, XML-based composition systems, filters into and out of page make-up systems, applications that interface with relational databases and other types of XML repositories—typically need customization. How to convey to tool developers which markup structures, elements, attributes, attribute values, and occurrence indicators are allowed and which are not, so that they can perform the customization? You may have carefully written your Schematron to make sure only the allowed structures and values are present in your XML, but even if you documented the requirements within the Schematron code, it is not a trivial task to communicate those requirements to tool developers. If, on the other hand, you created a subset then the three override modules (class, mix, and model overrides) is all that the tool developer needs to perform customization. What this reveals is that a DTD subset can serve as a helpful and efficient method of communicating or teaching a tag set to anyone who needs to learn it, such as in-house staff, conversion and composition vendors, business partners, and aggregators.

The costs of subsetting

Of course, there are some costs to creating a subset. The main component here is analysis: the data architect must have a very good understanding of both the document structure and the direction in which the organization's publishing program is going. Only then will she or he be able to determine which constraints should be imposed via subsetting and which are better to be checked via Schematron.

Another, lesser, expense is the cost of actually making and testing your subset using the JATS-recommended methods and mechanisms as described in the journal publishing tag library [3].

One has to also be mindful of the fact that the subset and each of its subsequent modifications will need to be shared with conversion vendors. This, however, is usually not a problem for a vendor; in fact, whether or not you, as a publisher, are aware of it, many vendors, to make their life easier, create a “production DTD” on their own, making the best guess of what your document model is. Wouldn't it be more effective and accurate if, instead, you were the one who created the subset in the first place?

Schematron QuickFixes

One of the factors that may help you decide how to achieve the right balance between creating a subset and building Schematron is availability of QuickFixes technology. It emerged only a few years ago and overturned the old adage that Schematron could not “do” anything, in the sense that Schematron only checks the XML document and emits error messages, and then people or software have to fix the detected problems. QuickFixes operates within Schematron and allows the QC person to add, delete, or replace various nodes, as well as replace textual strings with nodes or other strings.

Here is a real-life example of adding a missing *related-article* element based on the *article-type* attribute.

We need to implement the following requirement: “Articles of certain types, determined by the article-type lookup authority, must have related-article metadata.” Fig. 7 shows a fragment of article-type lookup authority.

Fig. 8 shows an article of type “correction,” which is missing a related-article element and its required attributes. While running the Schematron validation the user is presented with the error message and a suggested QuickFix.

Clicking the QuickFix popup results in the insertion of the missing element with some attribute values computed and others to be filled manually (Fig. 9).

Fig. 10 shows the relevant QuickFixes code embedded into a Schematron rule. Thus if the quality of your XML document may benefit from user-applied predefined automatic repairs then Schematron with QuickFixes is definitely the way to go.

Schematron vs. Subset: Validation Scenarios

Implementing business and style rules calls for imposing constraints on JATS. Since constraints can be implemented via developing Schematron and via creating a subset, how can we decide on the best balance between the two?

Clearly, there are conditions that no Schema-based language can check in principle. As Schematron inventor Rick Jelliffe put it, Schematron is “a feather duster to reach the parts other schema languages cannot reach.” As Schwarzman et al. [4]

```

132 <rule id="OSAH120" context="abstract">
133   <let name="ruleID" value="'OSAH120'"/>
134   <report role="info" test="$diagnostic"><value-of select="m:info($ruleID)"/>:
135     Checking rule <value-of select="$ruleID"/></report>
136
137   <d:req>Articles of certain types, determined by the article type lookup authority,
138     must have related-article metadata.</d:req>
139
140   <d:func>An article type specification is determined by lookup of an article whose
141     article/@type value is the same as that of this article/@article-type.</d:func>
142
143   <let name="articletype-spec"
144     value="$articletypes-lookup/*/m:article[exists(@related-type)]
145       [@type/tokenize(., '\s+') = $article-type]"/>
146
147   <assert sqf:fix="add-related-article"
148     test="empty($articletype-spec/@related-type) or exists(preceding-sibling::related-article)">
149     <value-of select="m:error($ruleID)"/>: related-article/@related-article-type
150     '<value-of select="$articletype-spec/@related-type"/>' is missing for
151     @article-type '<value-of select="$article-type"/>'</assert>
152
153   <sqf:fix id="add-related-article">
154     <sqf:description>
155       <sqf:title>add related article info</sqf:title>
156     </sqf:description>
157     <sqf:add position="before">
158       <related-article xmlns="" id="RA1" vol="" issue="" page="">
159         <xsl:attribute name="journal-id">
160           <xsl:value-of select="$journal-code"/>
161         </xsl:attribute>
162         <xsl:attribute name="related-article-type">
163           <xsl:value-of select="$articletype-spec/@related-type"/>
164         </xsl:attribute>
165       </related-article>
166     <xsl:text>
167   </xsl:text>
168     </sqf:add>
169   </sqf:fix>
170 </rule>

```

Fig. 10. QuickFix markup.

stated, Schematron can (1) validate or report on document structure, such as the presence or absence of elements (“Does my <abstract> contain a <disp-formula>?”). It can also look for the location of elements (“In which section is the table in my article?”); (2) validate or report on document content; i.e., there must be some content, or there must be some particular content and the content must follow some rule (“There must be a displayed equation in <abstract> and the label for this equation must always occur just after the math in the <disp-formula> element, not before it; tell me when this rule has been disobeyed!”); (3) validate or report on the presence or absence of attributes, or on the content of attributes (“The attribute ‘specific-use’ must occur on <aff> within <contrib> and this attribute’s value must be ‘internal-use’”); and (4) check co-occurrence constraints: if X is true, then Y should be true; or, A, B, C, and W must all be present (somewhere).

Therefore, when a condition cannot be checked via a DTD, it is obvious that it should be checked using another technol-

ogy, usually Schematron.

There are, however, many types of constraints that can be checked either via Schematron or via a subset, and which path to choose may depend on the nature of the workflows, tools, and markup structures, as well as on the available staff expertise and the resources available for software development.

Below we consider several scenarios where validation can be performed either via developing Schematron or via making a subset.

Validating content at various stages of production process

As an XML document goes through various stages of the production process, it may need different validation rules applied at each stage. For example, at the initial stage, a journal article may not have such metadata as an issue number, pagination, or a DOI. At the final stage, these metadata elements must be present and correct, but since copyediting has already been done, reference-checking rules might need to be relaxed. After the document leaves the vendor, some metadata may need

to be applied in-house, which would necessitate additional validation. When the published article is retrieved from the XML repository, e.g., for metadata correction or semantic enrichment, the XML may need to go through another validation before the article is loaded back into the repository. While it is possible to build a different subset for each production stage, and some publishers have done just that, it would probably be more practical to build a top-level Schematron instance for each production stage, with all top-level Schematron instances sharing a common modular library.

Validating different content segments

If your portfolio contains several types of journals with slightly different business and style rules, then similarly, it would probably be more practical to build top-level Schematron instances for each journal type rather than create a different subset for each.

Retrospective conversion

Suppose you would like to convert a few decades' worth of journal content to JATS-conforming XML. Since the business and style rules have changed through the years, unless you perform an extensive and exceedingly expensive document analysis, you cannot predict with 100% accuracy which structures you may encounter in the legacy content. Thus if you choose to go the subset route you may find yourself in a situation where some unexpected content variations in the back issues cannot be tagged. In this case, using full JATS with Schematron that could be changed in an agile manner as described in Dineen et al. [5], might be the best option.

Validating different genres

For a publisher that uses JATS to publish journal articles and

BITS to publish edited books (which contain chapters that resemble journal articles), a modular Schematron offers an opportunity to check both genres, since BITS includes JATS modules. Fig. 11 shows a reduced example of a large production Schematron that contains five top-level Schematrons for a book (BK), a book article (BA), and a journal article (JA) at the initial (I) and final (F) validation stages.

Since book article and journal article share the content models for body and back matter, the same modular library can be used to validate both. Because metadata models for these two document types are different, dedicated modules are employed to validate those.

Here again, the Schematron-based solution is probably preferable to making genre-specific subsets.

Quasi-static and quasi-dynamic changes

As observed earlier, the data architect must have a very good understanding of both the document structure and the direction of the organization's publishing program. If we know that certain markup structures, elements, and attributes are not likely to be used in the near future, or ever, then it may make sense to cut down the tag set by tightening content models, thereby simplifying human editing, tool customization, and downstream processing, to mention just a few benefits of subsetting.

A potentially useful conceptual framework is to think of the subset as being quasi-static, and of the Schematron as being quasi-dynamic: that is, while the subset (usually in the form of a DTD) should be revised infrequently, e.g., once a year; the Schematron code could be revised as needed, e.g., monthly on average.

This approach may help with deciding on the right balance between the two techniques. In an extreme case, in which your document structure and editorial style are very stable and you do not anticipate a lot of business and style rule changes in the near future, then your needs may well be met by a subset.

Conclusion

Implementing efficient XML quality control across the entire enterprise and covering the entire production cycle requires an imposition of constraints on JATS in order to comply with business and style rules. This can be achieved via a combination of creating a subset and building Schematron.

Naturally, constraints that cannot be enforced via subsetting should be implemented via Schematron. As for the constraints that could be implemented via either mechanism, in deciding which technique to use, one may want to consider (1) whether content needs to be validated at different points of the workflow; (2) whether content comprises various segments, includ-

	BK	BAI	BAF	JAI	JAF
book-meta.sch [BKMT]	✓				
bookarticle-meta-final.sch [BAMF]	✓		✓		
bookarticle-meta.sch [BAMT]	✓	✓	✓		
...					
figures.sch [FIGT]	✓	✓	✓	✓	✓
mathml.sch [MATH]	✓	✓	✓	✓	✓
...					
journal-meta-non-init.sch [MTNN]					✓
journal-meta.sch [META]				✓	✓
...					
ref-meta.sch [RFMT]	✓	✓	✓	✓	✓
tables.sch [TBLE]	✓	✓	✓	✓	✓

Fig. 11. Schematron: validating different genres.

ing converted legacy material; (3) whether content comprises different genres; (4) which markup structures are likely to change frequently (quasi-dynamic), which are likely to be stable (quasi-static), and which are likely not to be used at all; (5) whether XML quality is expected to benefit from applying QuickFixes; and (6) what budget, human resources, and expertise are available for developing and maintaining Schematron, Go and NoGo tests, and scripts for running them.

In practice, most publishers need to check the XML at multiple points in the production process, have different content segments, publish documents of different genres, perform retrospective conversion, and want to automate manual QC processes by having vendors or staff use QuickFixes. By using Schematron where appropriate and a subset where practical the publisher may realize savings on software development and maintenance while ensuring XML quality.

Both technologies have costs associated with them: (1) building Schematron requires knowledge of that technology, as well as developing an extensive base of Go and NoGo tests along with the testing protocol, and running the testbase prior to each Schematron release; and (2) making a subset requires knowledge of the publishing program's direction, a thorough understanding of the current and future production processes, rigorous document analysis, and the expertise in creating a subset in accordance with best practices.

In 2015, we moved from using Schematron with “out-of-the-box” JATS to using Schematron with a JATS subset. As a result, we reduced the number of requirements checked by Schematron at that time from 568 to 494, and the number of assert/report statements in it from 845 to 737. Not only did we significantly simplify the Schematron code but, perhaps more importantly, we also decreased the number of required Go and NoGo tests, making Schematron change management easier.

While there is no “one-size-fits-all” approach, knowing your publication program, production process, and tools, as well as being cognizant of the budget and human resources available for software development and maintenance will help you achieve the right balance between using a subset and Schematron.

Conflict of Interest

No potential conflict of interest relevant to this article was reported.

Acknowledgments

The author would like to thank anonymous reviewers, Pavel Exarkhopoulo, and Mary Seligy for their valuable critique and suggestions.

Sources

- Krawitz F, McAndrews J, O’Keeffe R. How well do you know your data? Converting an archive of proprietary markup schemes to JATS: a case study [Internet]. Bethesda, MD: National Center for Biotechnology Information; 2012 [cited 2018 Jan 20]. Available from: <https://www.ncbi.nlm.nih.gov/books/NBK100349/>
- O’Brien D, Fisher J. Journals and magazines and books, oh my! A look at ACS’ use of NLM tagsets [Internet]. Bethesda, MD: National Center for Biotechnology Information; 2010 [cited 2018 Jan 20]. Available from: <https://www.ncbi.nlm.nih.gov/books/NBK47083/>
- Schwarzman AB. Superset me—not: why the journal publishing tag set is sufficient if you use appropriate layer validation [Internet]. Bethesda, MD: National Center for Biotechnology Information; 2010 [cited 2018 Jan 20]. Available from: <https://www.ncbi.nlm.nih.gov/books/NBK47084/>
- Usdin T, Lapeyre DA, Glass CM. Superimposing business rules on JATS [Internet]. Bethesda, MD: National Center for Biotechnology Information; 2015 [cited 2018 Jan 20]. Available from: <https://www.ncbi.nlm.nih.gov/books/NBK279902/>

References

- Eden M, Cleghorn T. An implementation of BITS: the Cambridge University Press experience [Internet]. Bethesda, MD: National Center for Biotechnology Information; 2016 [cited 2018 Jan 20]. Available from: <https://www.ncbi.nlm.nih.gov/books/NBK350535/>
- Lapeyre DA. Why create a subset of a Public Tag Set [Internet]. Bethesda, MD: National Center for Biotechnology Information; 2010 [cited 2018 Jan 20]. Available from: <https://www.ncbi.nlm.nih.gov/books/NBK47099/>
- National Center for Biotechnology Information. How to make new tag sets [Internet]. Bethesda, MD: National Center for Biotechnology Information; 2015 [cited 2018 Jan 20]. Available from: <https://jats.nlm.nih.gov/publishing/tag-library/1.1/chapter/implement-howtodtds.html>
- Schwarzman AB, Seligy M. Schematron: a handy XML tool that’s not just for villains! [Internet]. JATS4R; 2016 [cited 2018 Jan 20]. Available from: <http://jats4r.org/schematron-a-handy-xml-tool-thats-not-just-for-villains>
- Dineen MS, Gross M, Ashlem D, Friedman B, Schwarzman A, Kupferstein G. NLM conversion to build “atomic” physics content in an agile fashion [Internet]. Bethesda, MD: National Center for Biotechnology Information; 2013 [cited 2018 Jan 20]. Available from: <https://www.ncbi.nlm.nih.gov/books/NBK159728/>