XML Prague 2017

Conference Proceedings

University of Economics, Prague
Prague, Czech Republic

February 9–11, 2017

XML Prague 2017 – Conference Proceedings

ISBN 978-80-906259-2-1 (pdf)
ISBN 978-80-906259-3-8 (ePub)

<oXygen/>®

The Complete Solution for XML Authoring & Development

XML Editor

oXygen XML Editor is a complete XML editing solution for developers and content authors.

XML Author

oXygen XML Author provides a visual interface designed for user-friendly structured authoring.

XML Developer

oXygen XML Developer is an effective and easy-to-use industry-leading XML development tool.

XML Web Author

oXygen XML Web Author is the ultimate tool for editing and reviewing content in browsers on any device.

WebHelp

oXygen XML WebHelp allows you to publish DITA and DocBook in a modern, interactive web-based help system.

www.oxygenxml.com

Table of Contents

General Information

Date

February 9th, 10th and 11th, 2017

Location

University of Economics, Prague (UEP)
nám. W. Churchilla 4, 130 67 Prague 3, Czech Republic

Organizing Committee

Petr Cimprich, *XML Prague, z.s.*
Vít Janota, *Xyleme & XML Prague, z.s.*
Káťa Kabrhelová, *XML Prague, z.s.*
Jirka Kosek, *xmlguru.cz & XML Prague, z.s. & University of Economics, Prague*
Martin Svárovský, *Memsource & XML Prague, z.s.*
Mohamed Zergaoui, *ShareXML.com & Innovimax*

Program Committee

Robin Berjon, *science.ai*
Petr Cimprich, *Xyleme*
Jim Fuller, *MarkLogic*
Michael Kay, *Saxonica*
Jirka Kosek (chair), *University of Economics, Prague*
Ari Nordström, *SGMLGuru.org*
Uche Ogbuji, *Zepheira LLC*
Adam Retter, *Evolved Binary*
Andrew Sales, *Andrew Sales Digital Publishing*
Felix Sasaki, *DFKI / W3C Fellow*
John Snelson, *MarkLogic*
Jeni Tennison, *Open Data Institute*
Eric van der Vlist, *Dyomedea*
Priscilla Walmsley, *Datypic*
Norman Walsh, *MarkLogic*
Mohamed Zergaoui, *Innovimax*

Produced By

XML Prague, z.s. (http://xmlprague.cz/about)
Faculty of Informatics and Statistics, UEP (http://fis.vse.cz)

Sponsors

oXygen (http://www.oxygenxml.com)
le-tex publishing services (http://www.le-tex.de/en/)
Antenna House (http://www.antennahouse.com/)
Saxonica (http://www.saxonica.com/)
speedata (http://www.speedata.de/)
Xeditor (http://www.xeditor.com/)

Preface

This publication contains papers presented during the XML Prague 2017 conference.

In its twelfth year, XML Prague is a conference on XML for developers, markup geeks, information managers, and students. XML Prague focuses on markup and semantic on the Web, publishing and digital books, XML technologies for Big Data and recent advances in XML technologies. The conference provides an overview of successful technologies, with a focus on real world application versus theoretical exposition.

The conference takes place 9–11 February 2017 at the campus of University of Economics in Prague. XML Prague 2017 is jointly organized by the non-profit organization XML Prague, z.s. and by the Faculty of Informatics and Statistics, University of Economics in Prague.

The full program of the conference is broadcasted over the Internet (see http://xmlprague.cz)—allowing XML fans, from around the world, to participate on-line.

The Thursday runs in an unconference style which provides space for various XML community meetings in parallel tracks. Friday and Saturday are devoted to classical single-track format and papers from these days are published in the proceeedings. Additionally, we coordinate, support and provide space for W3C XSLT and XProc working group meetings collocated with XML Prague.

We hope that you enjoy XML Prague 2017.

— Petr Cimprich & Jirka Kosek & Mohamed Zergaoui
XML Prague Organizing Committee

XPath 3.1 in the Browser

John Lumley
jwL Research, Saxonica
<john@jwlresearch.com>

Debbie Lockett
Saxonica
<debbie@saxonica.com>

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

This paper discusses the implementation of an XPath3.1 processor with high levels of standards compliance that runs entirely within current modern browsers. The runtime engine Saxon-JS, written in JavaScript and developed by Saxonica, used to run pre-compiled XSLT3.0 stylesheets, is extended with a dynamic XPath parser and converter to the Saxon-JS compilation format. This is used to support both XSLT's `xsl:evaluate` instruction and a JavaScript API `XPath.evaluate()` which supports XPath outside an XSLT context.

1. Introduction

XSLT1.0 was originally developed primarily as a client-side processing technology, to increase data-adaptability and presentational flexibility with a declarative model supported in the browser. By 2006 all the major desktop browsers supported it, but the rise in importance of mobile client platforms and their code footprint pressures then sounded the death-knell. However, remnants of support for the XPath 1.0 component of XSLT can be found in most, but not all, of the current desktop browsers.

In the meantime, spurred mostly by unexpected takeup of XSLT and XQuery in server-side XML-based architectures, the technologies have progressed through the "2.0" stage (temporary variables, type model and declarations, grouping, canonical treatment of sequences, extensive function suites...) to the current candidate recommendation standards for 3.0/3.1 in XSLT, XQuery and XPath. At this stage support for maps, arrays and higher-order functions join let constructs, increased standard function libraries and others to provide a core set of common functionality based on XPath. XQuery uses this with its own (superset

of XPath) syntax to support query-based operation and reporting. XSLT adds a template-based push model, within an XML syntax, which is designed to be able to support streaming processing in suitable use cases.

These developments are sufficiently robust and powerful that, other circumstances permitting, exploiting XPath 3.1 to process over XML data is highly attractive. But at present this can only be performed in server-side situations – no browser developers could contemplate either the development effort or the necessary memory footprints needed for the "2.0" level compilers for what they consider niche (i.e. non-mobile) applications, let alone that required for 3.0+.

What *has* been developed extensively in browsers are *JavaScript* processors, both internal compilers, JIT runtimes and development libraries, such that very significant programmes can be executed in-browser[1]. And in general the level of conformance of and interoperability between implementations in different browsers is reasonable.

Exploiting this JavaScript option for supporting XSLT/XPath /XQuery has been explored in a number of cases:

- Saxon-CE[1] cross-compiled a stripped-down XSLT2.0 Saxon compiler from Java into JavaScript using Google's GWT technology. This worked, but the loaded code was large, very difficult to test and debug and very exposed to GWT's cross-browser capabilities.
- There are a small number of developers working on open-source implementations in native JavaScript for XPath 2.0 (e.g.Ilinsky[2]) and XQuery (*XQIB*[2]), though it is very unclear their level of standards conformance. Other JavaScript-based implementations for XPath 1.0 include *Wicked good XPath*[3], supporting the DOM Level 3 subset and *XPath-js*[4] which supports the full XPath 1.0 standard.
- During 2016 Saxonica developed *Saxon-JS* [3], a runtime environment for XSLT 3.0, implemented in JavaScript. This engine interprets compiled execution plans, presented as instruction trees describing the execution elements of an XSLT3.0 stylesheet. The execution plan is generated by an independent compiler, which performs all necessary parsing, optimisation, checking and code generation. Consequently the runtime footprint is modest (~ 200kB in minimised form) and programme execution incurs no compilation overhead.

1.1. Saxon-JS Runtime – dynamic evaluation

The Saxon-JS runtime environment is written entirely in JavaScript and is intended to interpret and evaluate XSLT program execution plans and write results

[1] In the process pushing client-based Java towards oblivion too.
[2] http://www.xqib.org/index.php
[3] https://github.com/google/wicked-good-xpath
[4] https://github.com/andrejpavlovic/xpathjs

into the containing web page. These execution plans are presented as instruction trees describing the execution elements of an XSLT3.0 stylesheet in terms of sets of templates and their associated match patterns, global variables and functions, sequence constructors consisting of mixtures of XSLT instructions and XPath expression resolution programs and other static elements such as output format declarations.

XSLT programs for Saxon-JS are compiled specifically for this target environment, using *Saxon-EE* to generate an XML tree as a *Stylesheet Export File* (SEF)[5]. This tree is loaded by a `SaxonJS.transform()` within a web page, which then executes the given stylesheet, with a possible XML document as source and other aspects of dynamic context, writing results to various sections of the web page via `xsl:result-document` directives. The engine supports specialist XSLT modes (e.g. `ixsl:on-click`) to trigger interactive responses, based on a model originally developed for *Saxon-CE*.

Saxon-JS supports a very large proportion of XSLT3.0 and XPath3.1 functionality. As all the static processing (parsing, static type checking, adding runtime checks.…) of the presented XSLT stylesheet is performed by Saxon-EE during its compilation operation, programs running in Saxon-JS should benefit from the very high levels of standards compliance that the standard Saxon product achieves. At runtime Saxon-JS only has to check for dynamic errors and often these are programmed in the execution plan as specific check instructions. The consequences are that

- The Saxon-JS code footprint is very much smaller, consisting only of a runtime interpreter.
- Execution starts immediately after loading the instruction tree – there is no compilation phase
- *But* Saxon-JS assumes *the code is correct*, as all static errors have been removed and necessary dynamic checks have been added.
- As there is no runtime compiler, or XPath parser, there is no implicit support for dynamic evaluation of XPath expressions (through the `xsl:evaluate` instruction), nor indeed runtime definition and evaluation of functions.

Within some related work on streamability analysis, one of the authors had been working on parsing XPath expressions using XSLT code to generate reduced parse trees and manipulating and analysing their properties. One of the possibilities was to recast this work to run entirely in Saxon-JS. But it also opened the possibility of extending Saxon-JS to both parse the XPath and generate equivalent execution plans (i.e. *Stylesheet Export File*) and hence support dynamic evaluation.

[5]Apart from some additional Saxon-JS attributes, the tree is identical to that used for linking separately-compiled packages for execution by a Java-based server-side Saxon engine – hence it shares the same degree of compilation "correctness" as any other Saxon-processed stylesheet.

This paper describes the design, construction and testing of such an extension.

2. Overall design

The route from XPath expression to evaluated result involves threee major steps. Firstly the expression string must be parsed against an XPath grammar to check that i) it is correct and ii) what XPath instructions are present. This means that matched grammar productions and variable tokens must be identified and grouped. This is most smoothly reported as a tree. Secondly this representation of the XPath expression must be converted into a set of suitable instructions for the the Saxon-JS, which will be cast as an XML tree. Finally the instruction tree needs to be interpreted by the Saxon-JS runtime, to produce the given result.

The main facility is written as an additional JavaScript object `XPathJS` added to the `SaxonJS` runtime[6]. There are three significant phases as shown in the figure

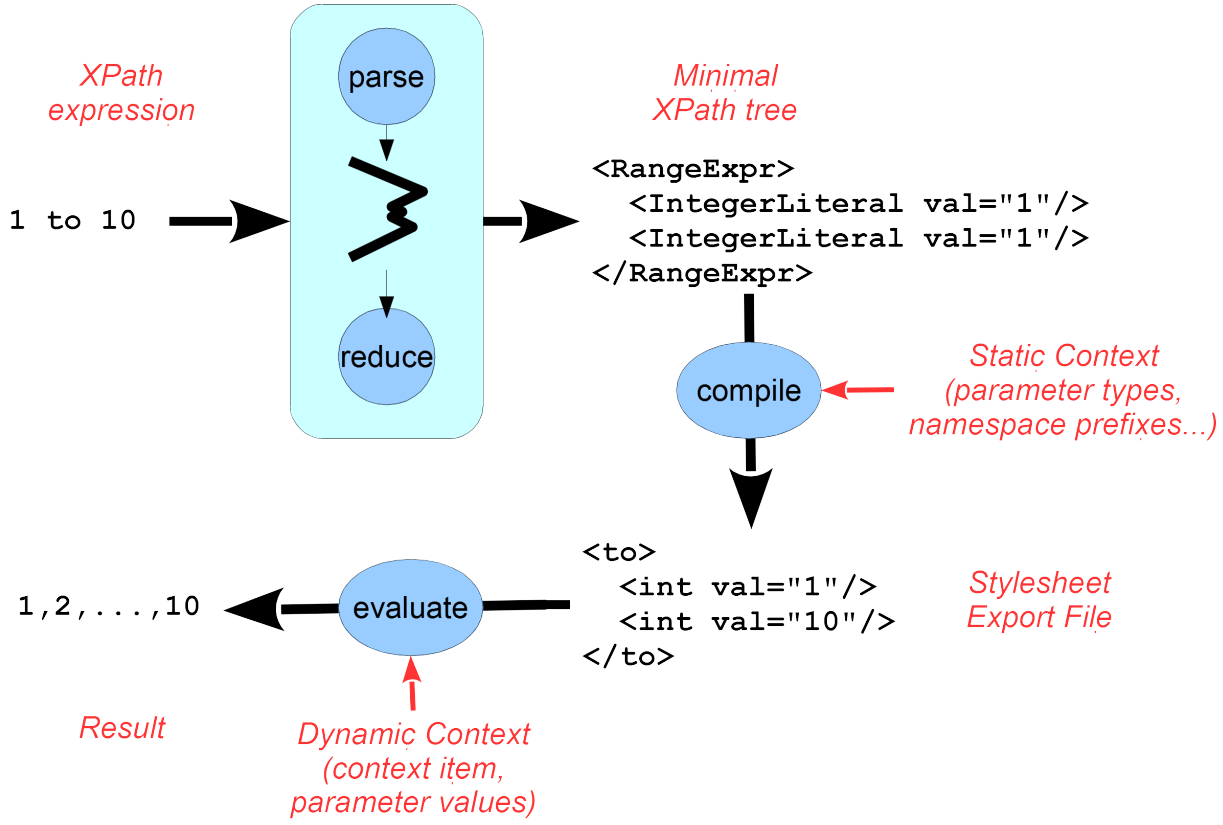


Figure 1. Processing phases

- `parse(xpath)` produces a (reduced) parse tree of the XPath expression, or an error if appropriate. This has similarities with Pemberton's *Invisible XML*,

[6]It has been arranged that the code for these features (which has a memory footprint as large as Saxon-JS itself) is loaded dynamically when first needed. Thus no additional overhead is incurred if a stylesheet does not use these features.

retaining only productions of interest, and decorating them with suitable properties (such as the `op` of a `MultiplicativeExpr`). It also converts sections defined gramatically as repeats such as `X op X (op X)*` (e.g. `let ….`) into nested trees of constant arity such that later phases only deal with strictly canonical forms.

- `compile(parseTree,staticContext)` generates a suitable SEF tree[7]. The parse tree is recursively converted to elements and attributes of the resulting SEF and static type checking is performed. More details below.
- `evaluate(SEF,contextItem,params)` interprets the execution plan with an optional context item, and given parameter value bindings. This uses the Saxon-JS evaluation engine. The result is then converted or wrapped to an appropriate final type, such as an iterator around XDM types for use in `xsl:evalute` or plain or arrayed Javascript types for a Javascript invocation.

2.1. Parsing the expression

XPath expressions are parsed by a two-step process: firstly the expression is checked for correctness and a full parse tree generated; secondly this parse tree is reduced to the essential details and converted to a canonical form.

A parser built from the XPath 3.1 EBNF grammar by Gunther Rademacher's REx parser-generator [5] is used. This is coded in JavaScript, using callback functions for starting and ending non-terminal productions, detected whitespace and terminal character phrases, which are indexed into the original input string. These callbacks are currently used to generate a DOM tree, which can be *very* large. For example the simple XPath `1 to 10` generates the following tree (which is up to 27 levels deep), some of whose closing tags have been elided:

```
<XPath>
 <Expr>
  <ExprSingle>
   <OrExpr>
    <AndExpr>
     <ComparisonExpr>
      <StringConcatExpr>
       <RangeExpr>
        <AdditiveExpr>
         <MultiplicativeExpr>
          <UnionExpr>
           <IntersectExceptExpr>
            <InstanceofExpr>
             <TreatExpr>
              <CastableExpr>
```

[7] In this case the tree is retained in memory and *not* serialized and written as an external file.

```
                <CastExpr>
                 <ArrowExpr>
                  <UnaryExpr>
                   <ValueExpr>
                    <SimpleMapExpr>
                     <PathExpr>
                      <RelativePathExpr>
                       <StepExpr>
                        <PostfixExpr>
                         <PrimaryExpr>
                          <Literal>
                           <NumericLiteral>
                            <IntegerLiteral>1</IntegerLiteral>
                           </NumericLiteral>
                          …
        </AdditiveExpr>
        <TOKEN>to</TOKEN>
        <AdditiveExpr>
         <the same…>
                           <NumericLiteral>
                            <IntegerLiteral>10</IntegerLiteral>
                           </NumericLiteral>
                          …
        </AdditiveExpr>
       </RangeExpr>
      </StringConcatExpr>
     </ComparisonExpr>
    </AndExpr>
   </OrExpr>
  </ExprSingle>
 </Expr>
 <EOF/>
</XPath>
```

As Pemberton[4] has pointed out, such trees, while correct, aren't very efficient, so the second phase reduces them to only the minimal essential elements. This is written as a recursive function `reduce()` which generally switches on the node name, with the following strategies:

- Literals have their value written as an attribute on the production.
- By default an element that has only one element child is reduced to the reduction of that child, e.g.

```
<AdditiveExpr>
  … <NumericLiteral>
      <IntegerLiteral>1</IntegerLiteral>…
→ <IntegerLiteral value="1"/>
```

- Tokens that convey variation meaning are added as suitable attributes to the main element. Non-essential tokens (which are constant for the given production) are deleted, e.g.

```
1 * 4 to ceiling(10.1)→
… <RangeExpr>
        …<MultiplicativeExpr>
          …  <IntegerLiteral>1</IntegerLiteral> …
          <TOKEN>*</TOKEN>
          …  <IntegerLiteral>4</IntegerLiteral> …
         </MultiplicativeExpr>...
        <TOKEN>to</TOKEN>
        … <FunctionCall>
             <FunctionEQName>
              <FunctionName>
               <QName>ceiling</QName>
              </FunctionName>
             </FunctionEQName>
             <ArgumentList>
              <TOKEN>(</TOKEN>
              <Argument>
               … <DecimalLiteral>10.1</DecimalLiteral> …
              </Argument>
              <TOKEN>)</TOKEN>
             </ArgumentList>
            </FunctionCall> …
       </RangeExpr>
…
→ <RangeExpr>
    <MultiplicativeExpr op="*">
     <IntegerLiteral value="1"/>
     <IntegerLiteral value="4"/>
    </MultiplicativeExpr>
    <FunctionCall name="ceiling">
     <DecimalLiteral value="10.1"/>
    </FunctionCall>
   </RangeExpr>
```

- Constructs that can have indefinite repetitions of subsections, such as `let $a:=1, $b:=2 return $a+$b` are regularised into nested trees (that can be either left or right associative) with constant arity, so that subsequent phases are presented with a canonical form, e.g.

```
<LetExpr>
 <SimpleLetBinding var="a">
  <IntegerLiteral value="1"/>
 </SimpleLetBinding>
```

```
 <LetExpr>
  <SimpleLetBinding var="b">
   <IntegerLiteral value="2"/>
  </SimpleLetBinding>
  <AdditiveExpr op="+">
   <VarRef name="a"/>
   <VarRef name="b"/>
  </AdditiveExpr>
 </LetExpr>
</LetExpr>
```

A few other sections of specialist code conversion are carried out at this stage, including replacing path shortcuts (`//`, `..` and `@`*`name`*) with equivalent full forms. With this reduced tree, code generation can then start[8].

2.2. Generating the execution plan

Many of the SEF instructions are in close correspondence with the XPath grammar productions and their "arguments" correspond to the results of evaluating their child subtrees, so the (unoptimised) code generation problem is basically to map the parse tree into a generally similar SEF tree, checking for static errors and adding runtime instructions to check for dynamic errors. This code generator is written entirely in JavaScript[9].

The bulk of the recursive compiling converter (`prepare(`*`node`*`,`*`context`*`)`) is a switch based on the XPath expression production types, e.g. `RangeExpr` for the *`x`* `to` *`y`* range generator. The `context` argument contains the static context, such as namespace prefix mappings, decimal formats and so forth, as well as semi-static information, such as the inferred type of the context item and the names, allocated storage slots and (declared or inferred?) types of the external parameters and local variables that are in-scope for the expression node being processed. (As usual name scoping follows the `following-sibling::*/ descendant-or-self::*` compound axis.)

For many of the productions the conversion is generally to produce an equivalent SEF instruction element (in this case `to`) with its two children (which could of course be anything from an `IntegerLiteral` to a full-blown computation tree) being processed recursively to produce their value-generating instruction trees.

For example the XPath `1 to 2 * 10` has a reduced parse tree of:

[8]It should be possible to perform some of this reduction during the original creation of the parse tree using smarter and language-sensitive callback functions. The authors haven't yet had an opportunity to explore this.

[9]It could have been written entirely XSLT and cross-compiled to produce an additional SEF tree that was used as a programme to generate another SEF tree for the XPath expression. The first author feels he learned more doing it "the hard way" and a rewrite in XSLT to support a more portable position is attractive, but it certainly won't be as fast as the native JavaScript version.

```
<RangeExpr>
  <IntegerLiteral value="1"/>
  <MultiplicativeExpr op="*">
    <IntegerLiteral value="2"/>
    <IntegerLiteral value="10"/>
  </MultiplicativeExpr>
</RangeExpr>
```

which is converted to an instruction tree:

```
<to type="xs:integer*">
  <int val="1" type="xs:integer"/>
  <arith op="*" calc="i*i" type="xs:integer">
    <int val="2" type="xs:integer"/>
    <int val="10" type="xs:integer"/>
  </arith>
</to>
```

where the type of arithmetic to be performed on the two inner values (integer times integer) is encoded in the `@calc` attribute – the runtime calculator is directed by this. (The `@type` annotations help show the determined type of the result during the compilation process – see below. They are not interpreted by Saxon-JS.)

In some cases, such as `LetExpr` and `ForExpr`, the context has to be altered to add a suitable variable binding and slot allocation to hold its value. In path expressions such as `RelativePathExpr` and `AxisStep`, the mapping is rather more complex and also involves the generation of specific JavaScript code, attached to the instruction element, to recognise candidate nodes in execution of the step.

The `FunctionCall` production can be used for many cases, apart from calls to core functions. Casting constructors for the `xs:atomic` types (e.g. `xs:double('NaN')`) are detected and converted to suitable `cast` instructions. Some function calls, such as `true()` are converted into direct instructions. For calls to the more regular core functions, the function signature is retrieved from a table and the arity of the call can be checked.

2.3. Static analysis and typechecking

Much of the power of XPath/XSLT comes from the ability to analyse the static context of sections of the expression and either determine errors (e.g. `1 + 'foo'`), find optimisations (e.g. `@foo/bar` will always yield the empty sequence `()`) or determine whether dynamic checks will be required (e.g. `string(foo)` will require the `child::foo` step to be checked for returning a result with a cardinality of zero or one.)

To do this requires a complete system to perform principally static type checking, and with the associated issues of type inference in operations such as arithmetic and determining when atomisation is needed, constitutes the hardest part of the development.

A static context is passed down through the recursive compilation, and all results have a type/cardinality computed for them. (The computed types are added as a JavaScript property to the elements – additionally writing their string values as `@type` attributes during development helps debugging as they appear in serializations of the instruction tree, but these are ignored by Saxon-JS.) Some instructions will need to refer to the type of the *current context item*; others may alter what the current context item is (e.g. `forEach`) and consequently alter its type for some of the children. Functions have definitive signatures which apply type constraints on their arguments and provide types for their result.

All these point to a requirement for a generic static type check mechanism which, given an instruction node and the required type from its parent context, either returns the construct if considered "type-safe"[10] or returns it surrounded with a suitable cast instruction if such is needed and permitted or detects and reports irreconcilable errors or wraps potentially errant instruction subtrees with suitable runtime check directives. This was achieved by transcribing the Java-based typechecker used in *Saxon-HE*[11] to a JavaScript equivalent, which uses many of the constants and tags in exactly the same form – this reduced transcription errors considerably.

Saxon-JS itself requires a type hierarchy model for runtime to satisfy `instance of` queries. This principally involves the built-in atomic types (`xs:NCName`, `xs:dayTimeDuration`…) and is defined alongside Javascript objects `Atomic.`*`typeName`* which acts as wrappers around suitable XDM implementations.

For static analysis however, more detail is required, particularly adding cardinality, so a compound `Type` object is used (wrapping a pointer to the singleton base type and a cardinality object). This is then used to generate an ensemble of type objects dynamically, such that many of the assertions and checks can be made using identity tests. For example `t = makeType("xs:string?")` produces a `Type` object such that `t.baseType == BuiltinAtomicType.STRING` and `t.card == StaticProperty.ALLOWS_ZERO_OR_ONE`. With this mechanism we can compute type assertions and checks quite smoothly.

2.4. Evaluation

With the execution plan now constructed as an SEF tree, then the expression can be evaluated, just like any other XPath subtree found within an XSLT stylesheet.

[10] Some operations effectively operate polymorphically.

[11] Perhaps one of the most critical sections of that product in terms of standards comformance.

The dynamic context must first be initialised. This involves setting the initial context item (if any) and setting up the values of the supplied parameters into their appropriate value storage slots. (When run under `xsl:evaluate` generating instructions for these are present as named subtree children.) Then the internal `SaxonJS.Expr.evaluate(SEF,context)` process is called. This will return an iterator over the results (or throw an error!), which will be treated as the result of `xsl:evaluate` and futher processed in the normal way.

3. Pure JavaScript XPath evaluation

The bulk of the work above was geared to supporting dynamic evaluation of XPath expressions within XSLT stylesheets run under Saxon-JS, through the `xsl:evaluate` instruction. But in the process we have constructed all the machinery to support evaluation from JavaScript itself, effectively through the compound:

```
evaluate(compile(parse(xpath),staticContext),contextItem,params)
```

This has been implemented as a function

```
XPath.evaluate(xpath,contextItem?,options?)
```

which will carry out the evaluation of the given XPath expression, with an optional binding to the initial context item, and a set of options which describe aspects both of the static (e.g. `xpathDefaultNamespace`) and the dynamic (e.g. `params: {conference : "XMLPrague", year: 2017}`) context, as well as controlling the result format[12].

This then means that with the Saxon-JS runtime loaded, but no XSLT stylesheet, it will be possible to evaluate XPath3.1 expressions. As an example here is a webpage on Figure 2 which sets the time on a number of contained clocks, both digital and analogue.

This is the JavaScript within that web page:

```
var thisDoc = window.document;

var timezones = {
  "London": "PT0H",
  "New York": "-PT5H",
  "San Francisco" : "-PT8H",
  "Delhi" : "PT5H30M",
  "Tokyo" : "PT9H"
};

function setClocks() {
```

[12]How a sequence of result items is presented - consistently as an array, "smart" (null, singleton or array) or as a (potentially lazy) iterator.

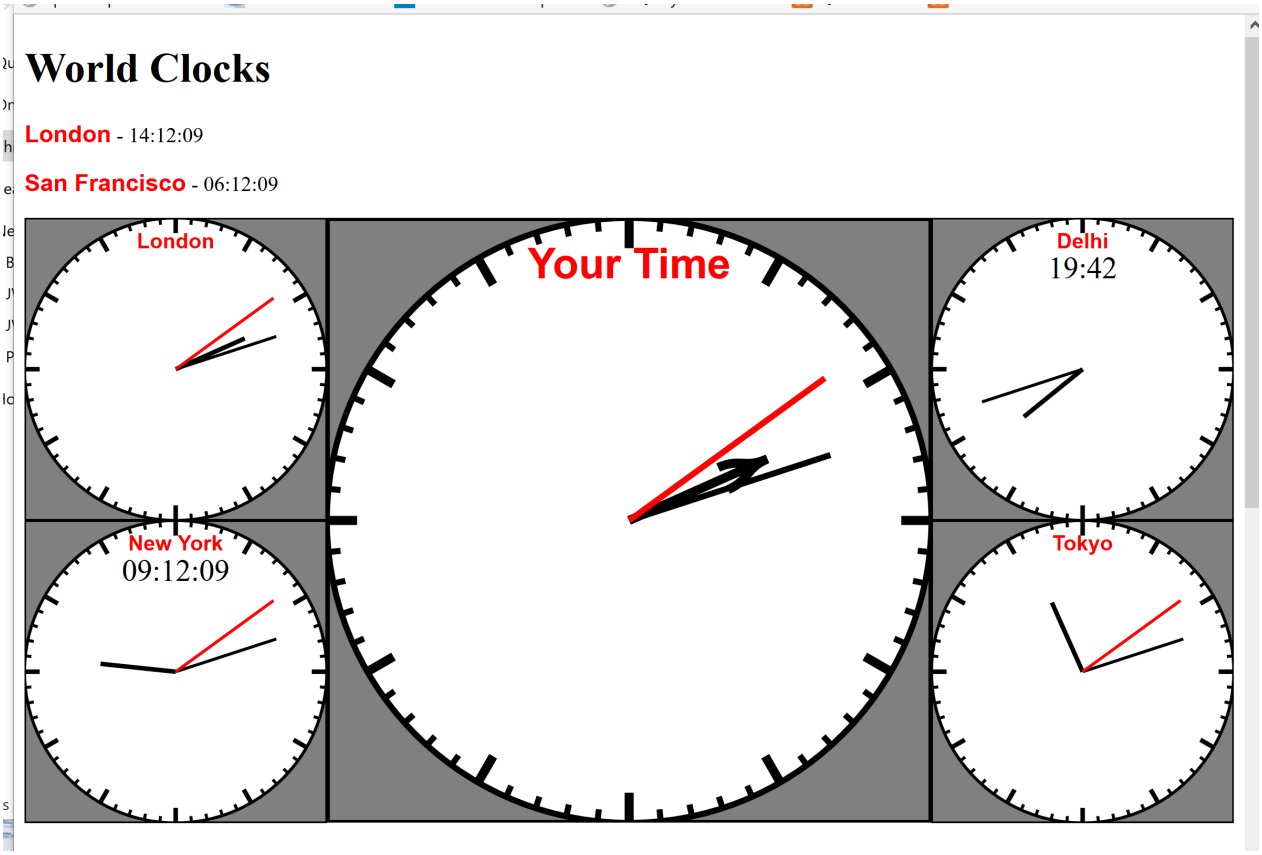


Figure 2. Clocks set by XPath

```
  var clocks = SaxonJS.XPath.evaluate(
    ".//*[tokenize(@class)='clock']",thisDoc,{resultForm:"array"});
  for(var i =0; i &lt; clocks.length; i++) {
    setClock(clocks[i]);
  }
}

function setClock(clock) {
  var findTime = "let $loc := normalize-space(.//*[@name='city']), "+
    "$t := if($loc = ('Local Time','')) then current-time() " +
    "else adjust-time-to-timezone(current-
time(),xs:dayTimeDuration(map:get($timezones,$loc)))" +
    "return map{'hour':hours-from-time($t), 'minute' : minutes-from-
time($t)," +
    " 'second' : floor(seconds-from-time($t))}";
  var time = SaxonJS.XPath.evaluate(findTime, clock,
                                    {params: {"timezones": timezones}});

  var findIndicators = "map:merge(for $class
                                  in ('hour','minute','second') " +
      "return map:entry($class, array {.//*[@class=$class]}))";
```

```
    var timeIndicators = SaxonJS.XPath.evaluate(findIndicators,clock);

    var computeAngles = "let $m := .?minute " +
      "return map{'hour':(.?hour mod 12) * 30 + $m div 2,
                  'minute' : $m * 6,
                  'second' : .?second * 6}";
    var angles = SaxonJS.XPath.evaluate(computeAngles, time);

    for(var part in timeIndicators) {
      var nodes = timeIndicators[part];
      for(var i = 0; i &lt; nodes.length; i++) {
        var node = nodes[i];
        if(clock.namespaceURI == "http://www.w3.org/2000/svg" &amp;&amp;
           !(node.localName == "tspan" || node.localName == "text" )) {
          node.setAttribute("transform","rotate("+ angles[part] + ")");
        } else {
          var t = time[part];
          node.textContent = t &lt; 10 ? "0"+t : t;
        }
      }
    }
  }
```

Each of the clocks (both HTML and SVG forms) are identified as being in class `clock` and each may contain an element with an attribute `name="city"`. For each clock the possible timezone offset is found from the name of that city (assumed the text content of the naming element) and the appropriate adjusted current time computed as separate hour, minute and second components. Then the indicators within that clock are found through an XPath returning a map of discovered nodes (which might be multiple and have `@class` either hour, minute or second) with suitable component label keys. Finally the time for each indicator is set: SVG non-text items are rotated by the appropriate angle using a `transform` attribute; all others have their text content set to the given number.

4. Testing

Very early work merely took an XPath expression and generated the candidate SEF, which was serialized and compared against what Saxon-EE would produce for the same expression. This was a very effective development method used throughout the work on the reasonable assumption that Saxon produces very highly compliant execution and that the Saxon-JS runtime had been tested by running with a server-side JavaScript interpreter such as Nashorn.

But there are many many aspects to XPath3.1, for which the dynamic compiler would have to be checked, so some (semi-)automated approach was attractive, or more likely essential. Obviously the QT3 testsuite has a large number of tests for

XPath and XQuery (more than 18,000 for XPath alone). Could we build a suitable test harness that ran some of these *entirely within the browser*? Yes we could and it proved to be highly effective, both in testing the parse/compile process and also in exercising and debugging the Saxon-JS runtime within the browser over more of the obscure facets of the QT3 test suite.

4.1. QT3 testing in the browser

This was the stimulus for getting some implementation of `xsl:evaluate` going very early in the project. It enabled an XSLT stylesheet to be written that processed the QT3 testsuite, passing each required test to `xsl:evaluate` and checking the result assertions and reporting the results *entirely within a webpage*. Development then became a question of running appropriate test sets by loading the web page, which loaded up the repertoire of QT3 test sets, clicking on those (groups) to be run and examining the results (assuming the JavaScript hadn't crashed!) In the case of failure of a specific test we can also display a serialisation of the compiled code to aid debugging.

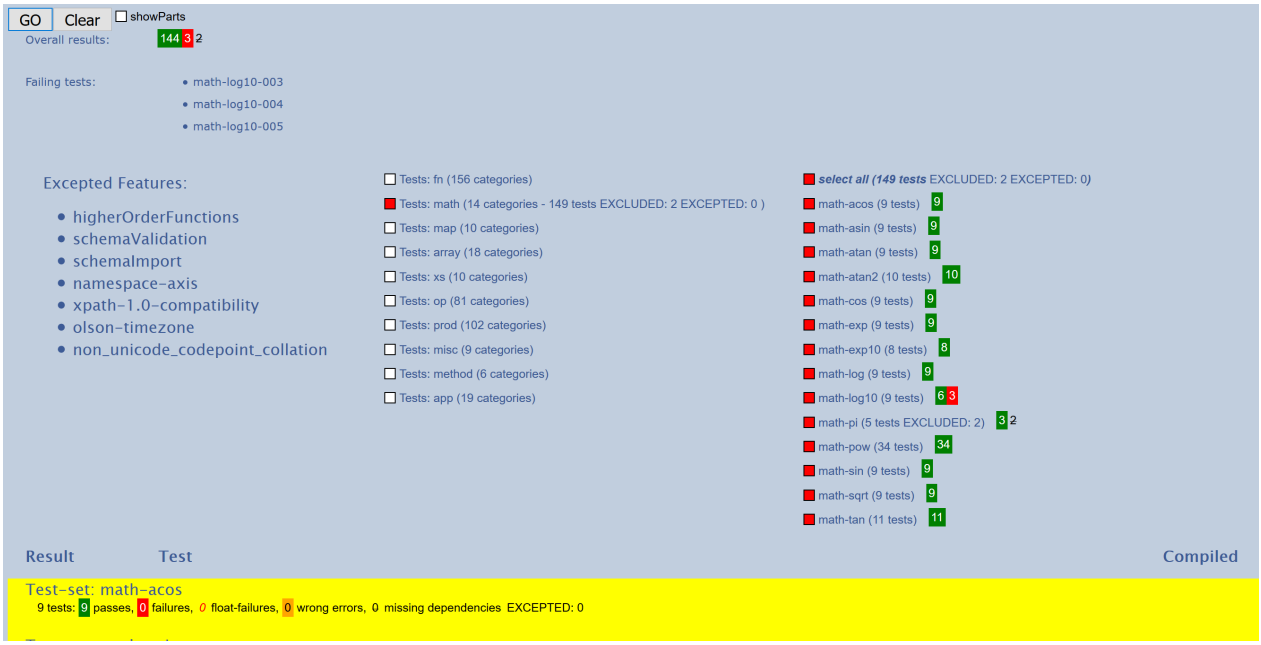


Figure 3. Testing the math: functions

This approach was used for almost all the development of the dynamic evaluation. As of writing some 18,000+ of the tests are passed with only between 110 and 190 failures, depending upon the browser used[13]. (These cover even tests for recent features such as arrays, maps, lookup operators, arrow application operators and so forth.) Here is an example of processing the `math:` function tests.

[13]There are some 4,000+ additional tests that are excluded because their dependencies include either XQuery or other features (such as higher-order functions or UCA collation) not supported by Saxon-JS

4.2. Comparing browser coverage

One of the advantages of running the QT3 tests entirely within a browser-borne setting is that performance can be checked for a variety of browsers, merely by loading the web page into the browser to be tested, clicking a few buttons and waiting for the result reports to be displayed. If we then arrange to save the final web page (`Save As...`)[14] then we have a machine-readable record of the results. By writing an XSLT stylesheet that reads in these pages[15] a comparison in coverage between the browsers can be generated. Here is the top-level comparison in terms of total errors[16]:

QT3tests - Saxon JS - browser comparison

Browsers tested:

Browser	Results					
Chrome	18347	116	173	~~4212~~	*49*	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (K
Edge	18325	138	173	~~4212~~	*49*	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (K
Unknown(IE?)	18303	188	173	~~4212~~	*21*	Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; .NET4.0C; .NI
Firefox	18344	117	175	~~4212~~	*49*	Mozilla/5.0 (Windows NT 10.0; WOW64; rv:50.0) Gecko/20100101 I
Opera	18347	116	173	~~4212~~	*49*	Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KH
Safari	18315	177	172	~~4212~~	*21*	Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/534.57.2 (KH

Excluded features

- higherOrderFunctions
- schemaValidation
- schemaImport
- namespace-axis
- xpath-1.0-compatibility
- olson-timezone
- non_unicode_codepoint_collation

Excepted test-sets

Test set	Reason
fn-collection	
fn-default-language	
fn-element-with-id	Crashes in Edge - Schema not detected?
fn-error	
fn-id	

Figure 4. Comparing browser results

---

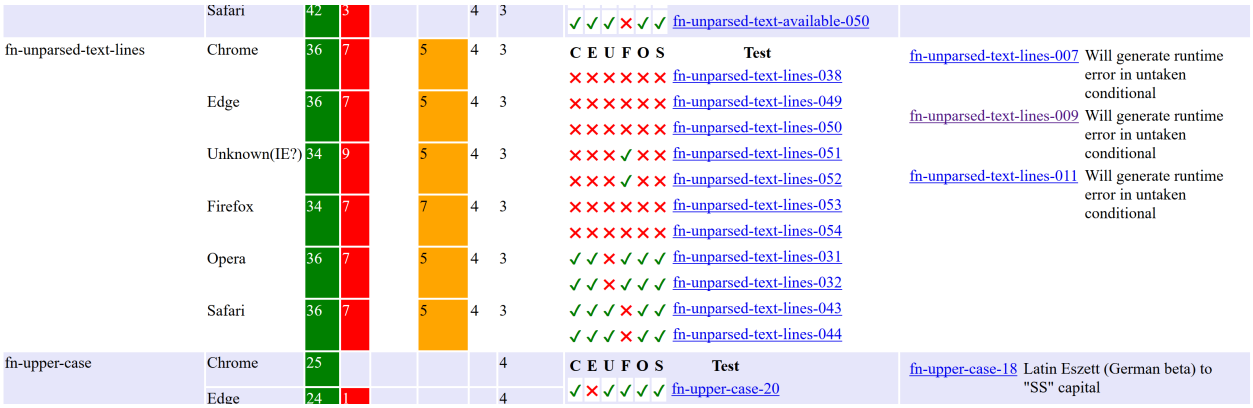
at present. A smaller number of tests (perhaps <100) are excluded specifically because they impose unrealistic demands (e.g. a map with 500,000 entries, which crashes Safari, but interestingly not the other main browsers) or are subject to some dispute (particularly so in case of statically inferred errors that dynamically will not occur)

[14]For the Edge and Internet Explorer browsers, the menu save is only the original source XHTML: to save for these browsers involves an "Inspect Element " action and a copy-and-paste of the whole active webpage DOM.

[15]Safari excluded, these are well-formed XML, so `doc()` suffices. Safari uses a binary `.webarchive` format, which can be read and decoded using EXPath File and Binary extension functions and `parse-xml()` on the resulting string.

[16]Green denotes successful tests, red failures, orange expected failures but with the wrong error code. Strike through are the number of tests not run, either by specific exception or excluded by unsupported features. Red italics indicate a failure where the XPath expression used the `xs:float` type – this is coerced to a JavaScript `Double` in Saxon-JS so results are computed to higher precision than anticipated or permitted and exact equality comparisons in the test assertions can fail.

Detailed comparison on failing tests by browser is also reported, as shown here, where different successes for tests of `unparsed-text-lines()` are revealed. (The last column lists tests that were not run deliberately and the reason why: in this case the test assertion of error is a pessimistic static assumption.)



	Safari	42	3			4	3	✓✓✓✗✓✓ fn-unparsed-text-available-050	
fn-unparsed-text-lines	Chrome	36	7		5	4	3	C E U F O S Test	fn-unparsed-text-lines-007 Will generate runtime error in untaken conditional
								✗✗✗✗✗✗ fn-unparsed-text-lines-038	
	Edge	36	7		5	4	3	✗✗✗✗✗✗ fn-unparsed-text-lines-049	fn-unparsed-text-lines-009 Will generate runtime error in untaken conditional
								✗✗✗✗✗✗ fn-unparsed-text-lines-050	
	Unknown(IE?)	34	9		5	4	3	✗✗✗✓✗✗ fn-unparsed-text-lines-051	
								✗✗✗✓✗✗ fn-unparsed-text-lines-052	fn-unparsed-text-lines-011 Will generate runtime error in untaken conditional
	Firefox	34	7		7	4	3	✗✗✗✗✗✗ fn-unparsed-text-lines-053	
								✗✗✗✗✗✗ fn-unparsed-text-lines-054	
	Opera	36	7		5	4	3	✓✓✗✓✓✓ fn-unparsed-text-lines-031	
								✓✓✗✓✓✓ fn-unparsed-text-lines-032	
	Safari	36	7		5	4	3	✓✓✓✗✓✓ fn-unparsed-text-lines-043	
								✓✓✓✗✓✓ fn-unparsed-text-lines-044	
fn-upper-case	Chrome	25					4	C E U F O S Test	fn-upper-case-18 Latin Eszett (German beta) to "SS" capital
	Edge	24	1				4	✓✗✓✓✓✓ fn-upper-case-20	

Figure 5. Detailed error comparison

4.3. Testing the JavaScript API

Testing the JavaScript API required a slightly different approach. If we assume the QT3 tests run through `xsl:evaluate` mechanism has proved both the XPath→SEF and SEF(args)→result paths are correct, we just need to test the calling options for `SaxonJS.XPath.evaluate()`. This is most conveniently carried out by constructing a web page with tabular entries containing argument components of XPath expression, context item and options, with a simple JavaScript that iterates across table rows, accumulating these components from textual values of cells, then calling the dynamic evaluator and writing the results (possibly serialized) into another result-holding cell.

5. Performance

There are two aspects of performance to consider. The first is accuracy of execution and the degree of conformance to standards. As has been indicated in the previous section, we've managed to achieve a very high degree of such as measured by the QT3 test suite, failing ~0.5% of the tests. Some errors are inevitable (e.g. the use of `Double` for `xs:float`), others are in very obscure cases (uncaught errors that actually can never be triggered), a few are just wrong and will eventually get corrected. In practice we consider the product has high enough conformance for production use.

The second of course is execution speed. This we have not measured directly yet, but the best indicator we have so far is the execution of sections of the QT3 test suite. As an example, on a high-end (2016) Windows laptop, the 3446 tests for the operators (`op-except`, `op-numeric-add` etc) are processed completely in 27

seconds under Firefox and 15 seconds under Opera, from original "click" to finished results, including execution of all the surrounding XSLT test harness[17]. As for each of the tests i) it needs to be compiled and executed and ii) its result assertions tested (some of which contain XPath expressions that must be evaluated, thus needing additional compilation), there are >3446 XPath evaluations required. This suggests that each evaluation takes somewhere in the 2-5 ms region. As we anticipate most use of these dynamic features will involve small numbers of XPath expressions, with "a human in the loop", we do not forsee significant problems with compilation/execution performance.

6. Future Developments & Conclusions

Like the Stylesheet Export File, we use XML trees (in this case in-browser DOM trees) to represent the intermediate parse results and generated code. But the manipulation of these trees involved is not complex, mostly concerned with testing node name, child and/or attribute existence and iterating over children. It has been suggested (in [3]) that the export format might also have a JSON-based alternate. If this was the case then much of the compiliation code described here could be recast relatively easily to use a JSON representation as the main data type e.g.

```
1 to 2 * 10  →
{ name : "RangeExpr",
  children : [
    { name: "IntegerLiteral", value: 1 },
    { name: "MultiplicativeExpr", op: "*",
      children: [
         { name: "IntegerLiteral", value: 2 },
         { name: "IntegerLiteral", value: 10 }
               ] }
            ]
}
```

We've suggested earlier that the reduction phase of the initial parse might be improved by a smarter use of callbacks within the parse-tree former. Obviously during the compilation process there will be many more opportunities to optimise the resulting code. The easiest would be the complete evaluation of literal subexpressions, i.e. those for which there is no dependency on execution context within the subtree. In such cases the compiled subtree can be evaluated and its sequence result projected as a sequence of suitable instructions, based on the literal forms. Whether it is worth doing this however is a moot point, as, unless the same (dynamic) XPath is going to be executed repeatedly, little extra would be

[17]Most browsers show similar performance within perhaps a factor of 2, with the noteable exception of Internet Explorer which seems to be consistently about 5x slower than the rest.

gained and indeed additional time would be taken up in determining *whether* an evaluation could be performed.

This leads on to the issue of whether an intermediate compiled output is desirable. Earlier versions of Saxon (within a Java environment) used a pair of functions `saxon:expression()` and `saxon:eval()`, with the former producing a stored expression that the latter would use to evaluate against a given dynamic context. A similar partitioning within JavaScript could be provided easily.

6.1. Conclusion

This paper has shown that when an XPath "instruction execution engine" is available within a browser context, it is possible to add a dynamic XPath evaluation facility provided that i) an accurate and efficient XPath parser is also available and ii) very accurate code for static type checking, coercion and casting is developed. Development of such a feature is aided considerably by the existence both of an external "oracle" to demonstrate what a correct "execution plan" should be for a given XPath expression (in this case using Saxon-EE and examining generated SEF) and the early construction of a test harness to exercise the QT3 test suite (in this case as an XSLT stylesheet invoking via the `xsl:evaluate` instruction.)

In the case of Saxon-JS and the `XPath.js` additional module we have developed an implementation with very high levels of conformance to the XPath 3.1 specification, demonstrated by rapid running of the the QT3 test suite *entirely within the browser*. We have also been able to demonstrate and record the (small) differences in conformance between the half-dozen major browsers.

References

[1] O'Neil Delpratt and Michael Kay. *Multi-user interaction using client-side XSLT.*. 2013. `http://archive.xmlprague.cz/2013/files/xmlprague-2013-proceedings.pdf`

[2] Sergey Ilinsky. *XPath 2.0 implementation in JavaScript* . 2016. `https://www.openhub.net/p/xpath-js`

[3] Debbie Lockett and Michael Kay. *Saxon-JS: XSLT 3.0 in the Browser.*. 2016. `http://www.balisage.net/Proceedings/vol17/html/Lockett01/BalisageVol17-Lockett01.html`

[4] Steven Pemberton. *Invisible XML.*. 2013. `http://www.balisage.net/Proceedings/vol10/html/Pemberton01/BalisageVol10-Pemberton01.html`

[5] Gunther Rademacher. *REx Parser Generator*. 2016. `http://www.bottlecaps.de/rex/`

[6] *XQuery in the Browser*. 2016. `http://www.xqib.org/index.php`

Soft validation in an editor environment

Schematron for non-technical users

Martin Middel
FontoXML
<martin.middel@fontoxml.com>

Abstract

To allow the use of Schematron[1] in a quickly changing environment like an editor, understandability is crucial. An author may not be an XML expert, so they must be guided in resolving the messages generated by Schematron.

A good understandability rests on two pillars: performance and user interface. An author needs constant feedback on the current state of the soft validation. The author must know which places in the document need attention, and how to resolve them. The report must then update as soon as possible to enable the author to see the result of a modification they just made.

To ensure good performance, a number of technical problems have been solved, this includes a novel dependency tracking system.

1. Introduction

FontoXML is an editor for XML content, used by non-technical authors. At this moment, FontoXML offers a standard editor for DITA 1.3[1] and can be configured to support any schema including TEI[2], JATS[3], Office Open XML[4] and a number of DITA specializations.

Since non-technical authors have no knowledge of XML or the schema, they will not understand nor be able to fix invalid documents. Therefore we guide the author to prevent them from creating invalid documents; we ensure loaded documents remain valid at all times. Valid in this context means both well-formed (no unclosed tags), but also schema-valid (titles can not contain list items).

However, some restrictions should not always be enforced. Constraining the length of a title will destroy usability, because typing text in it will suddenly be disabled. Additionally, schema restrictions may need to be relaxed in various use cases. For these reasons, we have implemented soft validation in the editor, which essentially are recommendations instead of requirements.

[1] http://docs.oasis-open.org/dita/dita/v1.3/dita-v1.3-part3-all-inclusive.html
[2] http://www.tei-c.org/index.xml
[3] https://jats.nlm.nih.gov/index.html
[4] http://www.ecma-international.org/publications/standards/Ecma-376.htm

2. The case for soft validation

In our experience adapting the FontoXML editor for various clients, we have seen two major cases for soft-validation:

1. Adapting content from a permissive schema used for editing to a strict schema used for publishing content.

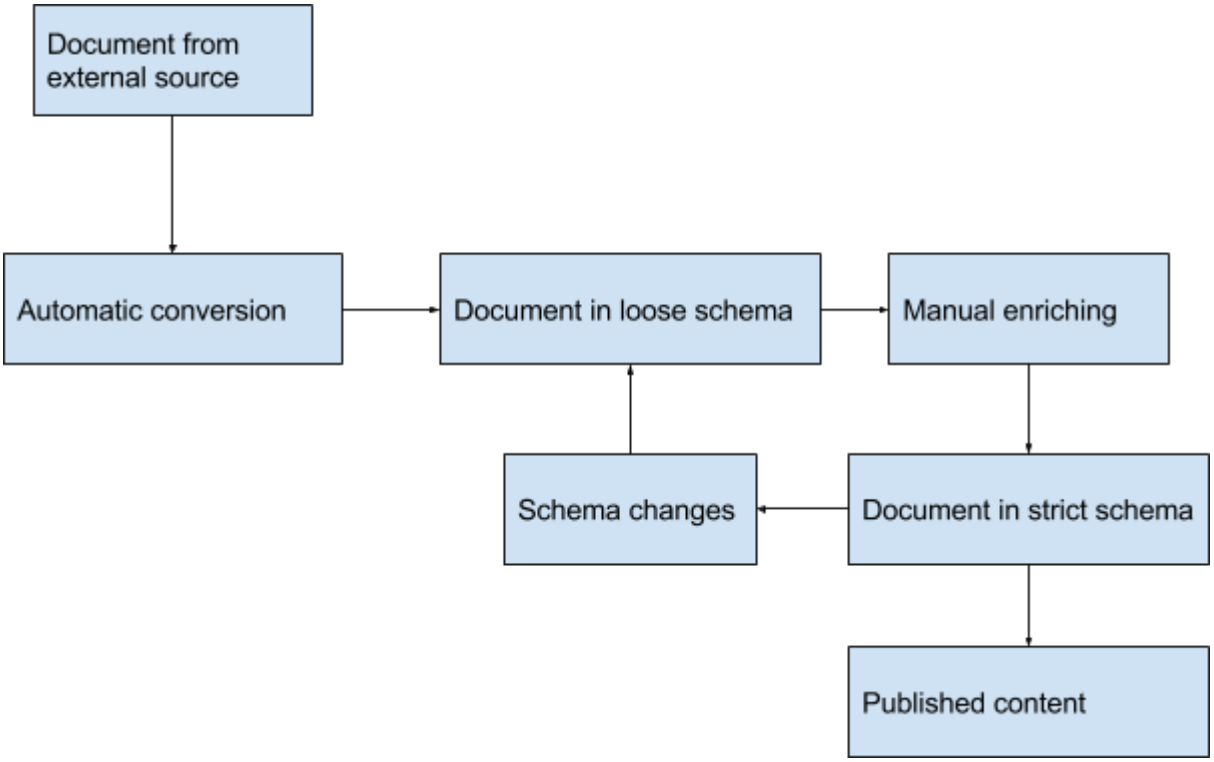
   One of our clients (automatically) imports content from a word processor. They can not automatically tag most elements, apart from distinct paragraphs, so this enrichment is a manual process. After the enriching phase of the workflow, all elements must be tagged: title paragraphs must be title paragraphs, abstract paragraphs must be abstract etc. Using the stricter schema as soft validation on top of the permissive schema provides valuable feedback during the enrichment process.

2. Schema with varying publication-specific constraints.

   Another client maintains a very high number of different schemas, each fitting their unique purpose. A schema for writing a book for teaching a language is different from an exercises book for learning math.

   To allow the same XML pipeline to be used for all of these schemas, the actual schemas are the same. Different soft-validation rules are enforced over them, in order to tailor to specific publication needs. This also allows for easy upgrades from one version of the (overlaying) schema to the next: the document is always schema-valid.

Soft-validation can also be used to guide an author into adhering to a style guide, such as writing short titles or paragraphs. These rules can often be expressed as textual constraints on certain elements (such as maximum character count or minimum word count).



3. Schematron

Schematron is a widely used standard for performing 'soft-validation': describing certain structures in XML which are technically valid, but deemed undesired.

In short, Schematron is a declarative format that works like this:

- Define “rules”, selecting nodes which should be validated.
- Per rule, define reports and asserts testing these nodes.
- Both rules and tests are XSLT patterns, which are very similar to and can be transformed to standard XPath queries without problems.

```
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron">
  <sch:title>An example schema</sch:title>
  <sch:rule context="p">
    <sch:assert test="@class">A paragraph must have a class</sch:assert>
  </sch:rule>
</sch:schema>
```

The rule context and test expressions can be transformed to a single XPath[2] query which can identify nodes that would trigger the assertion or report. For performance, it does not make much sense to split these queries: an assert for a body asserting a title node is present somewhere in the document still introduces a full-document-scan.

The Schematron reference implementation is based on XSLT. As FontoXML is implemented in pure JavaScript and aims to have as few server-side components

as possible, there are not many usable XSLT processors around. Also, because FontoXML is an editor, we expect many, generally very localized changes to happen very frequently. We do not want to fully process a large document multiple times per second. In order to provide a fluid user experience, we need the editor to update with a rate of 60 frames per second, just like a video game. This leaves us with a budget of roughly 16ms (1000ms / 60) per frame. Within this frame budget we need to do all the JavaScript updates and still allow time for the browser to do its layout and paint work. Another concern is the size of both documents and schemas: one of our clients loads documents up to 2MB in size, another client uses up to 3500 (automatically generated) Schematron rules. Fully processing these documents and their rules simply won't scale.

Because of these concerns, we have decided to roll our own implementation, optimized specifically for evaluating Schematron rules in an editor environment.

4. Implementation

4.1. Requirements

We decided to build our own optimized Schematron engine with the following concerns:

- Must be client-side, fully written in JavaScript
- Must be real-time (reactive to changes in the document)
- Must have quick-fix functionality (understandable by non-XML experts)
- Must not hold the UI thread hostage (remember: Javascript is single threaded)
  We aim to only work continuously for 16ms. Any more than this will make the browser framerate drop to below 60FPS.

4.2. Written in JavaScript, running client-side

The Schematron engine should run client-side so that it can provide feedback as fast as possible and does not introduce a server-side dependency. The editor should be able to run off-line: network latency and enterprise firewalls are largely not in our control. Besides this, the battery cost of a GSM or WiFi modem on mobile devices has to be taken into account.

The best XSLT processor in JavaScript at the time or writing is Saxon-JS[5]. For us, using this approach is not an option because of scalability. The naive XSLT-based approach would require the entire document to be processed after every change. As some of our clients work with XML documents ranging in the megabytes, this approach becomes infeasible due to performance constraints. The

[5] http://www.saxonica.com/saxon-js/index.xml

XSLT is not in our control, and we can not say anything about it. Ideally, we'd want to regard it as a black box.

A Schematron engine works using XSLT expressions, which are an extension of XPath queries. There are a few JavaScript XPath implementations available:

- Google's wicked-good-xpath[6], which is an XPath 1.0 implementation.
- XPath-JS[7], also an XPath 1.0 implementation.
- XPath.js[8], an XPath 2.0 implementation.

We also had the choice of building our own XPath engine, which would allow for anything we'd ever want. Because of our requirements stated earlier, we want to have tight control over the performance characteristics of the XPath implementation. We also wanted to use the same XPath implementation in other parts of the FontoXML editor and related products. We therefore decided to invest the time and build our own. This ensures we get the control we want, and also enables us to more easily implement any further optimizations or extensions in the future. For this implementation, we decided to implement an XPath 3.1[5] engine. This is the latest iteration of the XPath standard, which is currently in the candidate recommendation phase.

4.3. Real-time updates

Any change to the XML document could cause any of the Schematron rules to have different results. Because the size of the document is variable and could be very large, re-evaluating every rule on the entire document after every change would take too much time. We need a way to reduce the amount of Schematron rules that need to be to re-evaluated after each edit.

Let's take the XPath query `@someAttribute eq 'value'`. Given the element `<element someAttribute="value"/>`, we can see it will evaluate to `true()`.

If we change the value of this attribute to `<element someAttribute="some other value"/>`, we can see it will evaluate to `false()`.

If, instead, we change the node by adding an attribute to create `<element someOtherAttribute="meep" someAttribute="value"/>`, we can see the XPath will not change its' original value: `true()`.

We could say that the result of an XPath query is determined by the parts of the DOM it 'looks at'. These are its dependencies. These dependencies can be invalidated by changes on nodes. For instance, to be able to determine the result of the ancestor axis, the parent relation of all ancestors will be evaluated, this will introduce a dependency on the parent-child relation of these nodes.

[6] https://github.com/google/wicked-good-xpath
[7] https://github.com/andrejpavlovic/xpathjs
[8] https://github.com/ilinsky/xpath.js

The DOM standard defines the MutationObserver interface to allow a consuming API to react to changes in the DOM. It does this by recording each mutation as an object, called a MutationRecord[3], and exposing these objects for further processing after the mutation completes:

```
{
  type: 'childList' | 'attribute' | 'characterData',
  target: Node,
  addedNodes: Node[],
  removedNodes: Node[],
  nextSibling: Node?,
  previousSibling: Node?,
  attributeName: String?,
  oldValue: String?
}
```

The same interface is implemented by our editor to track changes made in an XML document. These records are used throughout the editor, for instance, for implementing the undo/redo functionality.

By tracking the dependencies of queries and using MutationRecords, we can make it so that for any given edit, only the affected Schematron rules (i.e. XPath queries) has to be re-evaluated, instead of all of them. For example, any rules on the presence of certain elements do not have to be evaluated if we're only working on an attribute. This removes the bulk of the processing needed to keep the Schematron results up to date.

We store the dependencies for our XPath queries in a two-level Map data structure:

```
node -> type -> query[]
```

This way, given a MutationRecord, we can look up the possibly affected queries in constant time (`O(1)`).

We update this data structure whenever we run an XPath query. First, we transform the XPath string to an abstract syntax tree using a parser generated with the wonderful peg.js parser generator[9]. We then compile this syntax tree to a set of DOM traversals.

By using a facade for accessing all DOM relations, we can record the traversed relations as dependencies of a specific operation such as the evaluation of an XPath query. As a bonus, this facade makes the engine independent of the interface of any specific DOM implementation, as it can be used as a translation layer.

The facade simply consists of functions such as the following:

```
class DependencyTrackingDomFacade {
  getChildNodes (node) {
```

[9] https://pegjs.org/

```
      registerDependency(
        this._dependenciesByNodeIdAndKind,
        node,
       'childList');
      return this._dom.getChildNodes(node);
    }
    getParentNode (node) {
      var parentNode = this._dom.getParentNode(node);
      if (parentNode) {
        registerDependency(
          this._dependenciesByNodeIdAndKind,
          parentNode,
          'childList');
      }
      return parentNode;
    }
    getAttribute (node, attributeName) {
      registerDependency(
        this._dependenciesByNodeIdAndKind,
        node,
        'attribute');
      return this._dom.getAttribute(node, attributeName);
    }
    getData (node) {
      registerDependency(
        this._dependenciesByNodeIdAndKind,
        node,
       'characterData');
      return this._dom.getData(node);
    }
  }
```

This dynamic analysis of an XPath query allows us to regard the XPath as a black box. We do not impose any additional requirements to the structure of XPath expressions, determining for instance streamability. It also does not block further improvements such as static analysis [4]. This approach for dynamic analysis is simple to implement. Although it does not make hard XPath queries easier to evaluate, it does provide a base for memoization[10]. In the future, we will want to use static analysis to provide partial queries, indices or query simplification.

4.4. Putting everything together

Given the following Schematron snippet:

[10]Memoization is a technique used to reuse the result of a function if its' parameters are the same as a previous execution.

```
<sch:rule context="/html/body/div/p">
  <sch:assert test="not(@class) or @class=('title', 'intro')">
    A paragraph should have either no class or the title or intro class.
  </sch:assert>
</sch:rule/>
```

Combining the context and test will result in the following query for reportable nodes:

```
/html/body/div/p[not(not(@class) or @class=("title", "intro"))]
```

Running this query will make the XPath engine 'look at' a number of properties of DOM nodes. This gives us the following dependencies:

- html -> 'childList'
- body -> 'childList'
- Every div in the body -> 'childList'
- Every p in these divs -> 'attribute'

This query will be triggered for re-evaluation by some of the following changes in the DOM:

- Adding a new body to the html element
  This changes the childList of the html element, and can possibly introduce new div elements with new p elements.
- Adding or removing divs
- Adding or removing paragraphs in these div elements
- Changing the class attribute on the paragraphs

This query will not be affected by changes like editing the contents of a paragraph or adding a title element in the head element of the html element.

Note that if the dependencies of a query have changed, it does not mean the result of the query has changed: `@someAttribute => string() => starts-with('abc')` can evaluate to true for many different values of `@someAttribute`: `'abc', 'abcd', 'abcX', etc`.

This causes a number of false positives. In the example given above, the following changes will also trigger a re-evaluation:

- Adding a head element to the html element (the childList of html changes)
- Moving around div elements in the body (the childList of body changes)

These will never change the result of this query, causing a needless re-evaluation. However, as these actions do not happen as frequently as typing text in a paragraph, the impact on performance of this limitation is minimal. The dependency tracking approach prevents re-evaluation of most queries, most of the time. This is sufficient to provide acceptable performance for our requirements. In the future we plan to investigate memoizing intermediate results in order to prevent full re-

evaluations, stopping evaluation if a part of the query resolves to the same result as before.

This also works in a subtly different way: the set of dependencies of a query may change, while the result remains the same. For example, take a query containing the conditional expression: `@A or @B`. Because the `or` expression may never evaluate `@B` if attribute A is present, removing this attribute forces it to also look at the other. The end result may remain the same, but the DOM has been traversed in a different way, causing a different set of dependencies.

4.5. Quick fix

Now that we can generate reports, and re-evaluate them quickly, we can move on to enabling users to fix any issues that have been detected.

The FontoXML editor uses the concept of 'operations', small units of functionality describing things like opening a modal, setting an attribute, wrapping a range of text, inserting a new element or any combination thereof.

These operations can be used to encode the mutations required to fix content issues, and can be mixed into the Schematron XML notation as follows:

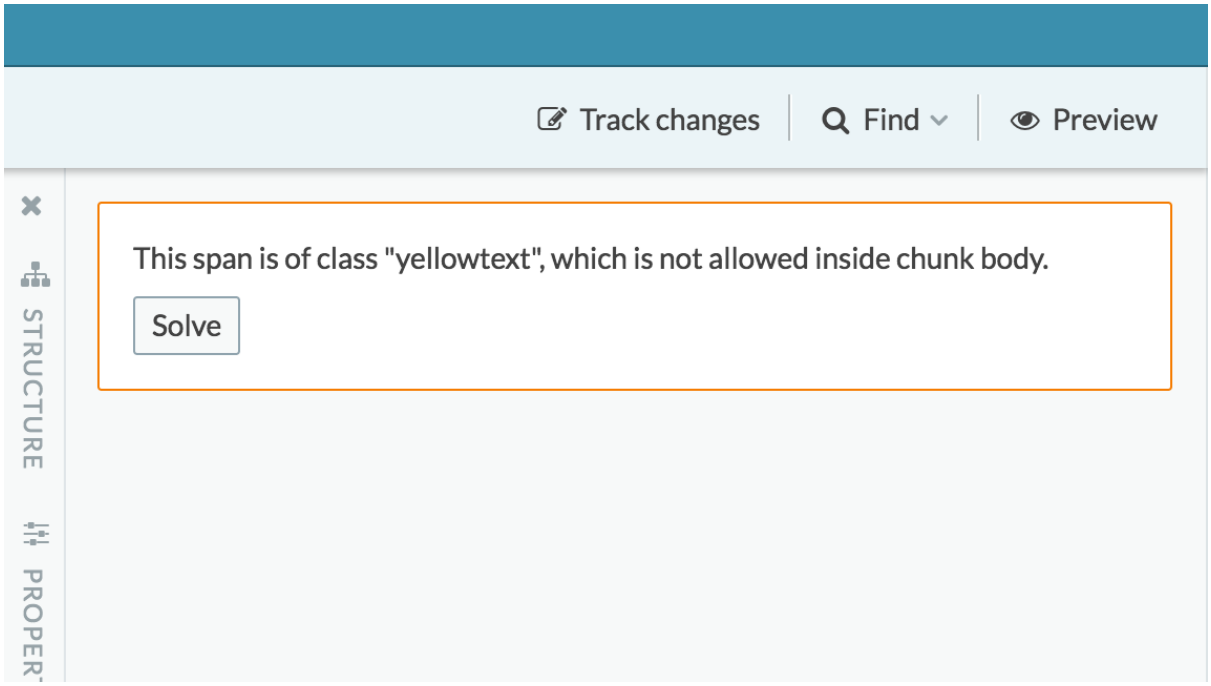
```
<sch:rule context="//span">
  <sch:assert test="@class = ('strong', 'italic')">
    <fonto:message>
      A
      <sch:value select="fonto:friendly-name(.)"/>
      must have the class 'strong' or 'italic'. It has the class
      <sch:value select="if (@class) then @class else 'No class'"/>
      . This is wrong.
    </fonto:message>
    <fonto:fix name="set-attribute" value="strong"
               label="Convert to Strong">
    <fonto:fix name="set-attribute" value="italic"
               label="Convert to Italic">
    <fonto:fix name="unwrap-node" label="Unwrap this">
  </sch:assert>
</sch:rule>
```

To make the report easier to parse, we have placed the message in an additional element.

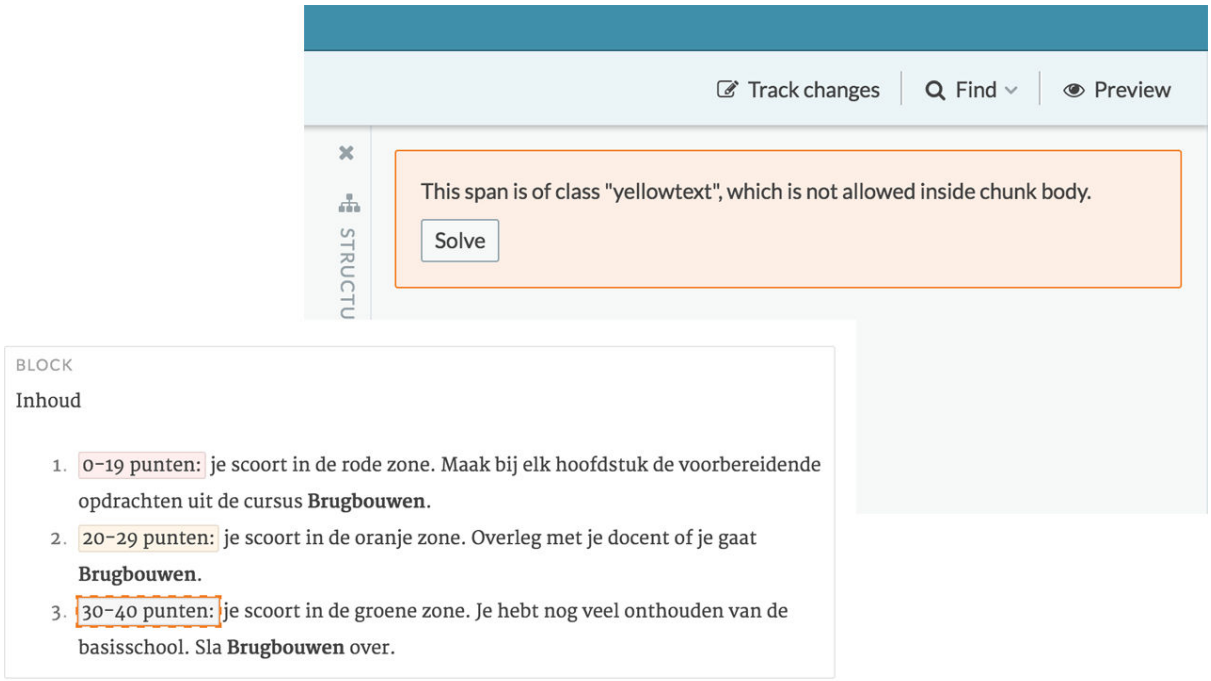
4.6. UI

The soft-validation reports can be used as a list of tasks that an author should process. As our authors are not all XML-experts, it is important to visualize as much context as possible. To do this, we did the following:

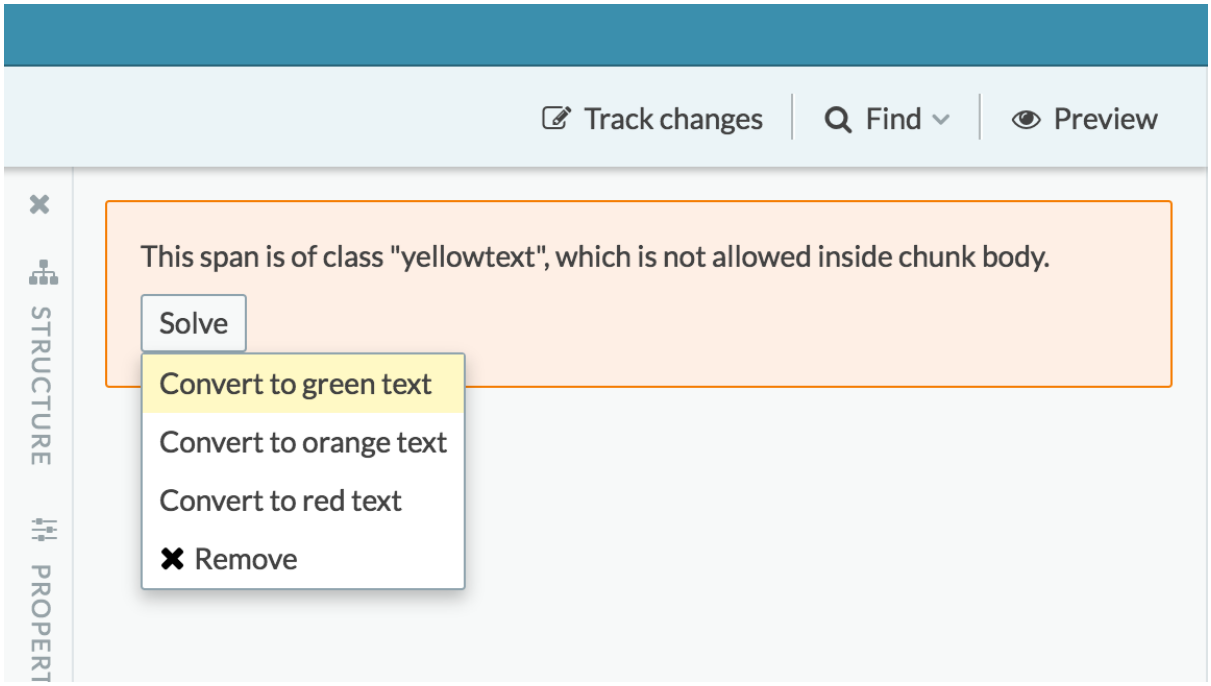
1. The reports are visualized as cards in a side panel, allowing them to be visible alongside the editor.



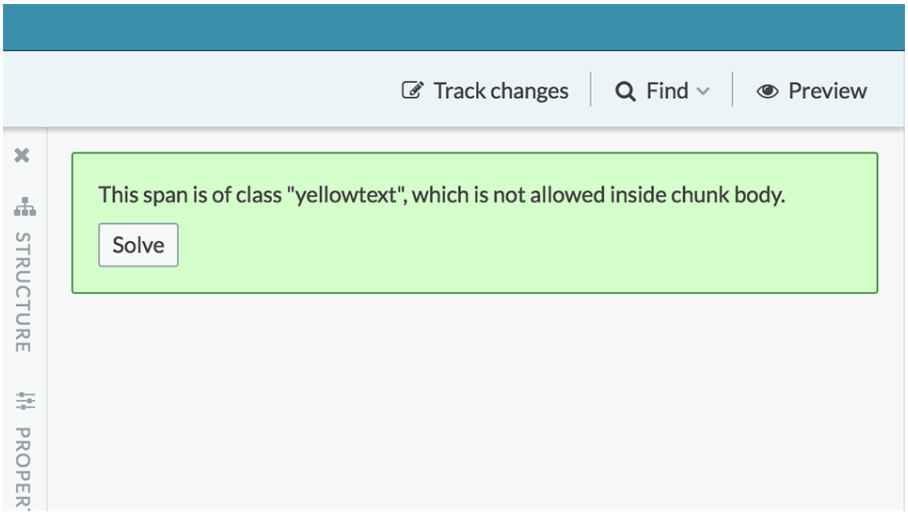
2. Clicking on a report highlights the element causing the report and brings it into view in the editor.



3. All reports having one or more quick-fix operations available display a ‘solve’ button, which opens the quick-fix menu.



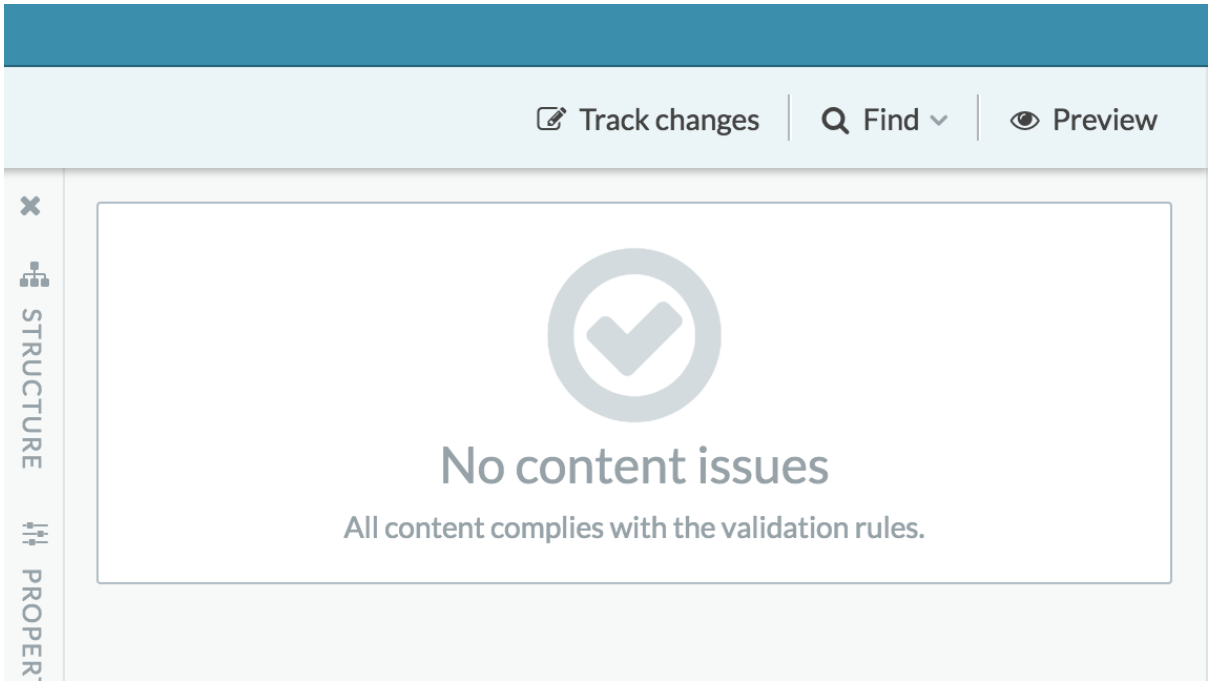
4. After resolving the error (using either the quick-fix menu or any other action), the card shows a brief animation and is removed.



Brugbouwen.

3. 30–40 punten: je scoort in de groene zone. Je hebt nog veel onthouden van de basisschool. Sla **Brugbouwen** over.

5. When all issues have been resolved, the editor reports this fact to the author.



5. Future work

5.1. Performance

At the moment, performance is highly dependent on the way the Schematron rules and tests are written. A constraint such as preventing referencing unknown ids can be written as this:

```
<rule context="@id-ref">
  <let name="idref" value="."/>
  <assert test="//*[@id=$idref]/>
</rule>
```

This rule will be rewritten to this XPath: `boolean(// *[@id-ref][let $idref := . return //*[@id = $idref]])`.

The XPath will cause us to evaluate the comparison `O(N2)` times: we will scan the whole document for id attributes once for every idref. Also, this query will introduce childList dependencies on all of the elements in the document. Building and maintaining a lookup table for all nodes with an id attribute can speed up these queries significantly.

5.2. Preventing worsening the document

In the current implementation, soft-validation rules never prevent any editing operation. However, we might want to influence a number of actions (like copy/paste or inserting new nodes) to prevent the document from getting less valid

according to these rules. An operation could be blocked if it ends up making the document fail more soft-validation tests. This needs to be thoroughly user-tested to identify the conditions where it would be useful to temporarily make the document less valid (such as when pasting content from somewhere else, or when splitting an 'invalid' paragraph using the enter key in order to convert the individual parts to a valid structure).

5.3. Using Schematron quickfix

A small schema has been whipped up to provide references to quick fix operations. In the future, these quick fix descriptions should be described using the Schematron Quickfix schema[6].

5.4. Open sourcing

We are looking into making the XPath engine open source, possibly including the dependency tracking mechanism. We expect it will have many applications outside of Schematron, like writing modern JavaScript apps that manipulate XML documents, or any other data representing a DOM.

6. Conclusions

The requirements of a soft-validation engine operating in a dynamic environment such as an editor are different from one reasoning over static documents.

Because changes in an editor are usually localized, we should not need to reprocess a full document every single keystroke. Using mutation records and dynamic dependency tracking as a way to mark queries on the DOM as dirty, we can use the edits to determine a smaller set of possibly affected queries.

Soft validation is a great help in guiding authors in writing "good" content.

By considering the soft-validation report as a "to do list", an author can keep track of their progress. By providing quick-fixes, the author always knows what to do and can efficiently resolve most content issues.

Bibliography

[1] Information technology — Document Schema Definition Languages (DSDL) — Part 3: Rule-based validation, Schematron, International Standard ISO/IEC 19757-3, Geneva, Switzerland : ISO

[2] XML Path Language (XPath) 1.0. W3C Recommendation, 16 November 1999. Ed. James Clark and Steve DeRose https://www.w3.org/TR/xpath/

[3] DOM living standard, 19 Januari 2017 on MutationRecord. https://dom.spec.whatwg.org/#interface-mutationrecord

[4] XSLT and XPath Optimization, March 2001, Ed. Michael Kay. http://www.saxonica.com/papers/xslt_xpath.pdf

[5] XML Path Language (XPath) 3.1. W3C Candidate Recommendation, 16 December 2016. Ed. Jonathan Robie, Michael Dyck, and Josh Spiegel. https://www.w3.org/TR/xpath-31/

[6] Schematron quickfix. Ed. Nico Kutscherauer. http://www.schematron-quickfix.com/

Improving text quality with automatic majority editions

How shall I count the ways?

Liam Quin
W3C
<liam@w3.org>

Abstract

This paper describes a method to improve accuracy of OCR-scanned text documents prior to conversion to XML. The paper also describes approaches to conversion of such documents to XML and consideration of the point in the conversion process at which it makes the most sense to start using XML tools.

Keywords: XML, DocBook, authoring

1. Introduction

The public domain texts on the Internet Archive (often scanned by Google Books) have made a considerable corpus of writing available. Unfortunately the texts often contain errors from the optical character recognition processes employed, greatly limiting their usefulness.

It turns out that the distributed nature of large-scale scanning of entire library shelves has meant that there are many duplicate volumes, the same edition having been scanned and processed multiple times at different institutions.

This paper describes a simple technique to harness that multiplicity and thereby produce texts containing fewer errors with relatively little human intervention compared to manual methods.

In addition, scripts to convert the texts to XML are described, mostly to illustrate techniques and possibilities.

The software is freely available for download.

2. The Problem

For many years the quality of easily-available public-domain texts has been low. In some cases this has been because of missing metadata: Project Gutenberg, one of the more widely-known projects making documents available, does not appear to take care to make edition information consistently available, greatly limiting the applicability of their work to research. Other projects often have too low a

quality to be of much use, with many transcription errors occurring within even a single paragraph.

The collaboration between Google, the Internet Archive and the academic library community has resulted in large numbers of books being scanned with very careful metadata. Unfortunately these texts turn out to contain many errors. An actual sample is shown in Figure 1.

```
He died of an
autumnal fever, . which was brought on by an intemperate
eating of melous, in the 70th year of his age, and (as is
believed) soon itfter his settlement in Rome ; but the time
of his death is uncertain, yet it must have be^n after 1478^

. because he survived Theodorus Gaza, who died in. that
year.
```

Figure 1. Sample OCR output

Our task, then, is to produce improved texts of sufficient quality that it seems worth-while to process them into XML documents and to be able to use them for research or redistribution.

3. Mitigating the Problem

The author of this paper has observed three mitigating factors to the otherwise uselessly low quality of the texts shown in the first tow figures. The first is that the Internet Archive provides the data files for the (usually proprietary) OCR program used, which one could load and use to improve the texts by hand. The author's experience is that this process takes several minutes per page so that, if one has not had the foresight to enslave enough graduate students, it is uneconomical except for the most important of texts.

The second mitigating factor, however, is that the Internet Archive often carries the result of scanning the same edition of the same book multiple times, presumably most often from multiple institutions. Figure 2 shows a different version of the sample text already shown, scanned at a different institution.

```
He died of an
autumnal fever, which was brought on by an intemperate
eating of melons, in the 70th year of his age, and (as is
believed) soon after his settlement in Rome ; but the time
of his death is uncertain, yet it must have been after 1478,
because he survived Theodorus Gaza, who died in tiiat
year.
```

Figure 2. Other versions contain different errors

It may be worth noting that the author tried scanning this same book using the same software and obtained a considerably lower error rate, with only a few errors per page. However, for that experiment, greyscale images were used rather than black and white, and at a higher resolution. This slows down scanning and increases the burden of image storing, but at any rate will not help with books that have already been scanned.

The third mitigating factor, essential to the approach to be described, is to observe that some of the versions are of markedly higher quality than others. Figure 3 shows yet another version of the same text and it is clearly of better quality. A check against the printed page also shows that the numbers are correct and that the zero in "70" is in fact a digit and not (as is common) a letter.

```
He died of an
autumnal fever, which was brought on by an intemperate
eating of melons, in the 70th year of his age, and (as is
believed) soon after his settlement in Rome ; but the time
of his death is uncertain, yet it must have been after 1478,
because he survived Theodorus Gaza, who died in tiiat
year.
```

Figure 3. A third version has fewer errors.

We could do much worse than simply choosing the best version whenever there are multiple copies of a text. But, as we hope to show, we can do much better.

4. A Majority Proposal

The first example includes approximately ten errors in seven lines of text, but if we apply a simple preprocessor to it we can reduce that to six errors. We simply delete spaces before semicolons, full-stops and spaces at the starts of lines, and spurious fullstops where they are obviously inappropriate. After doing that we arrive at a version of the text in which three of the lines appear in exactly the same form in at least two other versions of the text.

It ought, then, to be possible to examine each line of text, one by one, in each version of our text, and to take the version of the line that occurs the most often. If we try this we run into a snag: in the lower quality texts there are systemic errors, so that sometimes the majority is not always best. How can we avoid this difficulty?

5. The Algorithm

Instead of making a pure majority text, we choose by inspection the most promising version. We then process this text, along with all of the other versions, and produce a set of change instructions (actually a Unix "patch" file) that will mark

only places where the majority of *other* versions deviate from the best edition. The algorithm can be tweaked to require a replacement line to appear in *all* of the other versions before selecting it. The method is as follows:

1. Minimally pre-process each version of the text to remove trailing spaces, misplaced punctuation and spaces before colons or semi-colons. This step must not damage, for example, a comma after an abbreviation, and should therefore be cautious.
2. Choose version that appears most promising at this stage, by inspection.
3. Run the Majority Edition creation tool on all of the versions, with the most promising listed first.
4. The result is a patch file; both graphical and text-based tools to select patches are widely available and are also included in open source and libre text editing tools (even editors such as *vim* and *emacs*), and generally show the context for each change. The author has found that there are typically anywhere from a few hundred to a couple of thousand changes in a densely-printed 500-page reference work.

This sequence begs the important question of producing the majority edition. The present version is written in a subset of Perl and uses the `String::Similarity` module to give a similarity metric between two strings of zero (totally different), one (identical) or, more often, some intermediate value representing the calculated edit distance. The algorithm used is given as a reference in the Perl module as that of [2] and [3].

The script proceeds as follows:

1. Opens all input files, converting to Unicode UTF-8 as needed;
2. Uses a sliding window for each file to match up corresponding non-blank lines of text, first searching ahead as needed for identical lines and then falling back to "similar" lines;
3. If for a given line all files have identical characters, choose that;
4. If all but one file have the same text, use that (optional step);
5. If the primary version is sufficiently outnumbered, use the majority version.

A somewhat reduced version of the production script is given in an appendix to this paper. In addition to the operations already described, the version given here also rejects lines that contain too much improbable punctuation, such as less-than signs or curly braces or the caret by itself, that are commonly introduced by OCR recognition problems and only rarely, if ever, occur in historical texts. When curly braces do occur the OCR programs usually fail to recognize them as such. Another common problem is recognition of a semicolon as a lower-case "j" usually as a word by itself because of the so-called French spacing used in many

nineteenth-century books. The script therefore rejects a suggested change if it would eliminate semicolons. French spacing

6. A Variation: Word-at-a-time Mode

The imaginative reader might be wondering, why stop at lines? Why not process individual words or even characters? The script can in fact do a word-based majority edition but experience shows that with particularly poor text this can result in a lot of incorrect changes. The reason is that the OCR process often runs words together or adds spurious spaces, and the resulting word fragments are in that situation usually too short to be paired reliably with corresponding words in other files. This is, however, a possible area for future work.

It should also be noted that the script has a facility for recognising explicit page breaks as a mechanism to limit the scope of any problems. An input line starting with an @-sign is treated as an explicit page break and is assumed to contain any running header or foliation. The conversion scripts check that page numbers increase monotonically and that none are skipped, after allowing for unnumbered pages; this is especially useful when dealing with OCR output that might occasionally miss a page.

7. Converting to XML

The archive.org text files are often marked up very lightly in HTML, with a tag at the start and end of an entire book and plain text between, using blank lines for paragraph breaks. The author wrote additional scripts to handle recognition of sections or chapters, dictionary entries (where appropriate), page breaks, footnotes, footnote references, cross references and other corndoodles. The process has been to create the majority texts once and then edit them by hand (for example, to correct OCR errors) and then re-run the conversion scripts as needed using the **Make** program; keeping the majority edition scripts under some form of revision control is highly recommended in this case. The actual makefile the author uses for one project with these scripts is included as an appendix to this paper.

Since footnotes are asynchronous to the document structure it turns out to be simplest to identify them before converting to XML fully. A common idiom, invented by this author and independently (and probably earlier) by many others, is to convert unwanted XML tags into a text marker such as ##381## for the three hundred and eighty-first tag in the document,, then to process the unmarked text, process the footnotes, then restore the hidden tags. This allows footnotes to be marked up explicitly with <fn> elements by hand in the output of the majority edition texts, where the scripts fail to detect them. Another possible area of future work would be to improve footnote detection. Where footnotes had two columns in the printed book the OCR has sometimes made a single long thin

column, which is easy to process automatically, and sometimes made long lines by joining adjacent lines of text from the two columns, which must for the most part be separated by hand. But what software hath joined, no human should have to cast asunder.

By the end of pre-XML-tool processing, disparate parts of hyphenated words have been reunited; footnotes have been marked up, including the insertion of links to sources; other text features may have been marked, such as dates or titles of cited works; each step in the process improves the markup or removes errors inserted from earlier steps. The resulting XML document is then available for analysis. For Web purposes, an XSLT transformation might generate dozens or even tens of thousands of separate HTML documents, one per dictionary entry or chapter, and a separate process then adds cross-document site-wide links to those individual files.

8. Evaluation and Further Work

There is no question that the resulting text files will (and do) contain many fewer errors as a result of this process. The result is far from perfect and for a single short book it might be better to proof-read in a more traditional manner. For works such as a 32-volume Dictionary of Biography manual proof-reading is not an option and algorithmic improvements such as the process described here are required.

Future planned work is to make use of the XML markup, once generated, to improve error correction as well as to enhance the overall usefulness of the texts. An example of markup-specific correction already in use is that footnotes, once detected, are processed specially: proper names in them are usually citations and can be linked to the appropriate text automatically in many cases; because the text of footnotes was printed with a smaller typeface there are different punctuation errors, some of which can be corrected automatically.

The author is hoping to apply contextual stochastic analysis to identify errors in texts and to locate related texts: for example, a paragraph that mentions the name of an Oxford college is more likely to be referring to the City of Oxford than to the town of Orford even if the word “Oxford” does not otherwise appear.

9. Conclusion

Documents in XML have to get into XML somehow; that can be by creating them in XML or it can be through conversion. This paper has discussed a technique to improve the quality of XML documents acquired through OCR, and has also shown more of a context in which the technique can be used effectively.

Although none of the techniques in this paper are particularly new (most go back to the 1960s), what is new is the application of these techniques in combina-

tion to texts that have been scanned multiple times and that have sufficient metadata to identify the exact edition used with precision.

A. Majority Edition Script

The script that generates the majority edition is actually fairly simple. It's written in Perl to make use of the `String::Similarity` module, although it could equally well be written in another language. Performance was adequate on a reasonable system even several years ago, taking only a minute or two to run on a five-hundred-page book with half a dozen versions. The version given here works but the production version has some improvements and additional checks that were omitted to save space. See `http://words.fromoldbooks.org/xml/` for the full version.

```
#! /usr/bin/perl -w
use strict;
use String::Similarity;
binmode STDOUT, ":utf8"; # allow Unicode output

my $minmax = 3; # must be 3 texts the same to use a change

my %files;
my %furthestlinereached;
my $nfiles = 0;

{
    local $/; # "slurp" mode, read whole file at once
    # inside a {...} block so it doesn't affect the rest of the code

    foreach (@ARGV) {
        open(IN, $_) or die "can't open $_: $!";
        binmode IN, ":utf8"; # so we can read Unicode
        my $text = <IN>; # slurp mode: $text is the whole file
        close IN;
        # make an array of lines in the input file:
        @{$files{$nfiles}} = split('\n', $text);
        $furthestlinereached{$nfiles} = -1;
        ++$nfiles;
    }
}

sub latesttext($)
{
    my ($n) = @_;
    return ${$files{$n}}[$furthestlinereached{$n}]
```

```
}

sub quality($)
{
    # returns 0 for unacceptable, 1 for good, 0.5 for OK...

    # internally, use 100 for bad
    my ($line) = @_;
    my $result = 0;

    while ($line =~ m/[\{\}\\~%]/g) {
        # really bad if it has these characters before 1900
        $result += 70; # add 70 for each occurrence
    }
    if ($result >= 100) {
        return 0;
    }

    while ($line =~ m/[&](lt|gt|amp)\;/g) {
        $result += 50; # 50 for each < > or &-sign
    }
    if ($result >= 100) {
        return 0;
    }

    while ($line =~ m/ii/g) { # suggests bad OCR (or Latin)
        $result += 10;
    }

    while ($line =~ m/,,/g) {
        $result += 4;
    }

    while ($line =~ m/\*/g) {
        $result += 7;
    }

    while ($line =~ m/ j /g) {
        $result += 5; # should be a semicolon
    }

    if ($result >= 100) {
        return 0;
    }

    if ($result == 0) {
```

```
            return 1; # OK
        }

        return 1 - ($result / 100);
    }

    sub countchar($$)
    {
        my ($string, $pattern) = @_;
        my $result = 0;

        while ($string =~ m/$pattern/g) {
            ++$result;
        }
        return $result;
    }

    sub improves($$)
    {
        my ($original, $suggestion) = @_;

        if ($original eq $suggestion) {
            return 1; # it's no worse!
        }

        if (countchar($suggestion, ";") < countchar($original, ";")) {
            return 0; # lost one or more semicolons
        }

        if (countchar($suggestion, "\\^") > countchar($original, "\\^")) {
            return 0; # gained ^ (common OCR error)
        }

        return 1;
    }

    # the main processing loop:
    foreach my $i (0 .. $#{$files{0}}) { # for each line in file 0
        $furthestlinereached{0} = $i;
        my $line = ${$files{0}}[$i];

        # blank lines are to be treated specially:
        if ($line =~ m/^\s*$/) {
            print "\n";
            next;
        }
```

```
# explicit page breaks marked with @ at the start of a line
if ($line =~ m/^[@]/) {
    print ${$files{0}}[$i] . "\n";
    next;
}

my %foundlines = ();
push @{$foundlines{latesttext(0)}}, 0;

my $nfound = 1; # we found the line in the first file!
my $nclose = 0; # number of similar lines found
foreach my $f (1 .. $nfiles - 1) {
    foreach my $j ($furthestlinereached{$f} + 1 .. $#{$files{$f}}) {
        # If we get more than 200 lines out of sync, disregard
        # the out-of-sync version until we catch up.
        last if ($j - $furthestlinereached{$f} > 200);

        my $other = ${$files{$f}}[$j];
        my $similarity = 1;
        if ($line eq $other) { # found it in this version
            $furthestlinereached{$f} = $j;
            push @{$foundlines{latesttext($f)}}, $f;
            ++$nfound;
            last; # quit looking in this file
        } elsif ($other !~ m/^\s*$/) { # skip blank lines
            # use editing distance as a measure of similarity:
            $similarity = similarity($line, ${$files{$f}}[$j]);
            if (defined $similarity && $similarity > 0.9) {
                $furthestlinereached{$f} = $j;
                if (quality(latesttext($f)) < 0.8) {
                    last; # Do not want this version
                }
                $nclose++;
                push @{$foundlines{latesttext($f)}}, $f;
                last; # found it, quit looking in this file
            }
        }
    }
} # foreach file

if ($nfound == $nfiles) { # this line the same in all versions
    print ${$files{0}}[$i] . "\n";
} else { # our line was only found to be OK in some files...
    my $max = 0; my $which = 0;
    foreach my $version (keys %foundlines) {
```

```
            if (@{$foundlines{$version}} > $max) {
                $max = @{$foundlines{$version}};
                $which = ${$foundlines{$version}}[0]; # the file number
            }
        }

        # only accept the replacement if it is an improvement
        if ($max >= $minmax && improves(latesttext(0), ▶
latesttext($which))) {
            print ${$files{$which}}[$furthestlinereached{$which}] . "\n";
        } else { # keep the original:
            print latesttext(0) . "\n";
        }
    }
} # foreach line
```

B. Project Makefile

The `makefile` given here is for the Chalmers Dictionary project. When the author runs the *make* program, the entire conversion is run but not published onto the live Web site, to give an opportunity for error checking first.

An advantage of using *make* rather than a procedural shell script is that make is lazy, and if intermediate generated files are up-to-date it knows it doesn't need to rebuild them, potentially saving a lot of time.

If you use make in your own projects, be warned that the file format is fussy and considered spaces and tabs to be different, so you must use a text editor that behaves itself.

```
# Makefile for Chalmers Dictionary

# The filenames script returns a list of matching filenames, or all
# if no pattern is given:
FILES := $(shell ./filenames)

# The default target, "done", depends on the final XML and on the CSS ▶
that's
# produced by Saxon. It doesn't publish to the live site autoamtically.
done: all.xml fixed.xml fn.xml with-sources.xml out/entries.css
        @echo now run ./addlinks.pl and ./publish

# (1) First, identify page breaks
paginated.ok: $(FILES)
        bin/identify-page-breaks.pl $(FILES)
        touch paginated.ok

# (2) pre-process and then identify footnotes
```

```
all.xml: paginated.ok bin/prepare bin/fnpfix.pl
        -cp entries e1
        bin/prepare paginated/* 2> prepare.log | bin/fnpfix.pl > all.xml
        -wc -l errors.txt
        xmllint --noout all.xml

# (3) correct some common OCR errors
fixed.xml fixed.log: all.xml bin/fixup
        time bin/fixup > tmpfixed.xml 2> fixed.log
        xmllint --noout tmpfixed.xml
        mv tmpfixed.xml fixed.xml

# (4) connect footnotes to their references
fn.xml fn.log: fixed.xml bin/fixfootnotes.pl
        bin/fixfootnotes.pl < fixed.xml > fn.xml 2> fn.log
        ls -l fn.log
        xmllint --noout fn.xml || exit 1

# (5) Step (2) already dealt with hyphens, but added soft hyphens;
# This step removes those SHY chars and builds a hyphenation dictionary
hy.xml: fn.xml bin/hyphenate
        bin/hyphenate > hytmp.xml < fn.xml
        xmllint --noout hytmp.xml || exit 1
        mv hytmp.xml hy.xml

# (6) Make bibliographic references be links where possible
# This builds "with-sources.xml", the final XML file
with-sources.xml: hy.xml bin/sources.pl bin/addvocations.pl
        bin/sources.pl < hy.xml | bin/addvocations.pl > with-sources.xml
        xmllint --noout with-sources.xml || exit 1

# (7) make an HTML version of with-sources suitable for the Web site,
# using the fabulous incomparably wonderful Saxon program
out/entries.css: with-sources.xml fixed.xml split.xslt
        xmllint --noout links.xml
        rm -rf out
        time saxonsa-latest with-sources.xml split.xslt 2> xslt.log
        wc -l e1
        wc -l entries
        ls -l xslt.log
        touch done

# Any file listed as a dependency can get removed if Make is interrupted,
# so that you don't end up in an inconsistent state. Make won't delete
# files that are marked as previous:
```

```
.PRECIOUS: bin/sources.pl bin/fixfootnotes.pl bin/prepare bin/fixup \
    split.xslt Makefile filenames
```

Bibliography

[1] Chalmers, Alexander, F.S.A. (Ed.), The General Biographical Dictionary, Containing an historical and critical account of the lives and writings of the most eminent persons in every nation; particularly the British and Irish; from the earliest accounts to the present time, London, 1812–1817. http://words.fromoldbooks.org/Chalmers-Biography/

[2] Myers, Eugene, An O(ND) Difference Algorithm and its Variations, Algorithmica Vol 1 issue 2, 1986, pp. 251-266.

[3] Ukkonen, E., Algorithms for Approximate String Matching, Information and Control Vol. 64, 1985, pp. 100-118.

Checking documents for DTP with the free online service data2check

Mehrshad Zaeri Esfahani
parsQube GmbH
<Mehrschad.Zaeri@parsQube.de>

Hauke Brandes
parsQube GmbH
<hauke.brandes@parsqube.de>

Manuel Montero Pineda
data2type GmbH
<montero@data2type.de>

1. What is data2check?

With the help of the free service "data2check[1]", Word, InDesign and EPUB files can be checked for correctness.

Word documents are checked for compliance with the styles used (paragraph and character styles). In this way, important copy-editing guidelines of a publishing house or a company can be controlled. Furthermore, a document can be examined for Word-specific components making further processing much more difficult (e.g. text boxes, pictures, charts, etc.). InDesign documents can be checked for created paragraph and character styles but also for constructs making the export from InDesign to the EPUB format impossible.

After completion of a check, any errors found can be tracked with the help of comments in the output document. In Addition, the uploaded data is stored as XML files in a database, where they are available at any time.

2. Why do we need such checks?

When we focus on Word, the following initial situation occurs: publishing houses or companies need structured data in XML. However, authors are often not willing or able to create and provide these documents in an XML editor. There is a range of possible solutions for the publishing companies, though all of these are problematic.

One possible workflow is based on the usage of Word documents. The advantage of this method lies simply in the high acceptance among the authors. The big disadvantage is that authors often provide data being not well structured.

[1] http://www.data2check.de

These Word documents have to be expensively prepared in a copy-editing process which is performed either before or after a conversion into XML. In the end, this work has to be paid by the publishing house or the author and it delays the publishing process.

3. What is the technical difficulty to connect DTP and XML?

Normally, document formats for DTP programmes are flat and only contain a few structures. These include first and foremost paragraphs and tables.

The reason for this is that it shall be easily possible to cut out content and to re-insert it elsewhere, without any error messages. Almost all content is displayed in the form of paragraphs which, without any restrictions, may appear in a different order. Also headings, lists, pictures and so on are displayed in paragraphs. From the XML perspective, the only reasonably implemented structures are tables and footnotes.

These flat structures make a conversion into XML formats, such as DocBook, TEI or DITA, difficult since important parts of the semantics lie in the applied styles, which in turn can be used freely in the DTP programmes.

3.1. WordML is also XML, or not?

In technical terms, Word and InDesign are also XML formats. However, the purpose of these formats is to represent everything that was used, for example, in a Word document. In case the WordML file is valid, one can assume that Word is able to open the file and represent its content. But the format says nothing about whether it can be transformed into a publication structure such as DocBook.

As a consequence, when checking a document two layers have to be taken into account. On the one hand, for example, the WordML file must be valid, and on the other hand, the styles must be used in a further abstraction layer in such a way that they can be converted into another structure, as for example DocBook.

From a visual point of view, the following Word example is correct. Nonetheless, a transformation would not be possible here since the heading containing the author name is tagged as heading level 2 and then the book title follows.

```
<w:body>
  <w:p w:rsidR="009145BB" w:rsidRDefault="009145BB" w:rsidP="009145BB">
    <w:pPr>
      <w:pStyle w:val="Heading2"/>
      <w:rPr>
        <w:rFonts w:eastAsia="Times New Roman"/>
        <w:lang w:eastAsia="de-DE"/>
      </w:rPr>
    </w:pPr>
    <w:proofErr w:type="spellStart"/>
```

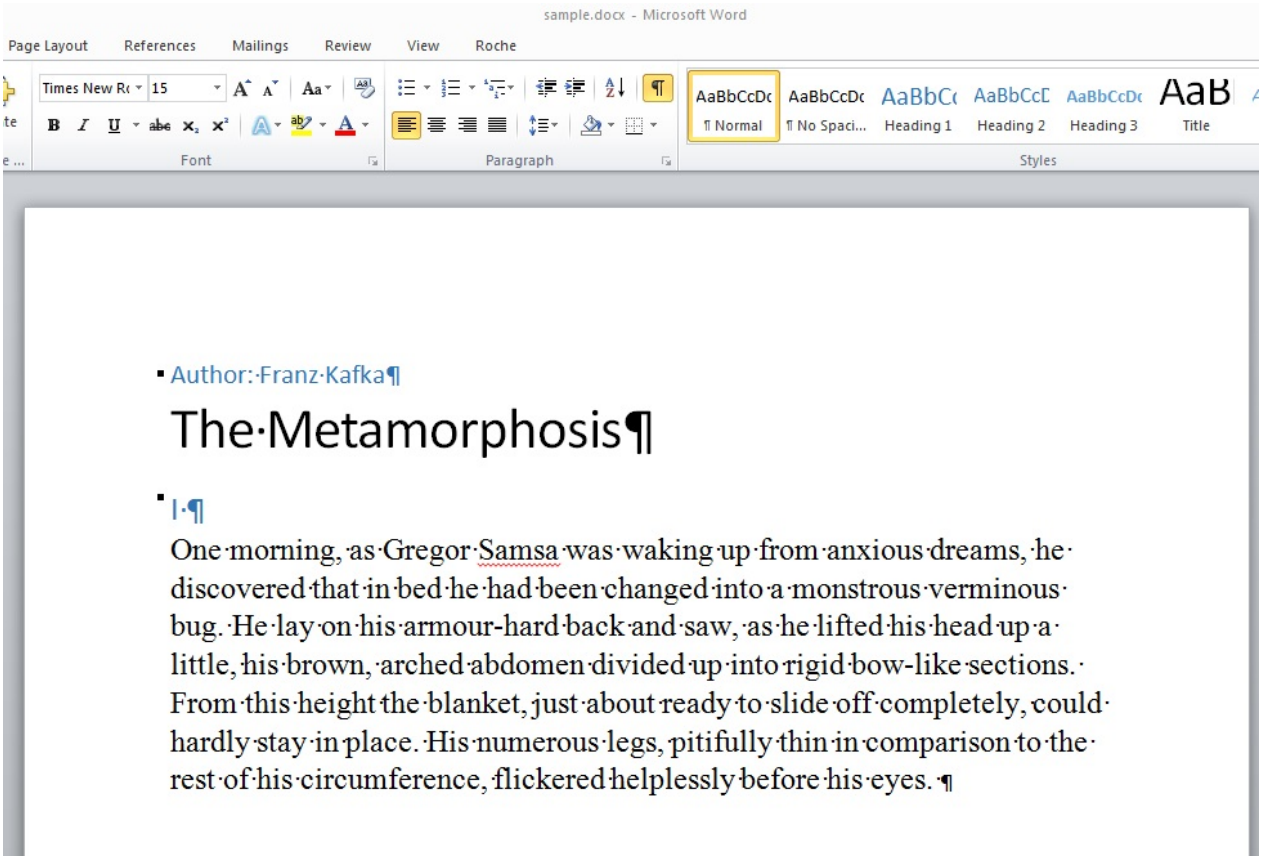


Figure 1. Excerpt from the example Word document

```
      <w:r>
        <w:rPr>
          <w:rFonts w:eastAsia="Times New Roman"/>
          <w:lang w:eastAsia="de-DE"/>
        </w:rPr>
        <w:t>Author</w:t>
      </w:r>
      <w:proofErr w:type="spellEnd"/>
      <w:r>
        <w:rPr>
          <w:rFonts w:eastAsia="Times New Roman"/>
          <w:lang w:eastAsia="de-DE"/>
        </w:rPr>
        <w:t>: Franz Kafka</w:t>
      </w:r>
    </w:p>
    <w:p w:rsidR="009145BB" w:rsidRPr="009145BB" ▶
w:rsidRDefault="009145BB"
     w:rsidP="009145BB">
      <w:pPr>
        <w:pStyle w:val="Title"/>
```

```
      <w:rPr>
        <w:lang w:eastAsia="de-DE"/>
      </w:rPr>
    </w:pPr>
    <w:r>
      <w:rPr>
        <w:lang w:eastAsia="de-DE"/>
      </w:rPr>
      <w:t xml:space="preserve">The </w:t>
    </w:r>
    <w:proofErr w:type="spellStart"/>
    <w:r>
      <w:rPr>
        <w:lang w:eastAsia="de-DE"/>
      </w:rPr>
      <w:t>Metamor</w:t>
    </w:r>
    <w:r>
      <w:rPr>
        <w:lang w:eastAsia="de-DE"/>
      </w:rPr>
      <w:t>phosis</w:t>
    </w:r>
    <w:proofErr w:type="spellEnd"/>
  </w:p>
```

As you can see in the above code snippet, there are several challenges. On the one hand the reference to the actual name of the style is put in a ID-RefID relation. The `w:val` attribute contains the RefID to an ID in another file which, in turn, contains the name of the style: `<w:pStyle w:val="Heading2"/>`.

This aspect alone shows that Word was not designed to validate content via styles. In this context, there is a series of further problems. Some of these are listed in the following:

1. Often, the inline content is fragmented:

```
<w:p w:rsidR="009145BB" w:rsidRPr="009145BB" w:rsidRDefault="009145BB"
     w:rsidP="009145BB">
      <w:pPr>
        <w:pStyle w:val="Title"/>
        <w:rPr>
          <w:lang w:eastAsia="de-DE"/>
        </w:rPr>
      </w:pPr>
      <w:r>
        <w:rPr>
          <w:lang w:eastAsia="de-DE"/>
```

```
      </w:rPr>
      <w:t xml:space="preserve">The </w:t>
    </w:r>
    <w:proofErr w:type="spellStart"/>
    <w:r>
      <w:rPr>
        <w:lang w:eastAsia="de-DE"/>
      </w:rPr>
      <w:t>Metamor</w:t>
    </w:r>
    <w:r>
      <w:rPr>
        <w:lang w:eastAsia="de-DE"/>
      </w:rPr>
      <w:t>phosis</w:t>
    <w:proofErr w:type="spellEnd"/>
  </w:p>
```

Without a reason, the word **Metamorphosis** is split here. This would be the same as if the word is tagged in DocBook as follows:

```
<para><emphasis role="bold">Meta</emphasis>
<emphasis role="bold">morpho</emphasis><emphasis
role="bold">si</emphasis><emphasis role="bold">s</emphasis></para>
```

2. Lists and headings are paragraphs. Indeed, this is very convenient for Word but often very difficult for a transformation.
3. Without using styles, the user can format the Word document wildly, so that it looks like as if it were correct.
4. Tables are often used for indentations, etc.
5. Text boxes can optically indicate content elsewhere as embedded in the WordML file.

4. Which check mechanisms are implemented in data2check?

For the check of the abovementioned second layer, a set of mechanisms is used which aims to enable a later XML export into a document structure. Technically, this layer is implemented by a Schematron process.

Therefore, the following checks are possible on the documents:

1. that they only consist of a fixed set of styles.
2. that the structure, e.g. of the hierarchy of headings, is respected.
3. that texts can be revised automatically via regular expressions.
4. that a start style can be specified.

5. that paragraph styles shall contain only certain character styles.
6. that on one style only a set of certain styles shall follow.

In order to ensure that also non-programmers are able to configure such checks, much time and effort was invested in a GUI which enables these configurations and excludes incorrect operation as far as possible.

The following screenshot shows how a configuration for checking a document can be edited. In the shown tab the settings for the styles used are made.

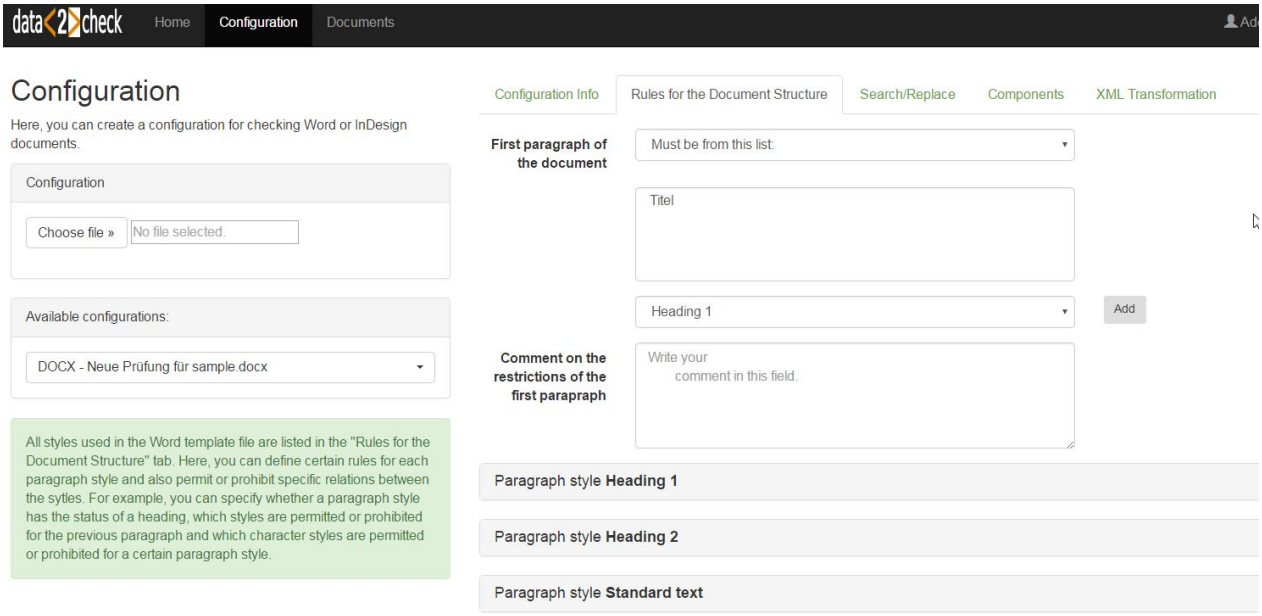


Figure 2. Excerpt from the data2check GUI - Rules for the Document Structure

In the following screenshot you can see which rules can be set up for a paragraph style.

The combination of these simple rules allows a convergence to valid structures in the target format. In practice, these checking processes can be applied in order to:

1. support the copy-editing procedures of publishing companies or service providers.
2. automatically check the CIs of companies.
3. support companies in the technical documentation which are working with Word and want to migrate their data to DITA or DocBook in the future.

In this session, we want to demonstrate the mentioned check mechanisms by means of example documents. Afterwards, we would be pleased to discuss with the audience possible further check routines for data2check.

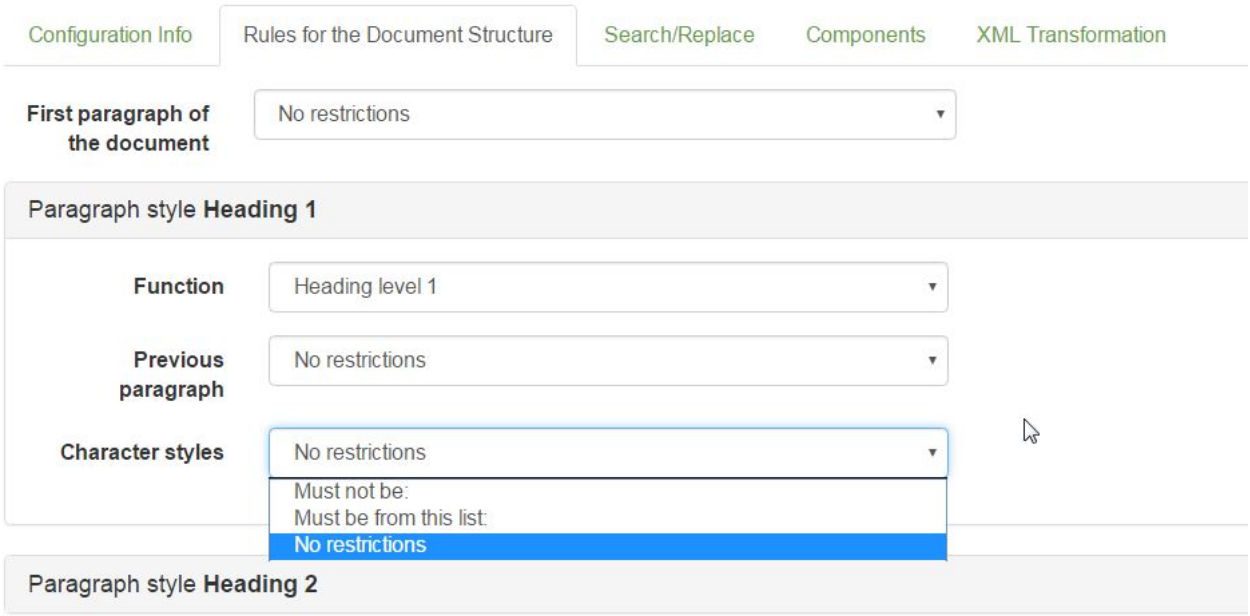


Figure 3. Possible settings for a paragraph style

W3C ITS 2.0 in OASIS XLIFF 2.1

Managing metadata throughout the multilingual content lifecycle

David Filip
ADAPT Centre at Trinity College Dublin
<david.filip@adaptcentre.ie>

Abstract

XLIFF is the XML Localization Interchange File Format. The current OASIS Standard version is XLIFF Version 2.0 [10]. XLIFF Version 2.1 ??? concluded the 1st public review period on 25th November 2016. The major new features added to [12] compared to ??? are the native [6] support and the Advanced Validation feature via NVDL and Schematron. The Advanced Validation feature for XLIFF 2 was first introduced at FEISGILTT 2014 [1] and covered extensively at XML London 2016 [2]. In this paper and presentation, we want to look in detail at the [6] native support feature.

In this paper and XML Prague presentation we will explain in detail about W3C ??? metadata categories support in [12] and which ITS data in XLIFF 2.1 are accessible or not to generic ITS Processors despite the use of the W3C namespace.

Keywords: W3C ITS, XLIFF, Internationalization, I18n, Localization, L10n, metadata, namespaces, mapping, roundtrip, lifecycle, multilingual content

This research was conducted at the ADAPT Research Centre, Trinity College Dublin, Ireland.

The ADAPT Centre is funded under the SFI (Science Foundation Ireland) Research Centres Programme (Grant 13/RC/2106) and is co-funded under the European Regional Development Fund.

1. Introduction

This paper is about managing *Internationalization* metadata throughout the multilingual content lifecycle. Even though corporations and governments routinely need to present the same, equivalent, or comparable content in various languages, *multilingual content* is usually not consumed in more than one language at the same time by the same end user. Typically the target audience consumes the content in their preferred language and if everything works well they don't even

need to be aware that the monolingual content they consume is part of a multilingual content repository or a result of a *Translation*, *Localization*, or cultural adaptation process.

Thus *Multilingualism* is transparent to the end user if implemented properly. To achieve end user transparency the corporations, governments, inter- or extranational agencies need to develop and employ *Internationalization*, *Localization*, and *Translation* capabilities. While *Internationalization* is primarily done on a monolingual content or product, *Localization*, and *Translation* when done at a certain level of maturity - as a repeatable process possibly aspiring to efficiencies of scale and automation - requires a persistent *Bitext* format.

This paper describes an open, standard, and transparent way of managing *Internationalization* metadata in multilingual repositories from seeding them in monolingual source or pivot content through extraction to an open *Bitext* format, manipulating or injecting relevant metadata categories during the *Bitext Roundtrip*, to keeping, archiving or throwing away the metadata that arrived processed in the target content.

2. Lay of the land

The foundational Internationalization Standard is of course [8] along with some related Unicode Annexes. But in this paper we are taking the Unicode support for granted and will be looking at the domain standards W3C ITS and OASIS XLIFF that are the open standards relevant for covering the industry process areas outlined in the Introduction.

For a long time, XML has been another unchallenged foundation of the multilingual content interoperability and hence practically all Localization and Internationalization standards started as or became at some point XML vocabularies. Paramount industry wisdom is stored in data models that had been developed over decades as XML vocabularies at OASIS, W3C, LISA (RIP) and elsewhere. Although ITS is based on *abstract metadata categories*, [5] had only provided specific implementable recommendation for XML. The simple yet ingenious idea of ITS is to provide a reusable namespace that can be injected into existing formats. Although the notion of a namespace is not confined to XML, again ITS 1.0 was only specifically injectable into XML vocabularies.

[6] provides local and global methods for metadata storage not only in XML but also in HTML 5, it also looked at mapping into non-XML formats such as [4], albeit in a non-normative way. Because native HTML does not support the notion of namespaces, ITS 2.0 has to use attributes that are prefixed with the string `its-` for the purpose of being recognized as an HTML 5 module. [6] also introduced many new metadata categories compared with [5]. ITS 1.0 only looked at metadata in source content that would somehow help inform the Internationalization and Localization processes down the line. ITS 2.0 brought brand new and some-

times complex metadata categories that contain information produced during the localization processes or during the language service transformations that are necessary to produce target content and are typically facilitated by Bitext. This naturally led to a non-normative mapping of [6] to ??? and to [10]. Thus ITS 2.0 became a very useful extension to XLIFF. One of the main reasons why [10] is not backwards compatible with [9] is that the OASIS XLIFF TC and the wider stakeholder community wanted to create XLIFF 2 with a modularized data model. [10] has a small non-negotiable core but at the same time it brings 8 namespace based modules for advanced functionality. The modular and extensible design aims at easy production of "dot" revisions or releases of the standard. XLIFF 2.0 was intended as the first in the future family of backwards compatible XLIFF 2 standards that will share the maximally interoperable core (as well as successful modules surviving from 2.0). XLIFF 2 makes a distinction between modules and extensions. While module features are optional, *Conformant XLIFF Agents* are bound by an absolute prohibition to delete module based metadata, whereas deletion of extension based data is discouraged but not prohibited. The *ITS Module* is the biggest feature that was requested by the industry community and approved by the TC for specification as part of ???.

The easiest metadata category to explain the idea of ITS is *Translate*; this is simply a boolean flag that can be used to indicate Translatability or not in source content.

Example 1. Translate expressed locally in HTML

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset=utf-8>
    <title>Translate flag test: Default</title>
  </head>
  <body>
    <p>The <span translate=no>World Wide Web Consortium</span> is
      making the World Wide Web worldwide!</p>
  </body>
</html>
```

Example 2. Translate expressed locally in XML

```
<messages its:version="2.0" xmlns:its="http://www.w3.org/2005/11/its">
  <msg num="123">Click Resume Button on Status Display or
    <panelmsg its:translate="no"
      >CONTINUE</panelmsg> Button on printer panel</msg>
</messages>
```

Since it is not always practically possible to create local annotations, or the given source format or XML vocabulary has elements or attributes with clear semantics

with regards to some Internationalization data categories such as Translate, in most cases, ITS 2.0 also defines a way to express a given data category globally.

Example 3. Translate expressed globally in XML

```
<its:rules version="2.0" xmlns:its="http://www.w3.org/2005/11/its">
  <its:translateRule translate="no" selector="//code"/>
</its:rules>
```

In the above the global `its:translateRule` indicates that the content of `<code>` elements is not to be translated.

XLIFF 2 Core has its own native local method how to express Translatability, it uses the `xlf:translate` attribute. Here and henceforth the prefix `xlf:` indicates this OASIS namespace `urn:oasis:names:tc:xliff:document:2.0`. Because XLIFF is the Bitext format that is used to manage the content structure during the service roundtrip in a source format agnostic way, XLIFF needs to make a hard distinction between the structural and the inline data. We know the structural vs inline distinction from many XML vocabularies and HTML. Some typical structural elements are Docbook `<section>` or `<para>` as well as HTML `<p>`. This is how XLIFF 2 will encode non-Translatability of a structural element:

Example 4. XLIFF Core @translate on a structural leaf element

```
<unit id='1' translate="yes">
        <segment>
          <source>Translatable text</source>
        </segment>
</unit>
<unit id='2' translate="no">
        <segment>
          <source>Non-translatable text</source>
        </segment>
</unit>
```

The above could be an *Extraction* of the following HTML snippet:

```
<p translate='yes'>Translatable text</p>
 <p translate='no'>Non-translatable text</p>
```

The same snipped could be also represented like this:

Example 5. XLIFF representing ITS Translate by Extraction behavior w/o explicit metadata

```
<unit id='1'>
        <segment>
          <source>Translatable text</source>
```

```
        </segment>
    </unit>
```

However, it is quite likely that the non-translatable structural elements could provide the translators with some critical context information. Hence the non-extraction behavior can only be recommended if the *Extracting Agent* human or machine can make the call if there is or isn't some sort of contextual or linguistic relationship.

In case of the Translate metadata category being expressed inline, XLIFF has to use its *Translate Annotation*:

Example 6. XLIFF Core @translate used inline

```
<unit id='1'>
  <segment>
    <source>Text
      <pc id='1'/><mrk id='m1' translate='no'>Code</mrk></pc></source>
  </segment>
</unit>
```

The above could be an Extraction of the following HTML snippet:

```
<p>Text <code translate='no'>Code</code></p>
```

Also inline, there is an option to "hide" the non-translatable content like this:

Example 7. XLIFF representing ITS Translate by Extraction behavior w/o explicit metadata

```
<unit id='1'>
    <segment>
        <source>Text <ph id='1'/></source>
    </segment>
</unit>
```

Again not displaying of the non-translatable content can be detrimental to the process, as both human and machine translation agents would produce unsuitable translations in case there is some linguistic relationship between the displayed translatable text and the content hidden by the placeholder code.

Because XLIFF has its own native method of expressing translatability, generic ITS decorators could not succeed. ITS processors can however access the translatability information within XLIFF using the following global rule:

Example 8. ITS global rule to detect translatability in XLIFF

```
<its:rules version="2.0" queryLanguage="xpath">
    <!-- Rules for Translate -->
```

```
        <its:translateRule selector="//xlf:*[@translate='no']"
                           translate='no'/>
        <its:translateRule selector="//xlf:*[@translate='yes']"
                           translate='yes'/>
    </its:rules>
```

In the section The nitty gritty we will explain why this rule will sometimes fail and how to address the fail cases.

3. ITS metadata categories and their purpose

In this section we will first have a look at the 19 [6] metadata categories from the functional point of view and later on from the XLIFF representation point of view.

3.1. Source metadata that inform Extraction behavior

Translate[1], Locale Filter[2], External Resource[3], and Elements Within Text[4] are actually all methods to inform Extraction behavior. As such, these are quite important for creation of Bitext management formats such as XLIFF but don't need to be necessarily explicitly expressed within those formats. See the detailed discussion of Translate above in Lay of the land and also below From the XLIFF point of view for the other Extraction informing datacats.

Locale Filter[5] is a method that can make content conditional based on a target locale. This can be used on source as well as target content. For instance, legal content will be different for the locales `fr-FR` and `fr-CA`.

External Resource[6] indicates an external usually non-text part of content that also needs to be changed in order to produce fully adapted target content. Typically this points to external graphics, scripts or binaries.

Many formats don't contain all the localizable text in a linear structure. Elements Within Text[7] helps extractors properly handle and not lose context for text placed in footnotes, endnotes, alt, or contextual hint text etc; on the other hand it can place externally located text within a linear sequence that makes sense for human consumers, but where it does not appear in the native environment.

[1] http://www.w3.org/TR/its20/#trans-datacat
[2] http://www.w3.org/TR/its20/#LocaleFilter
[3] http://www.w3.org/TR/its20/#externalresource
[4] http://www.w3.org/TR/its20/#elements-within-text
[5] http://www.w3.org/TR/its20/#LocaleFilter
[6] http://www.w3.org/TR/its20/#externalresource
[7] http://www.w3.org/TR/its20/#elements-within-text

3.2. Other metadata that inform localization behavior

Language Information[8] uses the [3] data model via `xml:lang` to indicate the natural language of content. This is obviously very useful in case you want to source translations or even just render the content with proper locale specifics.

Target Pointer[9] is used to lessen the pain when working with multilingual documents. In a document to become multilingual such as packaging desktop publishing file, certain areas are designated to hold translations in target languages. It is very bad idea to try and use multilingual documents or spreadsheets to actually manage versions of multilingual content. Such formats should be only used when all target content has been produced via some sort of a Bitext management roundtrip.

Localization Note[10] is basically a free text field with a human readable localization instruction/warning or advice. Although a Localization Note can contain any type of localization information it is not advisable to use it to prescribe Extraction behavior, except perhaps as a preliminary step before using one of the interoperable datacats that inform Extraction.

Directionality[11] has quite a profound Internationalization impact, it let's renderers decide at the protocol level (as opposed to the plain text or script level) whether the content is to be displayed left to right (LTR - Latin script default) or right to left (RTL - Arabic or Hebrew script default). But the Unicode Bidirectional Algorithm [7]as well as directionality provisions in HTML and many XML vocabularies changed since 2012/2013, so the ITS 2.0 specification text is actually not very helpful here. This obviously doesn't affect the importance of the abstract data category and of having proper display behavior for bidirectional content.

Preserve Space[12] indicates via `xml:space` whether or not whitespace characters are significant. If whitespace is significant in source content it is usually significant also in the target content, this is more often then not an internal property of the content format, but it's important to keep this characteristics through transformation pipelines. The danger that this category is trying to prevent is the loss of significant whitespace characters that could not be recovered.

ID Value[13] indicates via `xml:id` a globally unique identifier that should be preserved during translation and localization transformations mainly for the purposes of reimport of target content to all the right places in the native environment.

[8] http://www.w3.org/TR/its20/#language-information
[9] http://www.w3.org/TR/its20/#target-pointer
[10] http://www.w3.org/TR/its20/#locNote-datacat
[11] http://www.w3.org/TR/its20/#directionality
[12] http://www.w3.org/TR/its20/#preservespace
[13] http://www.w3.org/TR/its20/#idvalue

Allowed Characters[14] and Storage Size[15] are basically a way how to inform the Localization providers about certain system limitations that happen to apply to the source but are also expected to be fulfilled by the target. Often this is used to avoid issues from using localized content in insufficiently internationalized environments, be it by sticking to the rule or by manual post-processing if the source limitation cannot be possibly fulfilled by the target in certain locales. For instance a legacy system is ASCII only and all Localizations are required to keep to this restriction (which is impossible for non-Latin script based languages and isn't great for most Latin script based languages except for English). Localizations might need to be held in certain database fields that happen to impose a Storage Size restriction.

3.3. Subject Matter related datacats

Terminology[16] can simply indicate words or multi-word expressions as terms or non-terms. This is how the category worked in ITS 1.0. In ITS 2.0, Terminology can be more useful by pointing to definitions or indicating a confidence score, which is especially useful in cases the Terminology entry was seeded automatically. Terminology doesn't belong exclusively here. Together with Text Analysis it can be actually injected into the content during any stage of the lifecycle and is not limited to source. However, it is very important for the localization process, human or machine driven, to have Terminology annotated be it even only the simple Boolean flag.

Text Analysis[17] is a sister category to Terminology that is new in ITS 2.0. It is intended to hold mostly automatically sourced (possibly semi-supervised) entity disambiguation information. This can be useful for translators and reviewers but can also enrich reading experience in purely monolingual settings.

Domain[18] can be used to indicate content topic, specialization or subject matter focus that is required to produce certain translations. This can be for instance used to select a suitably specialized MT engine, such as one trained on an automotive bilingual corpus in case an automotive domain is indicated or. In another use case, a language service provider will use a sworn translator and require in country legal subject matter review in case the domain was indicated as legal. Although ITS data categories are defined independently and don't have implementation dependencies, Domain information is well suited for usage together with the Terminology and Text Analysis datacats.

[14] http://www.w3.org/TR/its20/#allowedchars
[15] http://www.w3.org/TR/its20/#storagesize
[16] http://www.w3.org/TR/its20/#terminology
[17] http://www.w3.org/TR/its20/#textanalysis
[18]

3.4. Metadata that are produced during or by localization transformations of content

It might seem that this type of metadata is completely new in ???, since [5] concentrated almost exclusively on source metadata. However, as mentioned above, Terminology - that was present in already in ITS 1.0 - can be injected at any point and is not confined to source. Also Directionality is a characteristics of source as well as target and has profound importance during the roundtrip. This was however not the focus in ITS 1.0.

MT Confidence[19], Localization Quality Issue[20], Localization Quality Rating[21], and Provenance[22] - all new categories in ITS 2.0 - can be only produced during Localization transformations; specifically, during Machine Translation, during a review or quality assurance process, during or immediately after a manual or automated translation or revision.

MT Confidence[23] gives a simple score between 0 and 1 that encodes the automated translation system's internal confidence that the produced translation is correct. This score isn't interoperable but can be used in single engine scenarios for instance to color code the translations for readers or post-editors. It can also be used for storing the data for several engines and running comparative studies to make the score interoperable first in specific environments and later on maybe generally.

Localization Quality Issue[24] contains a taxonomy of possible Translation and Localization errors that can be applied in annotations of arbitrary content spans. The taxonomy ensures that this information can be exchanged among various Localization roundtrip agents. Although this mark up is typically introduced in a Bitext environment on target spans, marking up source isn't exclude and can be very practical, especially when implementing the feedback or even reporting a source issue. Importantly, the issues and their descriptions can be Extracted into target content and consumed by monolingual reviewers in the native environment.

Localization Quality Rating[25] is again a simple score that gives a percentage indicating the quality of any portion of content. This score is obviously only interoperable within an indicated Localization Quality Rating system or metrics. Typically flawless quality is considered 100 % and various issue rates per translated

[19] http://www.w3.org/TR/its20/#mtconfidence
[20] http://www.w3.org/TR/its20/#lqissue
[21] http://www.w3.org/TR/its20/#lqrating
[22] http://www.w3.org/TR/its20/#provenance
[23] http://www.w3.org/TR/its20/#mtconfidence
[24] http://www.w3.org/TR/its20/#lqissue
[25] http://www.w3.org/TR/its20/#lqrating

volume would strike down percentages, possibly dropping under an acceptance threshold that can be also specified.

Provenance[26] in ITS is strictly specialized to indicate only translation and revision agents. Agents can be organizations, people or tools or described by combinations of those. For instance, Provenance can indicate that the Reviser John Doe from ACME Language Quality Assurance Inc. produced a content revision with the Perfect Cloud Revision Tool.

3.5. ITS metadata categories from the XLIFF representation point of view

When ITS is becoming an XLIFF module, the most important point of view is whether or not any of the data categories already exist or are partially present in XLIFF. Fully overlapping categories need to be mapped. Fully absent categories are simply implemented from scratch using the ITS provisions. Most challenging are the cases of partial overlap. Several datacats were not implemented for various reasons explained below.

3.5.1. Already in

The ITS data categories that already were available in [10] are Preserve Space[27], Translate[28], and External Resource[29].

We've already covered how Translate is represented in XLIFF 2 back in Lay of the land.

Both ITS and XLIFF do use the `xml:space` to represent the Preserve Space[30] behavior. However, XLIFF 2 prohibits setting of `xml:space` lower than on the `<unit>` element. The reason being that `xml:space` set lower would not evaluate properly on XLIFF inline pseud-spans. Thus XLIFF cannot explicitly express mixed Preserve space behavior inline. It can however normalize all inline content spans that had `xml:space="default"` and set `xml:space="preserve"` on the ancestor `<unit>`. Thus, XLIFF 2 can express the same Preserve Space behavior as ITS, despite the pseudo-spans complication.

External Resource[31] can be extracted into XLIFF 2 using the [10] Resource Data Module[32], which is actually more expressive and requires the indication of the media type for each external resource. XLIFF Extractors have the extra onus to check and set the media type when extracting the External Resource information.

[26] http://www.w3.org/TR/its20/#provenance
[27] http://www.w3.org/TR/its20/#preservespace
[28] http://www.w3.org/TR/its20/#trans-datacat
[29] http://www.w3.org/TR/its20/#externalresource
[30] http://www.w3.org/TR/its20/#preservespace
[31] http://www.w3.org/TR/its20/#externalresource
[32] http://docs.oasis-open.org/xliff/xliff-core/v2.0/os/xliff-core-v2.0-os.html#resourceData_module

While ITS Processors can be pointed to the external resources, they don't need to care for the extra media type info that is not required by the ITS datacat.

Localization Note[33] was listed in ??? as another data category that is fully available through XLIFF. However, it transpired based on the issue #5[34] raised by Yves Savourel that there are some subtle differences in scope between the ITS Localization Note and the XLIFF Core `<note`. Although it is possible to roundtrip Localization Note data in XLIFF 2, it is strictly speaking not accessible by ITS Processors in the exactly same way as by XLIFF Agents. The culprit here is inheritance. Inheritance is one of the basic principles of ITS data categories and so the ITS Localization Note information actually applies to all children of the node where it was specified. XLIFF Notes can apply to high level immutable structural elements `<file>`, `<group>`, and `<unit>` and from the business point of view the Notes do apply to their whole content, nevertheless the Notes content is isolated in a wrapper that is a sibling to the structural payload children and no defaults or inheritance are defined for the structural descendants of the structural host. Simply speaking if a Docbook `<section>` elements hosts a Localization Note, the information will be retrieved by ITS Processors from all of its descendants, while the `<group>` element based Note that will be created during extraction via the mapping will not be imposed on the `<group>` element's descendants. Although from the business point of view, the information does apply to all of the nested content, the Note information actually does not get inherited in the strict XML sense.

Thus, the Localization Note will move to the Partial overlap data categories in all upcoming [12] publications.

3.5.2. Implemented from scratch

Allowed Characters[35], Domain[36], Locale Filter[37], Localization Quality Issue[38], Localization Quality Rating[39], Text Analysis[40] are fully defined within the [12] ITS Module.

Each of the above defines it's own custom annotation to be applicable on XLIFF Inline Content. Application on structural levels is straightforward.

Localization Quality Issue[41] and Localization Quality Rating[42] are injected during the XLIFF roundtrip and can be Extracted to the target content in the native environment if supported by the Merger (with full Extractor knowledge).

33

34 https://issues.oasis-open.org/browse/XLIFF-5

35 http://www.w3.org/TR/its20/#allowedchars

36

37 http://www.w3.org/TR/its20/#LocaleFilter

38 http://www.w3.org/TR/its20/#lqissue

39 http://www.w3.org/TR/its20/#lqrating

40 http://www.w3.org/TR/its20/#textanalysis

Locale Filter[43] is also listed under Not represented. It shouldn't be used within XLIFF Documents that already have specified the target language. `xlf:trgLang` is optional if there are no `<xlf:target>` elements in the XLIFF Document. Corporate content owners participating in XLIFF 2 development expressed the requirement to be able to produce preliminary XLIFF Documents with only source content and the Locale Filter metadata that can be further processed by Localization Service Providers to create multiple XLIFF Documents with the target language set and the Locale Filter metadata fully consumed (no longer present) by inclusion of only the relevant source content.

The remaining datacats work just smoothly as intended in [6] and don't pose any specific implementation challenges.

3.5.3. Partial overlap

Language Information[44], MT Confidence[45], Provenance[46], Terminology[47], and Storage Size[48] have partial overlap with XLIFF 2.0 features.

Language Information[49] used on structural elements is fully supported by XLIFF Core `xlf:srcLang` and `xlf:trgLang` attributes. The culprit is in usage of foreign language spans inline within another source or target language. XLIFF didn't have provisions to handle this use case because the `xml` namespace (and hence `xml:lang`) is prohibited within the XLIFF inline data model. This is because `xml:lang` could not properly interact with XLIFF Core pseudo-spans. `itsm:lang` was defined in [11] but this is now gone since [12] now operates with the W3C `its` namespace that doesn't have its own language information attribute and reuses `xml:lang`.

MT Confidence[50] has an overlap with the XLIFF 2.0 Translation Candidates Module[51]. Explicit usage of ITS based MT Confidence on XLIFF Core is problematic as it only makes sense to be used on unmodified MT suggestions and this cannot be reliably discerned within XLIFF Core. MT Enrichers can be mandated to only use MT Confidence with unmodified MT suggestions. However, Core only Modifiers cannot be mandated to delete module based data after a human intervention. ITS Processors unfortunately cannot access the MT Confidence information contained in `mtc:matchQuality` (`mtc:` inicates

[41] http://www.w3.org/TR/its20/#lqissue
[42] http://www.w3.org/TR/its20/#lqrating
[43] http://www.w3.org/TR/its20/#LocaleFilter
[44] http://www.w3.org/TR/its20/#language-information
[45] http://www.w3.org/TR/its20/#mtconfidence
[46] http://www.w3.org/TR/its20/#provenance
[47] http://www.w3.org/TR/its20/#terminology
[48] http://www.w3.org/TR/its20/#storagesize
[49] http://www.w3.org/TR/its20/#language-information
[50] http://www.w3.org/TR/its20/#mtconfidence
[51] http://docs.oasis-open.org/xliff/xliff-core/v2.0/os/xliff-core-v2.0-os.html#candidates

`urn:oasis:names:tc:xliff:matches:2.0` which is the Translation Candidates module namespace) because ITS does not specify a gloal pointer for MT Confidence. This is where ITS is still in need of change or at least an extension.

Provenance[52] has an overlap with the Change Tracking Module. Here the situation is particularly convoluted since XLIFF 2.1 deprecates the `urn:oasis:names:tc:xliff:changetracking:2.0` namespace and replaces it with the `urn:oasis:names:tc:xliff:changetracking:2.1` because of significant changes in this module since XLIFF 2.0. In XLIFF Core, Provenance works fine as if from scratch; however, in the Change Tracking Module, Provenance metadata needs to interact in an interoperable way with some of its native attributes.

Terminology[53] overlaps with XLIFF Core Term Annotation[54]. This was relatively the easiest case of partial overlap, since the Core Annotation could be complemented with an additional ITS Annotation to cover additional ITS based parts of the Terminology information. ITS Terminology and XLIFF Term also nicely interacts with the XLIFF 2.0 Glossary Module[55] which easily maps to and from TBX Basic.

Storage Size[56] has been listed as a case of partial overlap, since this data category can be expressed as an extension of the XLIFF 2.0 Size and Length Restriction Module[57] (SLR). The specifics of the extension have not been given due to lack of industry interest. Nevertheless, ??? states that Storage Size has to be implemented (if at all) using the Size and Length Restriction Module, which is a readily extensible framework for giving all sorts of "fitting somewhere" restrictions; not only size and length, but also for instance custom display areas filled by letters in specific fonts and sizes, even volume or weight when needed. ITS Storage Size is simply a special case of a SLR based storage restriction.

3.5.4. Not represented

Directionality[58], Elements Within Text[59], ID Value[60], Locale Filter[61], Target Pointer[62] are listed as not represented datacats in [12].

Elements Within Text[63] is fully consumed by Extractor behavior. Properly extracted information will either result in formation of XLIFF Core Sub-flows[64] or

[52] http://www.w3.org/TR/its20/#provenance
[53] http://www.w3.org/TR/its20/#terminology
[54] http://docs.oasis-open.org/xliff/xliff-core/v2.0/os/xliff-core-v2.0-os.html#termAnnotation
[55] http://docs.oasis-open.org/xliff/xliff-core/v2.0/os/xliff-core-v2.0-os.html#glossary-module
[56] http://www.w3.org/TR/its20/#storagesize
[57] http://docs.oasis-open.org/xliff/xliff-core/v2.0/os/xliff-core-v2.0-os.html#size_restriction_module
[58] http://www.w3.org/TR/its20/#directionality
[59] http://www.w3.org/TR/its20/#elements-within-text
[60] http://www.w3.org/TR/its20/#idvalue
[61] http://www.w3.org/TR/its20/#LocaleFilter
[62] http://www.w3.org/TR/its20/#target-pointer

a standard linear order of XLIFF Core segments and units. The metadata will be however not represented in XLIFF Documents and the correct target creation behavior will be only accessible to a Merger that has full Extractor knowledge[65].

It is preferable that Locale Filter[66] is fully consumed by Extraction and the metadata doesn't need to be represented within XLIFF Documents. See however, Implemented from scratch[67] for an exception.

Target Pointer[68] information can form a part of the Extractor / Merger knowledge that is required to populate the native multilingual document, there are however no provisions to represent this in XLIFF Documents. On the other hand XLIFF as a Bitext format is a kind of multilingual document and there is a fairly simple ITS rule (within the ITS Module Schematron schema[69]) that lets generic ITS processors to parse XLIFF 2 for target content.

XLIFF Core doesn't use `xml:id`, all XLIFF ids are of the type `xs:NMTOKEN` and uniqueness scopes are defined at several separate levels. Thus the ID Value[70] information is fully consumed by the Extraction / Merge behavior.

XLIFF Core has it's own fully fledged Directionality[71] capability. So in a sense the abstract datacat should count as "already in", nevertheless the XLIFF Core Directionality[72] doesn't need mapped to the specific ITS 2.0 Directionality provisions, as explained in Other metadata that inform localization behavior[73].

4. The nitty gritty

In the discussion so far, we have seen that there is a fairly good semantic match between ITS and XLIFF. However, there is a couple of principal challenges that are not at all easy to address properly. The solutions adopted during the resolution of the 1st Public Review Comments [11], especially the issue #9[74], were driven by the intention to achieve maximum possible interoperability and making most of the XLIFF-encoded ITS metadata accessible to generic ITS processors without the need to implement specific XLIFF Agent conformance requirements.

The ITS Module namespace that was originally used to encode new ITS metadata within XLIFF was `urn:oasis:names:tc:xliff:itsm:2.1`. This has been replaced by the original W3C namespace `https://www.w3.org/2005/11/its/` for

[63] http://www.w3.org/TR/its20/#elements-within-text
[64] http://docs.oasis-open.org/xliff/xliff-core/v2.0/os/xliff-core-v2.0-os.html#subflowsdesc
[65] http://docs.oasis-open.org/xliff/xliff-core/v2.0/os/xliff-core-v2.0-os.html#d0e333
[66] http://www.w3.org/TR/its20/#LocaleFilter
[67] #fromscratch
[68] http://www.w3.org/TR/its20/#target-pointer
[69] http://docs.oasis-open.org/xliff/xliff-core/v2.1/csprd01/schemas/itsm.sch
[70] http://www.w3.org/TR/its20/#idvalue
[71] http://www.w3.org/TR/its20/#directionality
[72] http://docs.oasis-open.org/xliff/xliff-core/v2.0/os/xliff-core-v2.0-os.html#d0e9515
[73] #otherlocbehcats
[74] https://issues.oasis-open.org/browse/XLIFF-9

the 2nd Public Review Draft primarily to make more ITS categories directly accessible by generic ITS Processors. The principal cause that forced the OASIS XLIFF TC to reuse the W3C namespace was that ITS processors could not be directed by ITS global rules (within the ITS Module Schematron schema[75]) even to some data categories that were implemented in XLIFF from scratch. This was impossible due to the lack of global pointers for those categories within the [6] specification. While the theoretically proper action would have been to keep the distinct OASIS ITS Module namespace and add the needed global pointers to ITS (a new dot release), this would not be practically achievable. In W3C, Working Groups (WG) are only mandated to work on specific work items and are disbanded after the intended Recommendations are published, hence a new WG would need to be assembled and mandated to produce a new version of ITS.

In spite of [12] now using the W3C namespace for the ITS Module, there is a systematic scope mismatch between the XLIFF defined ITS attributes and the ITS defined XML attributes. Because ITS 2.0 has no provision to parse pseudo-spans, it will necessarily fail to identify spans formed by XLIFF Core `<sm/>` and `<em/>` markers.

In XLIFF, Modifiers can always transform `<mrk id="1">span of text</mrk>` into `<sm id="1"/>span of text<em startRef="1"/>`, which is fundamentally inaccessible by ITS Processors without extended provisions. Unmodified or unextended ITS Rules will find the `<sm/>` nodes, if those nodes do hold the W3C ITS namespace based attributes or native XLIFF attributes that can be globally pointed to by ITS rules, yet they will fail to identify the pseudo-spans and will consider the `<sm/>` nodes empty, ultimately failing to identify the proper scope of the correctly identified datacat. XLIFF implementers who want to make their XLIFF Stores maximally accessible to ITS processors are encouraged to avoid forming of `<sm/>` based spans, it is however often not possible. Had it been possible, XLIFF would have not needed to define `<sm/>` and `<em/>` delimited pseudo-spans in the first place.

Principal reasons to form pseudospans include the following requirements: 1) capability to represent non-XML content, 2) need for overlapping annotations, 3) capability to represent annotations overlapping with formatting spans as well as 4) annotations broken by segmentation (which has to be represented as well formed structural albeit transient nodes).

5. Impact and what's next

[12] gives guidance how to roundtrip each of the 19 ITS 2.0 datacats. All of the ITS module's based metadata is accessible by ITS Processors, except for the pseudo-span issue described above. ITS Procesors can easily implement an additional

[75] http://docs.oasis-open.org/xliff/xliff-core/v2.1/csprd01/schemas/itsm.sch

capability to detect spans like this one `<sm id="1"/ >span of text<em startRef="1"/>` without going into any more XLIFF specific features. Existing and overlapping features are not accessible in cases, where ITS 2.0 lacks global pointers. It is again relatively easy and straightforward to introduce these as extensions via the W3C ITS Interest Group (IG). This IG does not have the mandate to produce normative additions to ITS 2.0 or a new version, yet it can introduce new useful extensions and keep track of features needed for a potential new major version.

The release of a technically stable public review draft of [12] constitutes another important step in harmonization of Internationalization and Localization standards based at OASIS, W3C, Unicode Consortium and elsewhere. Early adopters of XLIFF 2.1 should subscribe to the XLIFF TC Comment List[76] to be notified on further progress of the review drafts towards the official publication as an OASIS Standard, hopefully in summer 2017. Importantly, XLIFF 2.1 contains the media type registration template which will result in a definitive registration of the extension `xlf` for the XLIFF 2 family of standards.

The abstract object model for XLIFF[77] as well as non-XML serializations of that model (such as JLIFF[78]) are being developed at the OASIS XLIFF OMOS TC[79]. This TC also looks into mappings to and from other Internationalization and Localization standards, as well as Localization service APIs and reference architectures. XLIFF proper - the original XML serialization of XLIFF 2 - continues being developed at the OASIS XLIFF TC[80].

Bibliography

[1] S. Saadatfar and D. Filip: Advanced Validation Techniques for XLIFF 2. Localisation Focus, vol. 14, no. 1, pp. 43-50, April 2015. http://www.localisation.ie/locfocus/issues/14/1

[2] S. Saadatfar and D. Filip: Best Practice for DSDL-based Validation. XML London 2016 Conference Proceedings, May 2016.

[3] M. Davis, Ed. Tags for Identifying Languages, http://tools.ietf.org/html/bcp47 IETF (Internet Engineering Task Force).

[4] S. Hellmann, J. Lehmann, S. Auer, and M. Brümmer: Integrating NLP using Linked Data. 12th International Semantic Web Conference, Sydney, Australia, 2013. http://svn.aksw.org/papers/2013/ISWC_NIF/public.pdf

[76] https://www.oasis-open.org/committees/comments/index.php?wg_abbrev=xliff
[77] https://github.com/oasis-tcs/xliff-omos-om
[78] https://github.com/oasis-tcs/xliff-omos-jliff
[79] https://www.oasis-open.org/committees/xliff-omos/
[80]

[5] C. Lieske and F. Sasaki, Eds.: Internationalization Tag Set (ITS) Version 1.0. W3C Recommendation, 03 April 2007. W3C. https://www.w3.org/TR/its/

[6] D. Filip, S. McCance, D. Lewis, C. Lieske, A. Lommel, J. Kosek, F. Sasaki, Y. Savourel, Eds.: Internationalization Tag Set (ITS) Version 2.0. W3C Recommendation, 29 October 2013. W3C. http://www.w3.org/TR/its20/

[7] M. Davis, A. Lanin, and A. Glass, Eds.: UAX #9: Unicode Bidirectional Algorithm.. Version: Unicode 9.0.0, Revision 35, 18 May 2016. Unicode Consortium. http://www.unicode.org/reports/tr9/tr9-35.html

[8] K. Whistler et al., Eds.: The Unicode Standard. Version 9.0 - Core Specification, July 2016. Unicode Consortium. http://www.unicode.org/versions/Unicode9.0.0/UnicodeStandard-9.0.pdf

[9] Y. Savourel, J. Reid, T. Jewtushenko, and R. M. Raya, Eds.: XLIFF Version 1.2, OASIS Standard. OASIS, 2008. Y. Savourel, D. Filip, R. M. Raya, and Y. Savourel, Eds.: XLIFF Version 1.2. OASIS Standard, 01 February 2008. OASIS. http://docs.oasis-open.org/xliff/v1.2/os/xliff-core.html

[10] T. Comerford, D. Filip, R. M. Raya, and Y. Savourel, Eds.: XLIFF Version 2.0. OASIS Standard, 05 August 2014. OASIS. http://docs.oasis-open.org/xliff/xliff-core/v2.0/os/xliff-core-v2.0-os.html

[11] D. Filip, T. Comerford, S. Saadatfar, F. Sasaki, and Y. Savourel, Eds.: XLIFF Version 2.1. Public Review Draft 01, 14 October 2016. OASIS http://docs.oasis-open.org/xliff/xliff-core/v2.1/csprd01/xliff-core-v2.1-csprd01.html

[12] D. Filip, T. Comerford, S. Saadatfar, F. Sasaki, and Y. Savourel, Eds.: XLIFF Version 2.1. Public Review Draft 02, February 2017. OASIS http://docs.oasis-open.org/xliff/xliff-core/v2.1/csprd02/xliff-core-v2.1-csprd02.html

Projection and Streaming: Compared, Contrasted, and Synthesized

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

This paper describes, compares, and contrasts two techniques designed to enable an XML document to be processed without building an entire tree representation of the document in memory. Document projection analyses a query to determine which parts of the document are relevant to the query, and discards everything else during source document parsing. Streaming attempts to execute a stylesheet "on the fly" while the source document is being read.

For both techniques, the paper describes the way that they are implemented in the Saxon XSLT and XQuery engine.

Performance results are given that apply to both techniques, in relation to the queries in the XMark benchmark applied to a 118Mb source document.

The paper concludes with a discussion of ideas for combining the benefits of both techniques and getting more synergy between them.

1. Overview

Document projection is a technique introduced by [1] designed to address the problem that some XML documents are too large to fit as a tree in main memory and therefore cannot be queried by XQuery processors that build a complete tree in memory.[1]The idea is to perform static analysis on the query to determine which parts of the source document are actually needed by the query, and then to build a tree in memory that omits the parts of the document that the query never visits.

Streaming, as we use the term in this paper, is a mechanism defined in the XSLT 3.0 specification [8] that allows a subset of the XSLT language to be processed in streaming mode. It attempts to identify those constructs in the language that can be processed in "constant memory", that is, whose memory requirement

[1]Memories were smaller in those days: Marian and Siméon report a maximum document size of 50Mb for Saxon, and as little as 7Mb for QuiP. But the fact that the limits are higher today does not mean the problem has disappeared. Indeed we are now starting to see the 2Gb boundary imposed by 32-bit addressing becoming a potential problem.

is independent of document size. Provided that the stylesheet conforms to these static constraints on the use of language constructs, a streaming XSLT processor is then expected to be able to handle documents of arbitrary size (with a few caveats, which are well documented).

So projection and streaming are addressing essentially the same problem, and there are similarities (but also significant differences) in the way they tackle it. Both mechanisms are implemented in Saxon [6], essentially independently of each other. Part of the motivation for this paper is to examine whether there are synergies to be obtained by integrating the two mechanisms more closely, for example by using common algorithms for the static analysis, or common data structures for the result of the static analysis, or by using ideas from one of the subsystems to improve the effectiveness of the other.

The paper starts with an overview of the two techniques of projection and streaming, in each case presenting first the published specifications, and then details of the implementation in Saxon. This is followed with an analysis section which attempts to draw out the similarities and differences, and the strengths and weaknesses of the two approaches. The final section presents ideas for how the two mechanisms could be made to work more effectively in tandem, hopefully giving a unified capability with the strengths of both and the weaknesses of neither.

2. Document Projection

2.1. Overview of the Marian/Siméon Technique

The essence of the Marian/Siméon document projection technique is as follows:

1. The query is analyzed (statically) to determine the paths to all elements and attributes that are needed to evaluate the query.
2. This set of paths is used to create a projection filter. The filter is applied to parsing events issued by a (typically) SAX-based parser to that some events are passed on to a tree builder, while others are discarded. This has the effect that only a subset of the elements and attributes in the original source document are retained in the tree (for example, a DOM tree) that is built in memory.
3. If the filter is constructed correctly, then running the query against the reduced (projected) tree produces the same result as running the same query against a full tree containing all elements and attributes in the original document.

Marian and Siméon illustrated this idea using query Q1 from the XMark benchmark [4], specifically:

```
for $b in /site/people/person[@id="person0"] return $b/name
```

For convenience, the XMark queries are listed in an appendix at the end of this paper. Many people would write Q1 using the simpler XPath expression:

```
/site/people/person[@id="person0"]/name
```

But the authors of XMark came from a SQL background and like many users from that tradition, their instinct was to write every query as a FLWOR expression. And document projection tries to handle the query in whatever form the users chose to write it.

We'll make an assumption here which Marian and Siméon never state explicitly, namely that the authors of such a query expect the result to be delivered either in serialized form (as lexical XML), or as a set of parentless element nodes, specifically the `name` element and its descendants (in the XMark data, this will be a single text node, but the analysis does not depend on any schema knowledge). In particular, document projection won't work if the client application expects to be able to do arbitrary navigation from the returned elements (for example, navigating from the `name` up to its containing `person`).

The set of paths needed to evaluate this query is:

```
/site/people/person/@id
/site/people/person/name #
```

where the `#` symbol can be read as meaning "together with the subtree rooted at this node".

In the case where the query was originally expressed as a FLWOR expression using the range variable `$b`, the path analysis needs to associate a set of paths with the variable binding, and then to expand these paths for all references to the variable.

The filter operation essentially takes these paths and retains a node if (a) the node matches one of these paths, or a prefix of one of these paths (for example `/site/people`), or (b) it is a descendant (or an attribute of a descendant) of one of the paths labelled `#`.

In a refinement of the technique, called *optimized projection*, the analysis considers not only which nodes are reachable, but how they are used by the query. For example, given the query `if (@discounted) then price else price - discount`, the query tests for the existence of the attribute `@discounted`; it is capable of returning the `price` element; but on the `else` branch, the `price` and `discount` elements are always atomized (though Marian and Siméon do not use quite this terminology).

Clearly the analysis becomes increasingly complex as additional XQuery features are used. Handling additional axes such as preceding and following sibling axes is one example; use of user-written functions is another. Marian and Siméon address these complications in part by reducing the XQuery language to a small core language which is more amenable to analysis. Despite this, they don't

present the full analysis in their main VLDB paper, but refer to internal technical reports for some of the detail.

With the kind of queries used in the XMark benchmark, document projection will often dramatically reduce the size of the tree that needs to be built. With one exception, all the queries in the benchmark can be executed on a document occupying only 5% of the memory of the full tree. As well as a big saving in memory, the technique gives a (smaller) improvement in execution time, because less time is spent building and searching unnecessary parts of the tree.

The main limitations of document projection, of course, are (a) that it is only effective in cases where the source document is parsed, and a source tree built, for the purpose of executing a single query, and (b) that it is only useful in cases where the query requires a small amount of information from the document. Nevertheless, these conditions apply sufficiently often in real life that the technique should probably be far more widely known and used than it is, especially as it is available in a number of popular XQuery processors.

2.2. Document Projection in Saxon: XMark Performance

Document projection has been implemented in Saxon's XQuery processor for many years, though there is little evidence that it is widely used.[2]When running a query from the command line, it can be activated simply by use of the `-projection` command line option. Statistics on its effectiveness are displayed when the `-t` option is on[3]. From the Java API it is possible to separate the two steps of analyzing a query to create a projection filter, and applying the filter to create a projected document. (This makes it possible, if you really want, to write a query whose sole purpose is to define a document projection, and then to run a variety of queries or transformations on the projected document. In this case, of course, you lose the guarantee that the results will be the same as on the full document. More realistically, you may well want to run the same query repeatedly, against the same projected source document, but with different query parameters.)

The benefits obtained by document projection in Saxon, when running the XMark benchmark queries, are very much in line with the results reported by Marian and Siméon.

For XMark Q1, against a 119Mb source document, the metrics with document projection off are:

[2]This is perhaps an understatement. In preparing examples for this paper, I found and corrected several bugs in the implementation. Since no bugs have been reported by users over several years, this leads one to suspect that the feature is essentially unused.

[3]Some of the metrics given in this paper were obtained using internal instrumentation that is not available via any public API.

```
Analysis time: 375.619 ms
Tree built in 1405.714 ms
Tree size: 4787932 nodes, 79425460 characters, 381878 attributes
Memory used: 368,331,992 [bytes]
Average execution time [across 20 runs]: 0.233 ms
```

Switching projection on changes this to:

```
Analysis time: 373.619 milliseconds
Document projection: Input nodes 5072525; output nodes 102002; reduction ▶
= 98%
Tree built in 1032.552 ms)
Tree size: 78822 nodes, 364998 characters, 25500 attributes
Memory used: 74,333,216 [bytes]
Average execution time: 0.228 ms
```

So we're seeing a reduction in the size of text nodes alone from 79M characters to 364K characters, together with a 98% reduction in the number of element nodes, leading overall to a reduction in the total memory requirement for the query from 368Mb to 74Mb[4]. There's negligible impact on the time for static query analysis, and a useful 40% speed-up in tree building time (useful because this is the dominant cost). The query execution time is negligible compared with the tree building time, so the fact that the query runs a little faster on the projected document is of no practical importance.

For the full range of XMark queries we see the following size reductions. The first figure is the reduction in the number of nodes, the second is the reduction in the number of characters in text nodes:

```
Q1   97.99%   99.95%
Q2   96.24%   99.67%
Q3   96.24%   99.67%
Q4   96.01%   99.96%
Q5   99.42%   99.94%
Q6   99.57%  100.00%
Q7   98.69%  100.00%
Q8   97.91%   99.55%
Q9   96.62%   99.42%
Q10  91.97%   97.40%
Q11  97.28%   99.47%
Q12  97.28%   99.47%
Q13  98.79%   96.40%
Q14  88.27%   65.51%
Q15  98.90%   99.99%
Q16  98.52%   99.99%
```

[4]Figures for memory usage are not very accurate or reliable, because garbage collection happens unpredictably.

```
Q17  97.99%   99.09%
Q18  99.53%   99.96%
Q19  97.86%   99.20%
Q20  98.99%  100.00%
```

I have used slightly different metrics from those used by Marian and Siméon (they reported on the size of the document in memory and on disc, rather than the total number of nodes and the size of the text nodes) so the figures are not directly comparable; however, there is a strong correlation. Marian and Siméon reported:

> *The projected document is less than 5% of the size of the document for most of the queries. On Query 19, Projection only reduces the size of the document by 40%, and it has no effects for Queries 6, 7, and 14. In contrast, Optimized Projection results in projected documents of at most 5% of the document for all queries but Query 14 (33%). The reason for this difference is that Queries 6, 7, 14 and 19 evaluate* `descendant-or-self() (//)` *path expressions for which projection without optimization performs poorly. Query 14 is a special case since it selects a large fragment of the original auction document. Obviously projection cannot perform as well for this kind of query.*

Saxon is therefore achieving very similar results to their *Optimized Projection* results, reducing the number of nodes to 4% or less for all queries except Q14. The problem with Q14 is that the query accesses the string value of `description` elements, which account for 70% of all the text content of the source document.

2.3. Implementation of Document Projection in Saxon

Saxon performs the analysis needed to implement document projection by examining the expression tree after all query parsing, type checking, and optimization is complete.

The first stage of analysis builds a data structure called the path map, which is essentially a graph representation of all the navigation paths performed by the query expression. Some of these navigation paths are explicit in the form of an axis step in the query; others are implicit in the use of constructs like a call to `fn:id` (which effectively searches all elements in the document). The roots (entry points) to this data structure represent the global (externally-supplied) context item, and calls on functions that return new documents such as `fn:doc` and `fn:collection`. Each node in the path map represents a set of XDM nodes that is visited by the expression, and the arcs emanating from this node represent the axis steps used to navigate away from these nodes to other nodes. The initial analysis represents all axes explicitly (including, for example parent and preceding-sibling axes).

Nodes in the path map are labelled at this stage with three boolean properties: *returnable* which indicates whether the relevant nodes can be returned from the

query (rather than being merely visited en route); *atomized* which indicates that the expression atomizes the nodes reached via this path; and *hasUnknownDependencies* which is discussed below.

When an expression is bound to a variable, the set of paths reachable by the initializer of the variable is noted, and when a path expression is used that starts with a reference to this variable, the relevant navigation steps are added to the graph starting at these nodes, effectively concatenating the navigation path used to evaluate the variable with the navigation paths starting at the variable's value.

Saxon does not attempt to analyze calls to user-defined functions (nor, in XQuery 3.0, dynamic function calls): it is assumed that when a node is passed to a function call (other than a known system-defined function) with no atomization, then the function can perform arbitrary navigation starting from that node, and this is indicated by setting the property *hasUnknownDependencies* on the path map node. (In some cases, however, the optimizer will have inlined function calls, in which case path analysis is still possible.[5])

The second stage of analysis is to reduce the path map so that it contains downwards navigation steps only. It's easiest to explain how this is done with some examples:

- If the query contains the path `/books//book/preceding::item`, we replace this with the two paths `/books/descendant::book` and `/descendant::item`. The logic here is that these are the elements we need to retain in the projected document: the presence of the path `/descendant::item` will ensure that all elements named `item` are retained, wherever they appear. Because the relative position of nodes is retained by the document projection process, we can be confident that the axis step `preceding::item` will return the correct `item` elements.
- If the query contains the path `/books/book/title[contains(., ../author)]`, the original analysis will produce the two paths `/books/book/title#` and `/books/book/title/parent::node()/author`. Reduction to downwards selection then changes the second path to `/books/book/author`, because the analysis is able to determine that the pair of steps (`child::title/parent::node()`) is a null navigation and can therefore be eliminated.
- If the query contains the path `/books/book/title/following-sibling::author`, the analysis will produce the two paths `/books/book/title` and `/books/book/author`. Again the analysis is able to determine that navigating down to a `title` element and then across to a sibling `author` element will always select an `author` that is a child of the `book` element.

The third and final step is the actual process of document projection. This is implemented as a filter (an instance of the Saxon class `ProxyReceiver`) on the

[5]The only XMark query to use a user-defined function is Q18, and this function is trivially inlineable.

push-based event pipeline between the SAX XML parser and the tree builder. The filter maintains a stack of currently opened-but-not-yet closed elements; on this stack is a reference to the path in the path map by which the nearest retained ancestor node was reached. An element is retained if (a) its parent is retained, and (b) there is an arc on the path map that selects this element from its parent or ancestor. Elements are also retained (c) if some ancestor node was marked with the `returnable` or `atomized` properties indicating that the entire subtree of the element is required. (Atomization actually only requires the descendant text nodes: descendants of other kinds could be discarded. But when elements are atomized, they almost invariably have a single text node child, so this additional optimization would deliver very little benefit.)

3. Document Streaming

3.1. Overview of Streaming in XSLT 3.0

Many constructs in XPath and XSLT (and XQuery, for that matter) do not lend themselves well to streamed evaluation: the semantics of the language are defined in terms of a tree that can be freely navigated in all directions (including, importantly, upwards). In principle one could identify a subset of stylesheets that only use streamable constructs, and implement these using streaming algorithms that avoid building the entire tree in memory. However, it is likely that very few real-life stylesheets would fall into this category. One particular reason for this is that XSLT, unlike XQuery, relies heavily on dynamic despatch of template rules, which makes it effectively impossible to perform static analysis of the stylesheet as a whole.

In addressing the requirement for streaming to process large documents, the XSL Working Group therefore adopted a different approach:

- Users would be required to indicate an intent that particular parts of the stylesheet (for example, template rules) should be streamable, and the XSLT processor would be required to analyze those parts for streamability.
- It should be possible within a single stylesheet to mix streamed and unstreamed processing; for example, it should be possible during streamed processing of a large document to build an in-memory representation of a small subtree of this document, and then process this subtree using the full power of the language unconstrained by streaming restrictions.
- New constructs should be added to the language to make it easier to write streamable stylesheet code. Examples of such constructs are the `xsl:merge` instruction (which allows multiple documents to be merged in a streamed operation), the `xsl:accumulator` declaration (which allows multiple aggregation functions, such as totalling and averaging, maxima and minima, to be be

computed during a single streaming pass of the document), and the `xsl:on-empty` and `xsl:on-non-empty` which make it possible to specify declaratively what should happen when required input data is absent.

XQuery is designed in the tradition of database query languages, whose developers typically aspire to the principle that optimization is entirely the responsibility of the language processor, not the user. XSLT comes more from the tradition of programming languages, where performance and selection of algorithms are the responsibility of the programmer. It is therefore to be expected that the two languages would take a different approach to streaming.

The static analysis determining whether particular XSLT constructs are streamable is complex. The rules are prescribed in the language specification, so that stylesheet authors can be confident that code designed to be streamable with one implementation will also be streamable with other implementations.

The analysis assumes the existence of an expression tree representing the results of parsing the stylesheet and the XPath expressions embedded within it. Each node in this tree represents an instruction or expression (generically, a *construct*). The analysis computes three properties for every construct:

- A static type (to which the result of evaluating the construct will always conform). The static type analysis is fairly simplistic and does not attempt to be over-precise. It is used mainly to distinguish expressions that return childless nodes (such as text nodes and attributes) from those that can return nodes with children (documents and elements), because this can make a difference to streamability.
- The *sweep* of the construct. Constructs are classified as *motionless*, *consuming* or *free-ranging*. This concept relies on the notion that when processing a document in streaming mode, there is a current position in the document representing the moving cursor that separates tags that have already been read, from tags that have not yet been read. A *motionless* construct is one that can be evaluated without moving this cursor: an example of such a construct might be `exists(@foo)` (because the processing model assumes that when the cursor is positioned on an element start tag, all the attributes of that element are available). A *consuming* construct is one that can be evaluated by moving the cursor from a start tag to the corresponding end tag, that is, by reading the descendants of the current element. An example of such a construct is `. = "foo"` (if the context item is an element, comparing its typed value to a given string requires atomizing the element, which involves reading its descendants). The third category, *free-ranging*, represents everything else: constructs like `following-sibling::x` which require navigation outside the boundaries of the current element. If the context item is a node in a streamed document (an important caveat), then a free-ranging construct leads to the stylesheet being deemed non-streamable.

- The *posture* of the construct. This concept is rather abstract and difficult to explain in simple terms; it characterizes the relationship of nodes selected by the evaluation of the expression to the position of the moving cursor. It also constrains what further navigation is allowed starting from the result of this expression.

  Streamed processing produces three possible postures. A *crawling* posture represents the result of a consuming expression that is processing all the descendants of an element, in the course of moving the cursor from the start tag to the end tag of that element. A *striding* posture is very similar, except that the expression is only processing descendant elements at a fixed depth (for example, children or grandchildren, but not a mixture of both). A *climbing* expression is one that navigates to ancestors of the element at the cursor position, or to attributes of these ancestors (the streaming model assumes that a stack of ancestor nodes and their attributes is available at all times). The key difference between these three postures is what kind of onwards navigation is permitted. When a node was reached in climbing posture (that is, by following the ancestor axis), no downward navigation is permitted, because in general this would need to access parts of the document that have already been read and discarded, or that have not yet been read. In striding posture, further downward navigation is allowed. It crawling posture, downward navigation is not allowed in the general case, because when two nodes A and B are reached in crawling posture, and A comes before B in document order, it is not necessarily the case that all descendants of A appear in document order before all descendants of B: for an example, consider the element:

```
<root>
  <section id="A">
    <section id="B">
      <footer/>
    </section>
    <footer/>
  </section>
</root>
```

  where the `footer` child of section A appears in document order *after* the `footer` child of section B. If downward selection were permitted from the results of a crawling expression such as `descendant::section`, it would not be possible to achieve this by processing one `section` at a time without buffering.

  The other two values for posture are *grounded* and *roaming*. Grounded posture applies to an expression whose result will never contain streamed nodes: for example, expressions whose value is an atomic value or a map, or a node in an unstreamed document. There are no streaming restrictions on the processing that can be applied to the result of a grounded expression. Roaming

posture, by contrast, represents the case where streamability analysis has essentially failed: it applies to an expression such as `preceding::item`, which cannot be evaluated in streamed mode.

Although the XSLT 3.0 specification is very prescriptive about the analysis used to determine whether constructs are deemed streamable or not, it has very little to say about how streamed evaluation of those constructs deemed to be streamable should actually work. That is left entirely to the implementation.

3.2. Streaming in Saxon: XMark Performance

Although streaming is specified by W3C only for XSLT 3.0, Saxon also offers the capability of streamed XQuery execution. If a query is run from the command line using the `-stream` option, Saxon will analyze the query for streamability using rules that are essentially equivalent to those of the `xsl:stream` instruction in XSLT 3.0, and if it is streamable, will execute it in streamed mode. Since we have been studying the set of XMark benchmark queries to examine the impact of document projection, it is instructive to look at the same queries from a streamability perspective.

It turns out that none of the XMark queries is streamable (in Saxon) as originally written, but many of them can easily be rewritten to make them streamable. The main changes required are described below:

- Most of the queries are written as FLWOR expressions, which are typically not streamable (a) because they use non-XPath syntax which is therefore outside the scope of the streamability rules in the XSLT specification, and (b) because they bind variables to streamed nodes. In most cases the rewrite is very simple. For example Q2 is originally written:

```
for $b in /site/open_auctions/open_auction return <increase> { $b/▶
bidder[1]/increase } </increase>
```

  which can be rewritten like this[6] to make it streamable:

```
<xsl:for-each select="/site/open_auctions/open_auction">
   <increase>
      <xsl:sequence select="bidder[1]/increase"/>
   </increase>
</xsl:for-each>
```

- Some of the queries require more extensive rewriting to avoid multiple downward selections. For example Q5 is originally:

[6]This rewrite preserves the bug in the original query whereby the output contains two levels of nested `<increase>` elements.

```
count(for $i in /site/closed_auctions/closed_auction
      where  $i/price >= 40
      return $i/price)
```

which (knowing that a `closed_auction` has only one `price`) can be made streamable by rewriting as:

```
count(/site/closed_auctions/closed_auction/price/number()[. >= 40])
```

Similarly, Q7 reads

```
for $p in /site
return count($p//description) + count($p//annotation) + count($p//▶
email)
```

which can be replaced by the streamable equivalent:

```
count(site//description | site//annotation | site//email)
```

- A few queries require limited buffering of the input. An example is Q4:

```
for    $b in /site/open_auctions/open_auction
where  $b/bidder/personref[@person="person18829"] <<
       $b/bidder/personref[@person="person10487"]
return <history>{ $b/reserve }</history>
```

where the processing can be done by building each `open_auction` in turn as a tree in memory, which can then be discarded to process the next `open_auction` (sometimes called "burst-mode streaming"). This can be achieved with the rewrite:

```
/site/open_auctions/open_auction/copy-of(.)
  [bidder/personref[@person="person18829"] << bidder/▶
personref[@person="person10487"]]
  ! <history>{ reserve }</history>
```

The queries that remain stubbornly unstreamable are those that involve joins (Q8, Q9, Q10, Q11, Q12) or sorting (Q19).

The performance of streamed versus unstreamed versions of the same query (for those queries where streaming is possible) is very consistent:

- For unstreamed execution, there is a one-off cost of building the document tree of around 1400ms, and the cost of evaluating the query is then between 2ms and 50ms.
- For streamed execution, the cost of query execution is in each case between 930ms and 1100ms.
- For comparison, we noted earlier that unstreamed execution against a projected document tree typically reduces the tree-building time from 1400ms to 1000ms, with a very small (and therefore negligible) improvement in query execution time.

We can see that both streaming and document projection (where applicable) give a modest improvement in execution time and a very substantial saving in memory usage, at the cost of having to parse the source document to run a single query: and it is the parsing cost that dominates. It's also worth noting that the query compilation cost, at 375ms, exceeds the document building cost for documents smaller than around 40Mb, even with these very simple queries.

3.3. Streaming in Saxon: Implementation

Saxon, in its latest incarnation (the current public release is 9.7 but this section applies more strictly to the planned 9.8 release), follows the streamability analysis rules in the specification very literally. It has to, because the Working Group has insisted on a clause that says that a processor that wants to claim conformance to the streaming feature must be prepared to report whether a stylesheet is streamable according to the W3C rules or not — no extensions allowed.

A difficulty here is that the streamability properties computed during the analysis are actually required in order to devise a streamed execution strategy, and they must therefore be computed for the final expression tree produced after all optimization rewrites have been completed. But some of the optimization rewrites (for example, function inlining) may convert a non-streamable construct into a streamable one, and would therefore cause Saxon's streamability verdict to differ from the W3C verdict. At user request, there is therefore an option to run the streamability analysis twice: the first run is done before all optimizations to deliver a W3C-conformant streamability assessment, and the second run is done after optimization to produce input to the execution plan.

The formalisms used in the W3C streamability analysis have inspired some changes to the design of Saxon's expression tree. Most notably, every expression now delivers information about its subexpressions (called *operands*) in a consistent way. The relationship between a parent expression and its children is now represented by an `operand` object which contains properties closely based on properties used in the W3C streamability model: for example every operand has a *usage* which explains how the parent expression makes use of the result of evaluating the child expression: this is one of *inspection* (the properties of the returned value are examined, for example using an `instance of` operator), *absorption* (the entire subtree of any nodes in the returned value is used, typically by means of atomization), *transmission* (the parent expression includes the result of the child expression directly in its own result value), or *navigation* (the parent expression performs arbitrary navigation from the nodes in the child expression's result to other nodes in the same tree).

Saxon has started to use these formalisms in other aspects of its static analysis unrelated to streaming (for example, in the analysis done to construct the path map for document projection) but there is much potential for increased reuse in this area.

The most complex aspect of streaming in Saxon, however, has nothing to do with the static analysis of streamability according to W3C rules, but rather with creating a streamable representation of the execution plan for the stylesheet, and with the actual execution of that plan.

As discussed in [3] and [5], streaming in Saxon operates in push mode. *Push* here means that the control flow is in the same direction as the data flow. This is the reverse of the usual interpreter design pattern. Instead of a parent expression requesting (pulling) data obtained from the evaluation of its subexpressions, the arrival of data from the XML parser triggers the evaluation of subexpressions dependent on that data, which in turn triggers the evaluation of parent expressions dependent on the results of those subexpressions. That is to say, the control flow is inverted: and in fact, the process of generating a push-mode representation of the stylesheet code corresponds to the process of program inversion as described many years ago in Jackson Structured Programming [2].[7]

The streaming code in Saxon essentially works in three parts:

1. The first part is the static streamability analysis. As already mentioned, this follows the rules in the XSLT 3.0 specification very closely. Every kind of expression node in the expression tree (representing different kinds of XSLT and XPath construct in the source code) has logic to compute properties such as the posture and sweep of the expression. Once computed, these are retained as cached properties on the expression tree. The logic for many expressions is delegated to the general streamability rules, which are driven by information about the operands of the expression and their usage; all expressions deliver their operands (including properties such as the operand usage, whether the focus changes, and suchlike) using a common interface.

2. The second part is the logic for expression inversion. Where a template or other construct is declared to be streamable, and where analysis reveals that it is actually streamable, the process of inversion constructs a representation of the construct suitable for streamed push-mode evaluation. In general (with some exceptions such as `xsl:choose`, `xsl:fork`, and `xsl:map`) a consuming expression will have exactly one consuming operand. It is therefore possible to identify a route through the expression tree that contains these consuming expressions. For the lowest-level such expression, we allocate a `Watch` which is triggered when the parser encounters a node that matches a particular (motionless) pattern. For each consuming ancestor expression, we allocate a `Feed` which evaluates a particular construct in push (event-driven) mode.

3. The third part comprises the push-mode implementations of every kind of instruction, expression, or system function. For example, the `Feed` for a call on

[7] This is no coincidence, because JSP was greatly concerned with the problems of managing hierarchic data in a streamed representation too large to fit in available memory, in this case using magnetic tape.

the `fn:sum` might look like this (ignoring subtleties in the actual specification of this function):

```
private double sum = 0;

public processItem(Item item) {
    sum += item.toDouble();
}

public void close() {
    getParentFeed().processItem(new DoubleValue(sum));
}
```

In this particular example, the implementation does not pass anything on to the `Feed` for its parent expression until the input sequence is exhausted. In other cases, for example the `Feed` for the `fn:distinct-values` function, values can be passed on as soon as they are known. Here is a simplified version of the `Feed` for `fn:distinct-values`:

```
private Set<AtomicValue> values = new HashSet<AtomicValue>();

public processItem(Item item) {
    if (values.add((AtomicValue)item)) {
        getParentFeed().processItem(item);
    }
}
```

When the processing of an input sequence is initialized, this code creates a new data structure holding the set of distinct values (which is initially empty). Each time a new value is received, it is added to this set (which is a no-op if it was already present in the set); and if it was not already present in the set, it is passed on to the feed for the parent expression.

These push-mode implementations need to be provided for every kind of construct that can take a sequence as input. For expressions that operate on singletons (for example, arithmetic expressions) a generic `Feed` that accepts a single item and invokes the ordinary non-streaming implementation suffices. For a few constructs it is necessary to provide more than one push-mode implementation, because the logic depends on which operand is the streaming operand: a notable example is the `LetExpression` (used not only to implement XPath `let` expressions, but also local variables declared in XSLT: the code for `let $x := distinct-values(//foo) return count($x)` is quite different from the code for `let $x := data(@id) return index-of(.//foo, $x)`, because in the first case the streamed operand is the expression used to initialize the variable, and in the second case the streamed operand is the expression that makes use of the variable.

It's worth remarking in passing that because the control flow is inverted, we can't do dynamic error handling using the normal Java machinery of throwing and catching exceptions. When evaluation of an expression fails, this must cause execution of the parent expression to fail, so the failure is pushed up the pipeline in the same way as success results.

4. Projection and Streaming: a Comparison

The major similarities between the two technologies are:

- They share the objective of using static analysis of a query or stylesheet to establish an execution strategy that reduces the amount of memory needed for the tree representation of large documents.
- There are many similarities in the details of the analysis that is carried out, although it is presented in rather different terms.
- In both cases, the technique is only effective if the source document is being parsed only for the purpose of executing a single query or stylesheet against it. Document projection has a bit more flexibility in that it allows the same query to be executed repeatedly with different parameters (for this use case, streaming requires the document to be re-parsed each time, document projection does not). If the workflow requires multiple queries or stylesheets to be executed against the same source document, neither streaming nor document projection is well suited to the task.

Some significant differences between the two approaches are:

- Document projection is defined for use with XQuery, streaming for use with XSLT. This difference is not entirely superficial, because the nature of the rule-based template processing characteristic of XSLT means that the potential for static analysis is very different from the XQuery case.
- Document projection can be applied to any query. It may be more effective for some queries than others (for some queries the memory requirement might be reduced by 5%, for others by 95%), but the mechanism is designed to work with any query. By contrast the streamability analysis in XSLT makes a binary classification of stylesheets as being either streamable or not streamable, and a stylesheet in the latter category achieves no benefit.
- Document projection typically achieves a linear reduction in the amount of memory needed (for example, to 20% of the original memory requirement), whereas streaming is designed to run in constant memory completely independently of document size. This means that streaming can even be applied to infinite documents, such as a continuous stream of telemetry data.
- Document projection is designed to happen entirely behind-the-scenes, without the knowledge or involvement of the query author, whereas XSLT stream-

ing provides explicit constructs for the invocation of streaming, and alternative ways of writing code when streaming is required, including brand-new mechanisms such as accumulators. This in turn reflects a philosophical difference between XQuery and XSLT, or between query languages and programming languages in general, where the intellectual focus in query language theory has always been automated optimization, while the focus for programming languages has been enabling the programmer to achieve the desired performance metrics as well as correct execution.

5. Seeking Synergy

In this section we'll start looking for ways in which the benefits and of streaming and document projection can be combined, and in which the weaknesses of both can be reduced. We'll start with an example to illustrate the challenges.

5.1. A Simple Example

Consider the following query, which computes the average value of the transactions in a transaction file:

```
[R1] avg(//transaction/(price * quantity))
```

As we've seen, many XQuery users, especially those trained in SQL, will write this by preference as:

```
[R2] avg(for $t in //transaction return $t/price * $t/quantity)
```

Document projection will analyze either of these queries and establish that the only nodes that need to be retained when building the source document tree are the document node, the `transaction` element nodes, and the `price` and `quantity` elements together with their text node descendants (or their typed value, if the processing is schema-aware). This may be a very substantial reduction on the size of the source tree that would otherwise be built, but it is still proportional to the number of transaction elements in the file, so it will run out of memory eventually.

Streamability analysis in XSLT 3.0 will reject both these queries as non-streamable. The second query (R2) is rejected because it binds variables to streamed nodes, and the streamability analysis in the XSLT specification gives up at this point. The first query (R1) is rejected because there is an expression `(price * quantity)` that makes two downward selections in the tree. In the language of the specification, it has two operands whose sweep is consuming. In implementation terms, the difficulty is that in the general case, an expression with multiple downward selections cannot be evaluated without buffering data in memory, and the amount of buffering cannot be predicted, and therefore the query cannot execute in constant memory so it cannot be said to be full streamable. Of course, you

and I can see that in this case the amount of buffering needed is trivial – but that is only because we know instinctively what kind of values we expect to find in elements named `price` and `quantity`.

There's another more subtle difficulty with streaming of this expression: because the `transaction` elements are selected using the shorthand notation `//transaction`, the processor has to be prepared to handle nested transactions: the query has a well-defined outcome for a pathological input such as:

```
<transaction>
  <price>8.25</price>
  <transaction>
     <price>13.50</price>
     <quantity>4</quantity>
  </transaction>
  </quantity>13</quantity>
</transaction>
```

The XSLT 3.0 solution to this is to ask the programmer to rewrite the query in a way that makes the buffering explicit, reflecting the fact that the programmer understands the data much better than any optimizer. Typically it will be rewritten as:

```
[R3] avg(/*/transaction/copy-of(.)/(price * quantity))
```

The injected `copy-of(.)` step explicitly makes an in-memory copy of the transaction element and its subtree (unless, of course, the optimizer finds a smarter way of doing things), and the navigation to the `price` and `quantity` elements is then conventional (unstreamed) navigation within an in-memory tree.

We can see that neither the projection approach nor the streaming approach is perfect here. Document projection uses more memory than it needs, because it fails to recognize that the `avg()` function only needs access to one transaction at a time (the only memory it needs is, at most, a running total and a running count). Streaming requires programmer intervention to copy the transaction nodes; and these subtrees are bigger than they need to be because we copy the whole subtree rather than merely the price and quantity elements. (In some transaction files the transaction element might be very large, for example it might contain an entire history of contract negotiations leading up to an eventual purchase.)

It seems evident that there's room for improvement in how we execute this query. We should be able to get the streaming benefits of only holding one transaction in memory at a time. We should be able (in some mode of operation) to automate the `copy-of()` operation, and we should be able to use a projection-like technique to ensure that this copy contains only the required elements (price and quantity) rather than the full transaction element.

The following sections examine these opportunities in more detail.

5.2. Tolerating Local Variables

The static analysis performed for document projection can handle the existence of local variable bindings within the expression, whereas the streamability analysis in XSLT 3.0 fails as soon as a streamed node is bound to a local variable (with one exception, the case where the variable is a reference to the first argument of a streamable stylesheet function). As we've seen, local variables are used liberally in practical queries, and it would be nice to handle them if we can.

An earlier draft [7] of the XSLT 3.0 specification did in fact attempt this (and indeed, it worked in terms of a path map that was explicitly inspired by Marian and Siméon).

The most obvious way in Saxon to make queries such as

```
for $b in /site/people/person[@id="person0"] return $b/name
```

streamable is to rewrite the query during the optimization phase to eliminate the local variable, relying instead on binding the context item. Saxon already does this for let expressions, in cases where the variable is only used once. Saxon also turns where clauses into predicates when possible, so for example

```
for $e in /*/employee where $e/salary > 10000 return $e/name
```

is rewritten as

```
for $e in /*/employee[salary > 10000] return $e/name
```

Rewriting it as

```
/*/employee[salary > 10000] ! name
```

is no more difficult. (I've used the "!" operator here as the direct equivalent of the original semantics, because there is no de-duplication or sorting into document order. However, the expressions X!Y and X/Y are equivalent in the case where both X and Y have striding posture, as is the case here.)

The basic condition for doing such a replacement is that the focus for evaluating the variable reference is the same as the focus for evaluating its declaration.

5.3. Tolerating Multiple Consuming Operands

We have seen that an expression such as

```
avg(/*/transaction/(price * quantity))
```

fails the streamability rules because it makes two downward selections (or in the language of the spec, it contains two consuming operands). But it can be made streamable by rewriting it as:

```
avg(/*/transaction/copy-of(.)/(price * quantity))
```

Two questions arise: (a) can this rewrite be automated, and (b) can we do better, by only copying the `price` and `quantity` elements, and not the whole `transaction`?

The danger in automating this rewrite is that in the worst case, it could generate a copy of the entire streamed document, thus effectively masking the fact that the query is not really streamable.

Rather than generating an implicit copy, another approach to making this streamable is to generate an implicit `xsl:fork` instruction. The `xsl:fork` instruction allows multiple consuming sub-expressions to be computed during a single pass of the input document, buffering the results of each sub-expression and combining them when the relevant subtree of the input document has been completely consumed.

Map constructors also allow multiple consuming operands, so as an alternative to `copy-of()` or `xsl:fork`, it is possible to make this streamable by writing:

```
avg(/*/transaction/map{"price":data(price), "qty":data(quantity)}!(?
price * ?qty)
```

(Note the need to explicitly atomize the streamed elements by calling `data()`, because streamed nodes cannot be stored in a map.)

Generating a projection of the input subtree can be seen as a further possibility.

So there is a range of possible ways foward that could make it easier to write streamable applications that require multiple downward selections. The main challenge, in fact, is to distinguish those cases that can be done with minimal buffering, like this one, from those where buffering data would essentially mean that the data was not being streamed at all. So it comes down to static analysis. We can see that this case works because the operator with multiple consuming operands (that is, the multiplication operator) requires those operands to be singleton atomic values. Perhaps this is the case to tackle first. This would mean relaxing the general streamability rules in the XSLT 3.0 specification so that the rule starting "If more than one operand is potentially consuming..." has an additional clause:

> *If each of the potentially consuming operands has operand usage absorption and has a required type with item type atomic and cardinality zero or one, then [the expression is] grounded and consuming.*

Implementing streaming for this case would not be difficult; the logic would essentially be the same as for map constructors.

5.4. Document Projection in XSLT

One reason that document projection has been relatively little used is that it only works in XQuery. Apart from the context of XML databases (where streaming

and document projection are not relevant), XSLT is far more widely used than XQuery. So we need to ask the question whether document projection could be implemented in an XSLT context.

The traditional difficulty here has been that XSLT does not permit the kind of static analysis necessary, because of the dynamic nature of template rules. However, this was also a problem for streamability analysis, and in that case the problem has been successfully overcome. The way it was overcome was to allow a mode to be declared as streamable, and to require all template rules in a streamable mode to conform to a number of statically-checkable constraints: chiefly, the match pattern must be motionless, the body of the template rule must be either motionless or consuming, and the body must be grounded (that is, the template must not return streamed nodes).

There's another potential objection to doing document projection in XSLT, which is that many transformations use almost all the data in the source tree when constructing the result tree. (Or to put it another way, XSLT is used for transformation rather than for query.) However, the fact that the technique isn't appropriate to all uses cases doesn't mean that it isn't appropriate to any, and it's easy enough to find examples of stylesheets that extract a small part of the input document.

The obvious place to start document projection in XSLT 3.0 is with the new `xsl:source-document` instruction. This could be given an attribute `projection="yes"` to be used when the stylesheet author thinks that use of document projection would be beneficial.

The main challenge in implementing this is the need to define the analysis rules determining how all XSLT instructions affect the path map. Some of these instructions like `xsl:number` are quite complex. The rules also need revisiting to consider new constructs that have appeared in XPath 3.0, notably dynamic function calls. However, this challenge also presents an opportunity: many of the concepts introduced for the sake of streamability analysis are probably reusable to create general rules for projection analysis. For example:

- The concept of *operand usage* distinguishes four kinds of processing that may be applied to the nodes returned by an expression: *inspection* reads properties of the node and performs no further navigation; *absorption* reads the subtree rooted at a node; *transmission* includes the nodes returned by a sub-expression in the result of the parent expression; and *navigation* permits arbitrary navigation from a node to anywhere else in the tree. These concepts are just as applicable to projection analysis as to streaming analysis, and the fact that all constructs already classify their operands according to these categories means that it should be possible to eliminate many ad-hoc rules that currently appear in the projection analysis.

- The concept of a *focus-setting container* with *controlling* and *controlled* operands is used by the streamability analysis to trace paths where there is a dependency on the context item. Again, Saxon already performs this analysis for streamability purposes, and this could be used to eliminate many ad-hoc rules in the projection analysis.
- In the opposite direction, some of the concepts used for path analysis could be exploited to improve streamability analysis. For example, code that uses the `following-sibling` axis is currently non-streamable; but one can envisage cases where path analysis could be used to identify cases where streaming of this axis is possible.

Performing data-flow analysis across calls of functions and templates remains challenging. The current document projection code in Saxon does not attempt it even for static function calls, let alone dynamic function calls or `xsl:apply-templates`. The new packaging facilities in XSLT 3.0, which allow a function to be overridden in another separately-compiled package, complicate this even further. The approach used for streamability analysis, which relies on modes and functions being manually annotated to describe the scope of their navigation, might be one way forward: realistically, however, only a very small minority of users will understand such features well enough to gain benefit from them. In addition, document projection works best for simple queries, so it might be best to concentrate on doing the best possible job in simple cases.

5.5. Automatic Streaming and Projection

The fact that document projection has been so little used should remind us of an awkward truth: any feature that delivers performance benefits, but needs to be actively enabled by users, will be ignored by the vast majority of the user population. Users are too busy to research all the tools available. If a job is run every day, they will tolerate the fact that it takes two minutes to run rather than investigating ways of reducing the run time to ten seconds.

By contrast, a feature that delivers improved performance "out of the box" will give more users an improved experience and will enhance the reputation of the product in the marketplace.

So we should ask the question whether there are situations where document projection (and indeed streaming) can be enabled automatically, by default.

In some cases we know that a query or stylesheet is only being executed once, and that a source document is being built in memory to be processed once. The main example of this is when the query or transformation is run from the command line. Unfortunately other cases, such as running within Ant, are harder to detect, because products like Ant use a general-purpose compile-build-transform API where the stages are invoked independently of each other. However, providing a "single-shot" API could achieve the twin benefits of (a) providing a much

simpler-to-use interface for applications, and (b) enabling the query or transformation processor to know that there are no hidden complexities.

That then raises the question of whether it is feasible and cost-effective to examine a query or stylesheet to decide whether document projection should be used. Or indeed, whether it would do any harm to use it unconditionally. The worst that can happen is that a slight extra cost is incurred during query analysis, and a slight extra cost during document building, in those cases where there is no benefit.

We could start with the following experiment: when running XQuery from the command line, if the query contains no user-defined functions that take nodes as arguments, then enable document projection by default.

If nothing else, this would at least ensure that any bugs in the projection code are quickly discovered.

References

[1] Amélie Marian Jérôme Siméon Projecting XML Documents Proc VLDB 29, Berlin, Germany, 2009 https://www.cs.rutgers.edu/~amelie/papers/2003/xmlprojection.pdf

[2] M. A. Jackson: Principles of program design. academic press, London, 1975.

[3] Michael Kay You Pull, I'll Push: on the Polarity of Pipelines Presented at Balisage: The Markup Conference 2009, Montréal, Canada, August 11 - 14, 2009. In Proceedings of Balisage: The Markup Conference 2009. Balisage Series on Markup Technologies, vol. 3 (2009). DOI: 10.4242/BalisageVol3.Kay01. Available at https://www.balisage.net/Proceedings/vol3/html/Kay01/BalisageVol3-Kay01.html

[4] XMark http://www.xml-benchmark.org

[5] Michael Kay Streaming in the Saxon XSLT Processor Presented at XML Prague 2014. Available at http://archive.xmlprague.cz/2014/files/xmlprague-2014-proceedings.pdf

[6] Saxon http://www.saxonica.com/

[7] XSL Transformations (XSLT) Version 2.1. W3C Working Draft (superseded), 11 May 2010. Ed. Michael Kay. http://www.w3.org/TR/2010/WD-xslt-21-20100511/

[8] XSL Transformations (XSLT) Version 3.0. W3C Candidate Recommendation, expected to be published 7 February 2017. Ed. Michael Kay. http://www.w3.org/TR/xslt-30

A. Appendix: XMark Queries

This appendix lists the XMark queries in the form they were run for the document projection tests.

They differ from the queries published at `http://www.xml-benchmark.org` in taking input from the context document rather than using the `doc()` function: this was done for convenience, allowing the same query to be used with different-sized input files.

Table A.1. XMark Queries

Name	Query
Q1	`for $b in /site/people/person[@id="person0"]` `return $b/name`
Q2	`for $b in /site/open_auctions/open_auction` `return <increase> {$b/bidder[1]/increase } </increase>`

Q3	`for $b in /site/open_auctions/open_auction`
	`where $b/bidder[1]/increase *2 <= $b/bidder[last()]/increase`
	`return <increase first="{$b/bidder[1]/increase}"`
	`last="{$b/bidder[last()]/increase}"/>`

Q4	`for $b in /site/open_auctions/open_auction`
	`where $b/bidder/personref[@person="person18829"] <<`
	`$b/bidder/personref[@person="person10487"]`
	`return <history>{ $b/reserve }</history>`

Q5	`count(for $i in /site/closed_auctions/closed_auction`
	`where $i/price >= 40`
	`return $i/price)`

Q6	`for $b in /site/regions/*` `return count ($b//item)`
Q7	`for $p in /site` `return count($p//description) + count($p//annotation) + count($p//email)`

Q8	`let $a := for $t in /site/closed_auctions/closed_auction`
	`where $t/buyer/@person = $p/@id`
	`return $t`
	`return <item person="{$p/name}"> {count ($a)} </item>`

Name	Query

Q9	`let $auction := (/) return`
	`let $ca := $auction/site/closed_auctions/closed_auction return`
	`let`
	`  $ei := $auction/site/regions/europe/item`
	`for $p in $auction/site/people/person`
	`let $a :=`
	`  for $t in $ca`
	`  where $p/@id = $t/buyer/@person`
	`  return`
	`  let $n := for $t2 in $ei where $t/itemref/@item = $t2/@id return $t2`
	`  return <item>{$n/name/text()}</item>`
	`return <person name="{$p/name/text()}">{$a}</person>`

Q10	`declare boundary-space strip;`
	`for $i in distinct-values(`
	`  /site/people/person/profile/interest/@category)`
	`let $p := for  $t in /site/people/person`
	`  where  $t/profile/interest/@category = $i`
	`  return <personne>`
	`  <statistiques>`
	`  <sexe>{ $t/profile/gender }</sexe>`
	`  <age>{ $t/profile/age }</age>`
	`  <education>{ $t/profile/education}</education>`
	`  <revenu>{ $t/profile/@income } </revenu>`
	`  </statistiques>`
	`  <coordonnees>`
	`  <nom>{ $t/name }</nom>,`
	`  <rue>{ $t/address/street }</rue>`
	`  <ville>{ $t/address/city }</ville>`
	`  <pays>{ $t/address/country }</pays>`
	`  <reseau>`
	`  <courrier>{ $t/emailaddress }</courrier>`
	`  <pagePerso>{ $t/homepage }</pagePerso>`
	`  </reseau>`
	`  </coordonnees>`
	`  <cartePaiement>{ $t/creditcard }</cartePaiement>`
	`  </personne>`
	`return <categorie>`
	`  <id>{ $i }</id>`
	`  { $p }`
	`  </categorie>`

Q11	`for $p in /site/people/person`
	`let $l := for $i in /site/open_auctions/open_auction/initial`
	`  where $p/profile/@income > (5000 * $i)`
	`  return $i`
	`return <items name="{$p/name}">{ count ($l) }</items>`

Name	Query

Q12	`for $p in /site/people/person`
	`let $l := for $i in /site/open_auctions/open_auction/initial`
	`  where $p/profile/@income > (5000 * $i)`
	`  return $i`
	`where  $p/profile/@income > 50000`
	`return <items person="{$p/name}">{ count ($l) }</items>`

Q13	`for $i in /site/regions/australia/item` `return <item name="{$i/name}">{ $i/description }</item>`

Q14	`for $i in /site//item`
	`where contains ($i/description,"gold")`
	`return ($i/name, $i/description)`

Q15	`for $a in /site/closed_auctions/closed_auction/annotation/`
	`  description/parlist/listitem/parlist/listitem/text/emph/keyword`
	`return <text>{ $a }</text>`

Q16	`for $a in /site/closed_auctions/closed_auction`
	`where exists ($a/annotation/description/parlist/listitem/parlist/`
	`  listitem/text/emph/keyword/text())`
	`return <person id="{$a/seller/@person}" />`

Q17	`for  $p in /site/people/person`
	`where  empty($p/homepage/text())`
	`return <person>{$p/name}</person>`

Q18	`declare namespace f="http://f/";`
	`declare function f:convert ($v)`
	`{`
	`  2.20371 * $v (: convert Dfl to Euro :)`
	`};`
	
	`for  $i in /site/open_auctions/open_auction`
	`return f:convert($i/reserve)`

Q19	`for  $b in /site/regions//item`
	`let  $k := $b/name`
	`order by $k`
	`return <item name="{$k}">{ $b/location } </item>`

Name	**Query**

Q20	`<result>`
	` <preferred>`
	`  {count (/site/people/person/profile[@income >= 100000])}`
	` </preferred>`
	` <standard>`
	`  {count (/site/people/person/profile[@income < 100000`
	`  and @income >= 30000])}`
	` </standard>`
	` <challenge>`
	`  {count (/site/people/person/profile[@income < 30000])}`
	` </challenge>`
	` <na>`
	`  {count (for  $p in /site/people/person`
	`  where  empty($p/profile/@income)`
	`  return $p)}`
	` </na>`
	`</result>`

The HTML 5.1 DTD

Marcus Reichardt
<u123724@gmail.com>

Abstract

Based on W3C's HTML5.1 specification, a new SGML DTD for HTML5.1 with broad applicability for validation and normalization is developed.

A formal grammar for HTML is as useful as ever in any HTML-centric workflow, such as content authoring, preservation of web content and digital heritage, establishing a baseline for defining contractual obligations of web authors, agencies, and other content providers delivering web content, archival of legal documents in web-mediated business or administrative transactions.

The DTD presented here is biased towards security and forward-engineering tasks in that it dispenses with legacy practices such as uppercase HTML markup support and unsafe handling of script data. Due to SGML's flexibility, with only very minor changes to the DTDs and accompanying SGML declaration, the DTD can also be employed for parsing legacy web content for applications such as crawlers, extracting web content for further processing, web scraping, etc.

Moreover, DTD customization can also be used for emerging web markup practices such as HTML custom elements, and Google's AMP profile of HTML. A formal and well understood, rather than ad-hoc process of making web content available to robust XML back-end processing pipelines is also valuable for the XML community.

While SGML has largely been treated as a legacy technique by the web community for years now, this work also shows that not only is SGML capable of describing and parsing HTML5 precisely and elegantly, it's the only game in town being able to tackle formal processing of web content based on an international standard.

The DTD developed here, as well as additional material not included in the proceedings due to space reasons, is available at [4], including coverage of SGML declaration details such as the lexical space for HTML element names and other identifiers, predefined entities, coverage of custom elements/attributes, and contributions to a future revision of ISO/IEC 8879

Keywords: SGML, HTML, XML

1. Introduction

W3C has recently published its HTML5.1 specification ([1]) based on work by WHATWG. HTML5.1 is the first published HTML specification after the initial HTML5 specification two years earlier. Informed by feedback of a large community of practitioners, HTML5.1 can be considered the first published HTML specification with broad applicability since the HTML4 specification in 1999.

W3C's HTML5.1 specification text is addressed at both browser implementers and web authors, hence contains redundancies, and possibly even contradictory rules, insofar as the HTML markup language is specified twice: once as what can be considered the nominal grammar productions for HTML ([1], chapter 4), and a second time as a partial procedural specification for parsing HTML aimed at implementers of HTML user agents.

In this paper, the focus is on the HTML markup language as expressed by the nominal HTML grammar presented in chapter 4 of the specification. The procedural specification for parsing HTML, having recovery rules for parsing almost anything as HTML, isn't considered in this work; it's role within the specification framework is that of avoiding unspecified and potentially malicious behaviour, and hence is considered strictly addressed at browser implementers rather than the larger web community.

In reformulating HTML5's parsing rules as DTD grammar, fundamental rules about HTML's grammatical construction known to the developers of earlier HTML DTDs, but lost in HTML5's grammar presentation are recovered, instilling confidence in the transcription process, and also aspiring to a more conclusive, but in any case more succinct expression of the HTML markup language.

A new formulation of HTML in a formal framework also benefits further HTML specification work. For example, the analysis of HTML5.1's parsing rules, afforded by reformulation as SGML DTD, has uncovered the following apparent flaws in the definition of HTML parsing rules:

- parts of the the table parsing rules are ambiguous, and not enforceable using W3C's validation software ([3]); also, the specification text contains invalid markup with respect to table data

- the definition of the the `datalist` element puts HTML parsing (the HTML membership decision problem) into a higher computational complexity class than without, which is believed unintended.

The DTD developed here is a straightforward translation of the specification text for HTML's content model and tag omission rules into DTD grammar rules. In the DTD, specification text is included as SGML comment along with the translated DTD rule for reference; where HTML content model rules are represented incompletely, a note is included in the SGML comment for the declaration as well.

In the next section, the most important HTML markup features and their mapping to SGML DTD rules are discussed.

2. The HTML5.1 DTD

The HTML5 specification document states that

> *XML DTDs cannot express all the conformance requirements of this specification. Therefore, a validating XML processor and a DTD cannot constitute a conformance checker. Also, since neither of the two authoring formats defined in this specification are applications of SGML, a validating SGML system cannot constitute a conformance checker either ([2])*

but doesn't provide examples where SGML can't *specifically* be used for checking. For XML DTDs and other XML-based schema languages it's easy enough to conclude these can't describe HTML for their lack of a way to express empty elements, omitted tags, omitted attribute names, and unquoted attributes, among other things.

For SGML, on the other hand, it's less obvious, hence this text discusses parsing and validation issues of modern HTML using SGML in depth.

2.1. Flow and phrasing content

The HTML5.1 specification text introduces the elements of HTML using a taxonomic approach and presents a classification and accompanying Venn diagram depicting inclusion relationships between HTML element categories derived from definitions contained in earlier HTML DTDs.

It is felt that the fundamental grammatical construction of the HTML vocabulary as inline content wrapped in an optional layer of block-level content isn't quite apparent in this definition. Earlier HTML DTDs included the following definition for `flow` capturing this in a rather straightforward way:

```
<!ENTITY % flow "%block; | %inline;">
```

While HTML5 lacks it, a definition for *block content* rises again by subtracting those elements from the *flow content* category that aren't also in the *phrasing content* (inline) category.

Nesting of phrasing into flow elements is about the most basic property of the HTML grammar. In SGML, the flow/phrasing hierarchy is expressed by declaring the content model of flow elements as allowing `%phrasing;`, where `%phrasing;`, like in earlier HTML DTDs, is substituted into a string such as `a|abbr|...` containing all phrasing elements as a name group.

For example, the element declaration for the `p` element is as follows:

```
<!ELEMENT p - O (#PCDATA|%phrasing;)* -(%flow_only;)>
```

meaning that

- the content of `p` elements can be any sequence of text content (`#PCDATA`) and phrasing elements, and
- flow content isn't just forbidden in direct child content of `p` (via admitting `%phrasing;` elements only), but also isn't admitted anywhere in descendant content of `p`, as expressed by the `-(%flow_only;)` SGML exclusion exception, and
- the end-element, but not the start-element tag of `p` can be omitted, as declared in `p`s tag omission indicator `- O` (see Tag Omission)

where the `flow_only` parameter entity contains block-level elements as described above, ie. flow elements that aren't also phrasing elements.

2.2. Tag omission

The HTML5 specification lists tag omission rules for each applicable element (or element combination) individually. For example, the specification text for the `p` element reads

> *A p element's end tag may be omitted if the p element is immediately followed by an address, article, aside, blockquote, details, div, dl, fieldset, figcaption, figure, footer, form, h1, h2, h3, h4, h5, h6, header, hr, main, menu, nav, ol, p, pre, section, table, or ul, element, or if there is no more content in the parent element and the parent element is an HTML element that is not an a, audio, del, ins, map, noscript, or video element.*

For this HTML5 DTD, the text description is transcribed into SGML tag omission indicators in the most straightforward way, based on whether start-element tag, end-element tag, or both start- and end-element tag omission is allowed *at all*, avoiding verbose enumeration of specific elements.

For example, the tag omission rules for `p` are represented using this simple SGML element declaration:

```
<!ELEMENT p - O (#PCDATA|%phrasing;)*>
```

where the `-` (hyphen/minus, meaning *not omissible*) and the `O` (letter O, meaning *omissible*) prescribes `p`s start-element and end-element tag omission behaviour, respectively, and `%phrasing;` expands to the string `a|abbr|...` containing HTML's phrasing elements.

This element declaration will make an SGML parser end a paragraph element if encountering any element not in the `phrasing` category, or an end-element tag which isn't balanced within `p`'s content, thereby capturing HTML's parsing rules.

As presented in the HTML5 specification, the choice of explicitly enumerated elements that cause the paragraph element to be terminated may seem arbitrary,

but is in fact (up to potential minor omissions considered errors) the set of HTML elements that are in the "flow" category, without also being in the "phrasing" category. This isn't surprising, since HTML parsing rules were originally specified using SGML grammars such as the above. By recovering HTML's original parsing rules from HTML5's specification text, we conclude that HTML5's parsing rules are represented adequately, and more succinctly, since avoiding redundantly specifying `p`-terminating elements.

This interpretation is also supported by the fact that for the HTML5.1 specification update (vs. HTML5), the new `details`, `figcaption`, `figure` and `menu` elements (which are flow-only elements) but not the new `picture` element (which can also be used in phrasing content) have been added to the set of `p`-terminating elements.

2.2.1. End-element tag omission

End-element tag omission is commonly used in HTML in the following situations

- on list items when directly followed by other list items or when followed by `ul` or `ol` end-element tags
- on definition list terms or definitions, when followed by other terms or definitions, or when followed by a `dl` end-element tag
- on paragraphs, when followed by other paragraphs or by a parent element's end-element tag
- in `head` content, upon encountering an element that can't be placed into `head` element content
- at the end of a document. for `html`, `body`, and other elements (these elements also allow start-tag omission)

All of these uses are parsed/validated by this HTML5 DTD in the expected way and in the same way as W3C's HTML5 validation software (as far as can be told).

2.2.2. Start-element tag omission

Start-element tag omission (other than in table content), is only allowed on the `html`, `head`, and `body` elements, which is trivially supported by this HTML5.1 DTD in the expected way.

2.2.3. Start- and end-element tag omission in table content

Tag omission in table content deserves a closer examination.

The relevant specification text reads as follows:

> *Content: In this order: optionally a caption element, followed by zero or more colgroup elements, followed optionally by a thead element, followed by either zero*

or more tbody elements or one or more tr elements, followed optionally by a tfoot element, optionally intermixed with one or more script-supporting elements.

Tag omission: Neither tag is omissible.

Content: Zero or more tr and script-supporting elements

Tag omission: A thead element's end tag may be omitted if the thead element is immediately followed by a tbody or tfoot element.

Content: Zero or more tr and script-supporting elements

Tag omission: A or element's end tag may be omitted if the tbody element is immediately followed by a tbody element.

Content: Zero or more tr and script-supporting elements.

Tag omission: A tbody element's start tag may be omitted if the first thing inside the tbody element is a tr element, and if the element is not immediately preceded by a tbody, thead, or tfoot element whose end tag has been omitted. (It can't be omitted if the element is empty.). A tbody element's end tag may be omitted if the tbody element is immediately followed by a tbody or tfoot element, or if there is no more content in the parent element.

By the specification text, the following HTML fragment (representing typical use of tag omission in table content) is allowed and accepted by both this HTML5 DTD, as well as by W3C's HTML5 validation software:

```
<table>
    <thead>
        <tr>ONE
        <tr>TWO
        <tr>THREE
    <tbody>
        <tr>One...
        <tr>Two...
        <tr>Three...
</table>
```

However, the specification text for `tbody` also admits omission of a `tbody` start-element tag *if the first thing inside the tbody element is a tr element, and if the element is not immediately preceded by a* `tbody`*,* `thead`*, or* `tfoot` *element whose end tag has been omitted*.

The relevant part of the content model for `tbody`'s parent element, `table`, admits either *zero or more* `tbody` *elements, or one or more* `tr` *elements*.

Hence, table content such as the following

```
<table>
    <thead>
        <tr>ONE
        <tr>TWO
```

```
        <tr>THREE
    </thead>
    <tr>One...
    <tr>Two...
    <tr>Three...
</table>
```

is valid according to at least *two* rules in the HTML specification: the `tr` elements following `thead` can be parsed by HTML5's parsing rules either as `tr` elements being placed directly into `table` content, or as an instance of a `tbody` element with omitted start- and end-element tags.

This HTML5 DTD (as does W3C's s validator and web browsers) interprets such content as an instance of the former case, and requires that a `tbody` start-element is present in content to force the latter interpretation. The ambiguous production rule for `tbody`, as stated in the HTML5.1 specification, can never apply in the absence of start-element tags for `tbody`.

Presumably, ambiguousness of tag omission rules for table content is inadvertent; even the specification text (chapter 6.7.6) itself seems to use tag omission in table models incorrectly. The W3C HTML5.1 specification contains this fragment (the leading paragraph is included for locating the text place in the document):

```
<p>The element <var>host element</var> to create for the media
is the element given in the table below in the second cell of
the row whose first cell describes the media. The appropriate
attribute to set is the one given by the third cell in that same row.</p>
<table>
    <thead>
        <tr>
            <th> Type of media
            <th> Element for the media
            <th> Appropriate attribute
        <tr>
            <td> Image
            <td> <code>img</code>
            <td> <code>src</code>
        <tr>
            <td> Video
            <td> <code>video</code>
            <td> <code>src</code>
        <tr>
            <td> Audio
            <td> <code>audio</code>
            <td> <code>src</code>
</table>
```

Note this table doesn't contain a body which however isn't the point here. Going by how the table is rendered and by analogy with other text places containing table use which *do* have a `tbody` elements specified, it can be concluded what is probably intended here is that the first `tr` element should be treated as major table heading row, while subsequent rows should be treated as table body rows.

The rule for tag omission of the `thead` element reads *A thead element's end tag may be omitted if the thead element is immediately followed by a `tbody` or `tfoot` element* (in both the W3C HTML5.1, and the current WHATWG specification text), hence we expect the above fragment to be rejected since the rule does *not* say that a `thead` end-element tag can be omitted if followed by a `table` end-element tag (when other parsing rules for end-element omission state this kind of condition explicitly).

However, the fragment is happily accepted by W3C's validation software, and hence slipped into the published specification text; the HTML5 DTD follows W3C's validator here and accepts it as well.

In an attempt to interpret HTML5's syntax rules, we note that a sentence such as *A `thead` element's end tag may be omitted if the `thead` element is immediately followed by a `tbody` or `tfoot` element* is inherently self-referential since the `thead` element's end isn't yet established while assessing it's end-element omission status, hence whatever follows it in content isn't either (the definition of `tbody`s start-element tag omission stated earlier has a similar problem).

Obviously, standard co-inductive/well-founded semantics for grammars can't be applied here without further qualification. Given quite obvious flaws in table content models on a cursory look already, and mildly surprising results when using the reference validation software, further discussion of HTML5's nominal table parsing rules seems hopeless, and isn't expected to contribute to the definition of an inter-operable table content model.

Hence, while the HTML5 DTD behaves the same as the reference validation software, authors are advised to not rely on tag omission in table content beyond basic idiomatic usage as described.

2.2.4. Tag omission in body content

In older HTML DTDs, formally only block-level elements can appear directly in a HTML document `body`; phrasing content had to be wrapped into at least a paragraph (or generic block-level `div`) container element.

However, browsers never inferred block-level elements when they where missing in content (or made their presence visible in the DOM). Essentially, this constraint was never enforced.

The HTML5.1 grammar follows actual browser behaviour, in that *any* flow content, including phrasing content, is formally accepted as direct child of the `body` element.

2.3. The `datalist` element

The datalist element's content definition has changed from previous releases. It's specification text now reads

> *Content: Either: phrasing content. Or: Zero or more option and script-supporting elements.*

and the mapping into an element declaration is as follows:

```
<!ELEMENT datalist - - ((#PCDATA|%phrasing;)*|(option|script)*)>
```

Note that only the `script` element, rather than any script-supporting element is supported. The script-supporting elements in HTML5.1 includes the `template` element. However, the `template` element is also phrasing content. When using `%scripting;` (which includes both the `script` and the `template` element, and is used as parameter entity reference elsewhere in the HTML5.1 DTD), the grammar for `datalist` will become 1-ambiguous] ([5]). This means that upon encountering a `template` element in a `datalist` parent element, the parser cannot decide which of the two branches declared in the choice submodels of `datalists` grammar rule is to be selected for subsequent parsing. This is not permitted in SGML, and either disallowed or undesirable in other markup languages as well.

Semantically, it doesn't make sense to use `template` elements in `datalist` child content, hence the allowance of `template` is considered accidental (or a consequence of HTML5's grammar presentation which doesn't facilitate basic automated grammar checks).

2.4. Boolean Attributes

HTML5 lists the following as Boolean attributes ([6]): `reversed`, `ismap`, `typemustmatch`, `default`, `autoplay`, `muted`, `checked`, `readonly`, `required`, `multiple`, `disabled`, `selected`, `readonly`, `required`, `reversed`, `disabled`, `autofocus`, `autoplay`, `novalidate`, `formnovalidate`, `hidden`, `lang`, `async`, `defer`, and the `truespeed` attribute on the deprecated `marquee` element.

Note the `paused` attribute isn't a boolean attribute.

HTML5's boolean attributes are modeled as SGML attribute declarations having a singleton name group as declared attribute value, ie. an enumerated value where the name group contains only a single value.

For example, the `selected` (and `disabled`) attribute on HTML5's `option` element, according to the HTML5 specification, must be specified as `<option selected>`, and the HTML5 DOM API is supposed to treat the `selected` attribute as either `true` or `false`. If a `false` value is desired, the `selected` attribute must be omitted in an attribute specification.

In SGML, this is modeled as

```
<!ATTLIST option selected (selected) #IMPLIED>
```

meaning the attribute *name* can be omitted.

According to the declaration, specifying

```
<option selected>
```

is equivalent to specifying

```
<option selected=selected>
```

or

```
<option selected="selected">.
```

Note that, formally, WebSGML (ISO 8879 Annex K) allows use of the same name token as enumerated value for multiple attribute declarations. In prior versions of SGML, the following wasn't valid:

```
<!ATTLIST x a (true|false) #IMPLIED>
<!ATTLIST y a (true|false) #IMPLIED>
```

because the name tokens `true` and `false` could only be used in a single attribute declaration; one had to declare:

```
<!ATTLIST (x|y) a (true|false) #IMPLIED>
```

At the same time, pre-Annex K SGML only allowed a single attribute list declaration for a given element.

WebSGML relaxes this constraint by allowing

- declared attributes to be assembled from multiple `ATTLIST` declarations for the same element(s), and
- enumerated attributes (token name groups) to contain the same token in different attribute declarations (including on the same element); note that *if* a token is declared on multiple elements, it cannot be used with omitted attribute name

However, (Open)SP doesn't seem to implement Annex K in this respect, and will reject multiple `ATTLIST` declarations on the same element and also multiple declarations for the same name token.

While sgmljs.net SGML doesn't have this restriction, for interoperability, the HTML 5 DTD generator outputs the boolean attributes inline along with other attributes on an element.

2.5. The `contenteditable` and `spellcheck` attributes

The `contenteditable` and `spellcheck` attributes are handled special; these can have their values omitted in HTML5 but cannot be modeled in SGML analogously to *boolean attributes*, because they both use `true`/`false` as enumerated values, and thus can't be handled via SGML `MINIMIZE ATTRIB OMITNAME` (which requires that name tokens be unique among those declared for *all* attributes in a

DTD, not just those declared on a given element, in order to make use of OMITNAME).

2.6. Void elements

The HTML5 specification lists area, base, br, col, embed, hr, img, input, keygen, link, meta, param, source, track, and wbr as *void* elements.

Note that the HTML5 specification suggests that the (legacy) elements basefont, bgsound, and frame should also be treated as void, but these don't have declared content EMPTY in this HTML 5 DTD.

HTML5's void elements happen to coincide with those labeled as having the Empty content model in the section on individual elements in the specification text.

Void elements are expected to neither have child content nor an end-element tag, and are adequately modeled as SGML elements with declared content EMPTY.

In the HTML5 specification text, void elements are described as having "No end tag" (in addition to having "Nothing" as content); however, an element declared EMPTY in SGML usually isn't qualified with an end-tag omission indicator, since having declared content EMPTY isn't considered a tag minimization feature in SGML.

2.7. Self-closing elements

"Self-closing" elements are uses of HTML void elements which have a slash before the > (U+003E GREATER THAN SIGN) character (eg. the STAGC delimiter in SGML) in the start-element tag, such that void elements appear as XML-style empty elements.

For example, in:

```
<link href="..." rel="stylesheet" type="text/css" />
```

the / (U+002F SOLIDUS) character is bogus.

This syntax was used in the past to make HTML processable using XML parsers, and its use is generally discouraged.

While tolerated (ignored) by HTML5 on void elements, in the HTML5 SGML DTD, self-closing elements are subject to the EMPTYNRM YES and other settings in the SGML declaration for HTML5 ([7]) which synthesize HTML's parsing rules in this respect.

2.8. RAWTEXT and RCDATA

The child content of the style element is modeled as SGML CDATA declared content, meaning that any markup delimiters are ignored (up to the sequence of terminating characters as discussed in Script data).

Note that legacy HTML also might assume CDATA semantics with xmp, noembed, and noframes content.

The child content of the textarea element is modeled as RCDATA declared content, which behaves same as CDATA, except delimiters for entity references (SGML's ERO and ERE delimiters, ie. the U+0026 AMPERSAND and the U+003B SEMICOLON characters in the SGML reference concrete syntax, respectively) *are* recognized, and entity references formed by those are substituted by replacement text.

2.9. Script data

Dispensing with earlier DTDs for HTML declaring the content of the script element CDATA, in this DTD for HTML5.1, child content of the script element is modeled as (#PCDATA) content model for security reasons.

HTMLs script element and it's historic use has been known to be a problem since at least as early as 1996 (cf. Joe English's posts in [8].

According to the HTML5.1 specification,

1. after transitioning from *script data less-than sign state* ([9]) (which is the state reached in script element child data after having encountered an unescaped < (U+003C LESS THAN) character),
2. '/' (U+002F SOLIDUS) transitions the parsing state to the *script data end tag open state* ([10]), which in turn will be sent to the
3. *script data end tag name state* ([11]) over any ASCII character
4. In the *script data end tag name state* an HTML5 parser is supposed to check the longest sequence of ASCII characters for a case-insensitive match of script (and finish the script element if this is the case).

For SGML, on the other hand, expected behaviour is to end CDATA or RCDATA on a "delimiter-in-context" ie. < (U+003C LESS-THAN SIGN), followed by / (U+002F SOLIDUS) followed by a name start character (ignoring other irrelevant delimiter-in-context cases here), *irrespective of whether the generic identifier started by the name start character is actually* `script` (or, more generally, the same that started CDATA/RCDATA), whereas HTML5 is supposed to treat character data looking like an end-element tag but not actually representing a </script> tag as part of the character content of script.

For example, the following HTML fragment

```
<script>
    document.innerHTML =
        "<html><head><title>Oops</title><body>Pwnd</body></html>"
</script>
```

will be parsed by HTML as a single script element with child content, but, provided the script element has been declared

```
<!ELEMENT script CDATA>
```

will be parsed by SGML as

- the `<script>` start-element tag,
- the text `document.innerHTML = "<html><head><title>Oops`,
- the `</title>` end-element tag,
- the `<body>` start-element tag,
- the text `Pwnd`,
- the `</body>` end-element tag,
- the `</html>` end-element tag, and
- the `</script>` end-element tag.

While a DTD using either `CDATA` or `(#PCDATA)` for the `script` element will reject this particular sequence of markup events (because `script` can't have it's end-tag omitted), in general, this behaviour is undesired since it could be used to mount script injection attacks.

As explained in the HTML 5 specification ([13]), there's an additional, rather unnecesary, twist in HTML5's dealing with script data, in that what looks like starting an SGML comment (ie. the character sequence `<!--`) within script data will make the parser enter *script data escaped dash dash state* ([12]) which is only exited on a subsequent `-->` character sequence, potentially parsing well beyond what looks like the regular end of the script element.

That is, SGML's `<!--` and `-->` character sequences are recognized as JavaScript comment start- and end- sequences, respectively; presumably, this was an (ill-conceived) attempt in early JavasScript revisions to present uniform commenting syntax across HTML and JavaScript.

It is problematic since it is completely invisible to SGML. Needless to say, this style of script comments is an avoidable XSS attack vector in web pages.

SGML has never recognized comments in CDATA or RCDATA at all, hence this cannot be handled by SGML other than by using a regular `(#PCDATA)` content model.

Treating `script` content as `(#PCDATA)` can be inconvenient, since it requires that verbatim occurrences of the < (U+003C LESS-THAN SIGN) character might have to be specified using the `&lt;` entity, or that all or parts of the child content is put into `CDATA` or `RCDATA` marked sections.

If this turns out to be a problem, the declaration for `script` can easily be changed to `CDATA` to re-establish former behaviour.

2.10. Attribute defaults

This HTML5.1 DTD doesn't declare attribute defaults. Instead, it always declares `#IMPLIED` as default value.

Generally speaking, making subtle distinctions with respect to whether attribute (and other) defaults are specified with their default values in content explicitly as opposed to left unspecified is considered a bad practice, since whether an attribute is specified or implied isn't adequately represented in eg. DOM and similar APIs lacking attribute defaults and other type-related metadata. About the only applications in need of access to this kind of information are HTML authoring and developer tools.

However, the HTML5.1 specification recommends (specifically, for the ARIA attributes) to *not* specify their default values explicitly (ie. unless their actual value differs from the default).

2.11. Transparent content

Arguably, the most characteristic element of HTML is the anchor (`a`) element for hyperlinking. Apart from hyperlinks, the core HTML content models are merely variants of paragraph, flow, table, and other content models that were already in wide use for marking up documents for printing in the pre-WWW era.

In HTML5, the content model of the `a` element (and that of `map`, `ins`, and `del`) is specified as *transparent*, which means that `a` "inherits" (for lack of a better word) its parent content model: permitted child content is determined by the parent element's permitted child content (which can inherit its permitted content from its parent element, in turn, and so on).

HTML5's transparent content model concept is an artifact of adding the ability to annotate any piece of flow content as hyperlink, rather than just phrasing content as in previous HTML specifications. In practice, it is commonly used when eg. an image or icon and some belonging text (and possibly some background) is hyperlinked to a common target using a single `a` element, rather than having to wrap the image, the text, and boilerplate content into `a` elements individually. Note an extension for using `href` and other attributes of anchors on any HTML element (with the expectation that this makes those elements behave like hyperlinks, thereby effectively making the anchor element redundant), was proposed ([14]) already around 2008. Arguably, had this been further pursued, the HTML vocabulary could have been made much simpler and more orthogonal, but it was rejected at the time on the grounds that browser vendors already had implemented the ability to place anchor tags around most HTML elements.

As applied in the HTML specification, transparent content just means that eg. an `a` element accepts either just phrasing or also flow content as child content, depending on whether it appears in a flow or phrasing context, respectively (and similarly with `map`, `ins`, a `del`). The concept of transparent content, however, doesn't extend to arbitrary elements, because it trivially conflicts with the content model descriptions of those elements into which elements having transparent content can be placed. Specifically, for any element which accepts an element hav-

ing transparent content, it needs to be stated whether, and how, elements wrapped into transparent content-allowing child elements should contribute to the parent's content model.

Another formulation for the constraints imposed by the a element's transparent content restriction is that an HTML document can be validated by removing all a start-element and end-element tags (but keeping child content of a elements), and validating the result document against a tight HTML5 grammar lacking an a element. In sgmljs.net SGML, this notion can be directly expressed using SGML LINK, ie. by declaring an explicit link process projecting a permissive variant of HTML as source markup into a restrictive HTML variant as result markup, and by declaring a link rule that maps all source elements to the same-named target elements, respectively, except for a and other HTML elements with transparent content.

From a practical point of view, though, to facilitate HTML validation using mainstream SGML parsers (which don't support SGML LINK and/or don't perform validation and tag inference on result markup events of link processes), it might be desirable to express the effective content model restrictions imposed by transparent content using DTD declarations.

Fortunately, it can be easily shown that HTML's a element (and also HTML's other elements having transparent content) behave in a tame and modular way that doesn't interact with the content model into which an a element is placed:

- Since a is member of the flow and phrasing element categories, and the content model declarations of HTML only ever use a elements as part of flow or phrasing content, rather than as a element in isolation, and the flow content and phrasing content productions are interpreted as "any sequence of the respective elements, or the empty sequence", a can only be used as an optional content token, hence can't be put into a content position in such a way that it changes the interpretation of content model tokens with respect to validation and tag inference.
- Since the flow and phrasing categories (with the exception of the p element which is covered below) only contain elements which have either void content (ie. have declared content EMPTY in SGML parlance), or don't admit tag omission, flow and phrasing content (up to p elements) is always fully-tagged markup without omitted tags. Hence, markup wrapped into a elements can't alter the interpretation of neighbouring content (and the effect of omitting a p end-element tag within an a element, even if it were allowed, can't interact with neighbouring content either, since the a element doesn't admit tag omission).
- The p element is the only element in flow content that admits end-tag omission, and hence could be seen to interact with the placement of an a element in a non-modular fashion. The HTML5 specification addresses this problem spe-

cifically in the content model description for the `p` element (by disallowing `p` end-tag omission in child content of `a`).

The *transparent content* constraint is inherent in the fundamental construction of the HTML vocabulary as phrasing (in-line) content wrapped in flow (block-level) content, and already sufficiently represented in this HTML5 DTD via exclusion exceptions as discussed above.

3. Limitations

As discussed in the full paper, the following SGML limitations impede full HTML 5.1 parsing support:

- the set of permissible characters for ID values and value references cannot be modeled tightly without being overly permissive with respect to other name tokens such as element and attributes at the same time
- HTML5.1 admits transparent element content only in descendant elements of a single content model token, rather than as descendant content of any element in the respective parent content model, exceeding the expressiveness of SGML exclusion/inclusion exceptions
- as already mentioned in the introduction, HTML script data parsing cannot be implemented in the exact same (historic and arguably broken) way as HTML.

While the second issue is considered insignificant, the first and third issues have been addressed by formulating a proposal for a future revision of ISO/IEC 8879.

4. Conclusion

The HTML5.1 DTD developed here can demonstrate fitness for its intended purpose by being able to parse HTML in the same way as W3C's validation software on test cases for tag omission and other HTML syntax features discussed in the text. Moreover, the large HTML5.1 specification text itself can be parsed using the HTML5.1 DTD, with only minor modifications.

The DTD has been tested with both (Open)SP and sgmljs.net SGML.

This also demonstrates that, while nominally not based on SGML, owing to HTML requiring legacy compatibility, HTML5 hasn't striven far from its SGML roots, containing numerous characteristics traceable to SGML idiosyncrasies. In those cases where it has, such as in its definition of ID values and custom data attributes (as explained in the extended version of this paper) HTML5 has introduced accidental inconsistencies of its own making, though WHATWG's newer custom elements specification shows a saner approach as far as the definition of the basic lexical properties of custom element names are concerned.

This initial HTML5.1 DTD is published in the hope those who continue to see value in a formal grammar for the world's most used markup vocabulary will contribute to its further development.

In future work, it's intended to put this DTD to work for parsing larger existing web content corpora and gaining practical experiences in its application, starting with the web platform test suite maintained by W3C (as well as various other test case repositories maintained by W3C, WHATWG, Mozilla, and others on github). However, its alignment with the HTML 5.1 specification, with the W3C validation software, and its status and endorsement by browser vendors is unclear to the author.

The author thanks the XML Prague 2017 reviewers for valuable feedback.

Bibliography

[1] S. Faulkner et. al. (ed.) HTML 5.1 W3C Recommendation, 1 November 2016 https://www.w3.org/TR/html51/

[2] I. Hickson et. al. (ed.) HTML 5 conformance classes https://www.w3.org/TR/html5/single-page.html

[3] Nu Html Checker https://validator.w3.org/nu/

[4] M. Reichardt HTML5.1 DTD Reference http://smgljs.net/docs/w3c-html51-dtd.html

[5] A. Brueggemann-Klein, D. Wood One-Unambiguous Regular Languages http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.4.3440

[6] I. Hickson et. al. (ed.) HTML5 Boolean attributes https://www.w3.org/TR/html5/single-page.html#boolean-attribute

[7] M. Reichardt SGML declaration for HTML5.1 http://sgmljs.net/docs/w3c-html51-sgmldecl.html

[8] J. English Cougar DTD: Do not use CDATA declared content for SCRIPT (on the www-html mailing list, July 1996) http://xml.coverpages.org/rcdataCTS1.html

[9] I. Hickson et. al. (ed.) HTML5 Tokenization - Script data less-than sign state https://www.w3.org/TR/html5/single-page.html#script-data-less-than-sign-state

[10] I. Hickson et. al. (ed.) HTML5 Tokenization - Script data end tag open state https://www.w3.org/TR/html5/single-page.html#script-data-end-tag-open-state

[11] I. Hickson et. al. (ed.) HTML5 Tokenization - Script data end tag name state https://www.w3.org/TR/html5/single-page.html#script-data-end-tag-name-state

[12] I. Hickson et. al. (ed.) HTML5 Tokenization - Script data escaped dash dash state https://www.w3.org/TR/html5/single-page.html#script-data-escaped-dash-dash-state

[13] I. Hickson et. al. (ed.) HTML5 - Restrictions for contents of script elements https://www.w3.org/TR/html5/single-page.html#script-content-restrictions

[14] E. Meyer Any-Element Linking Demo http://meyerweb.com/eric/thoughts/2008/07/23/any-element-linking-demo

The X-definition 3.1

Jiří Kamenický
Syntea software group a.s.
<jkamen@syntea.cz>

Jindřich Kocman
Syntea software group a.s.
<kocman@syntea.cz>

Václav Trojan
Syntea software group a.s.
<trojan@syntea.cz>

1. Preamble: What is The X-definition?

The X-definition is a programming language designed for description of the structure of an XML document, its validation, processing and even construction. The X-definition itself is an XML document. The content of the X-definition is composed of models of elements. As it has been already several years since our last presentation let us recollect in few words what the X-definition is.

1.1. Model of element

The design of a model of an element in the X-definition is derived from the structure of the XML element data to be described. The text values of attributes or text nodes are replaced by the X-definition "script". The occurrence of elements in the model is described by the script in auxiliary attribute `xd:script` (`xd` is the prefix of namespace of the X-definitions). Let's illustrate it on the following example. Let's have the XML data:

```
<books date="2017-01-21">;
  <book isbn="123456789" published="2017">
     <title>The X-definitions 3.1</title>
     <author>Jiří Kamenický</author>
     <author>Jindřich Kocman</author
     <author>Václav Trojan</author>
  </book>
  <book isbn="987654321">
     <title>The Bible</title>
  </book>
</books>
```

The model of element "books" looks like:

```
<books date="date()">0
  <book xd:script="+"
    Isbn     ="  int()"
    published="? gYear">
    <title>string()</title>
    <author xd:script="*">string()</author>
  </book>
</books>
```

The X-definition complete is an envelope which contains one or more models:

```
<xd:def xmlns:xd="http://www.syntea.cz/xdef/3.1"
           xd:root="books" xd:name="Bookstore">
  <books date="dateTime()" ... see model above
</xd:def>
```

Note the script contains information about occurrence of items and specifies a method used for validation of input data. This is a "validation" section of the script. However, the script may contain other sections with the code invoked on different events during the processing. Since the result of the validation may be true or false you can specify what to do in such situation eg. in the section `onTrue` or `onFalse`. An important feature of the X-definition is that the code may refer to a method connected to X-definition from external Java environment. This allows to provide the validation check in a database etc.

2. Processing of large data

If you have a large number of books and you have not got enough memory you can do several things.

1. To ignore all irrelevant items in the process by specifying that they should be ignored:

   ```
   <book xd:script="+; "
   isbn="int()" published="ignore gYear()" >
     <title>string()</title>
       <author xd:script="ignore">string()</author>
   </book>
   ```

   The result will be a XML document containing only attribute "isbn" and the element "title" from each book. the items specified as `ignore` are simply to be ignored and they do not appear in the final document.

2. To release any book from the memory after it was processed by specifying the action `forget` to the script of the book:

   ```
   <book xd:script="+; forget"
   isbn="int">
   ```

All books will be removed from the result document after being processed. However, the "forgotten" items are processed before they are removed from the memory (i.e. we still have information about errors etc.).

3. Sometimes it might be useful to ignore all items you do not need. You simply omit description of those objects you do not need if you specify the option `ignoreOther` in the script of the model of an element:

```
<books date="dateTime()">
  <book xd:script="+; option ignoreOther">
    <title>string()</title>
    <author xd:script="*">string()</author>
  </book>
</books>
```

In this case the result will contain only the data described in the model. Other data from the source XML will simply be ignored.

3. The stream mode

If you need to keep large processed data, you still have the possibility to store the data to an output stream before you release it from the memory. The result document will contain only such elements, which are not "forgotten". However, the output stream will contain the full processed data (of course, not those ignored items). The data is written to the output stream before it is released.

The output stream is assigned externally when the X-definition is invoked:

```
<books date="dateTime()">
  <book xd:script="+; forget"...,
...
```

In the real project you usually know which data you can "forget" after it was processed or even ignore it. The description in the X-definition is very transparent and simple.

4. Processing of JSON data

The main problem of processing JSON data in the X-definition is to provide fully reversible conversion of JSON data to XML and vice versa. There are following problems:

1. Conversion of the named items in the JSON map to XML names. Since the name of the mapped item is any kind of string (even the empty one), A convertor was designed which replaces the character of JSON name which is not acceptable in Java name to the sequence "_uX_" where "X" is a hexadecimal representation of such a character in the UTF character set. The empty string is represented by the sequence "_u0_".

2. Simple JSON values can be a number, string, "true", "false" or "null". The convertor respects the idea of readability and simplicity: if the string doesn't start with numbe , boolean or "null" and if it doesn't contain whitespaces r quotation marks, it is converted to the character sequence without quotation marks, otherwise the quotation marks are added. There are implemented special validation methods for description of JSON format of values.
3. Each named item can be converted to an attribute or element. In order to minimize XML structure and to improve its readability, it is preferred to create an attribute rather than an XML element. If the element is generated, it is inserted to an auxiliary element "MapExtension" with the namespace of JSON.
4. JSON array is converted to an auxiliary element "js:Array". However, since the sequence of an XML element can be considered as an array, the element "Array" is generated only when it must be generated (e.g. array in array).

Let's have data from the example above in a JSON format:

```
{ "books": [
    {"book": {
        "isbn": 123456789,
        "title": "X-definition 3.1",
        "edited\": 2017,
        "author": ["Jiří Kamenický", "Jindřich Kocman",
                   "Václav Trojan"

        }
      },
      {"book": {
        "isbn": 987654321,
        "title": "The Bible"
      }
    }
  ]
}
```

In the first step the X-definition model is generated from the JSON data:

```
<books
xmlns:js="http://www.syntea.cz/json/1.0">
  <js:array>
    <book edited="jnumber()"
          isbn  ="jnumber()"
          title ="jstring()">
      <js:array>
        <author xd:script="+"> jstring() </author>
      </js:array>
    </book>
    <book isbn="jnumber()" title="string()"/>
```

```
  </js:array>
</books>
```

Now we can optimize the above X-definition by adding occurrences of items a by deleting the redundant ones. The result will be:

```
<books>
  <book xd:script="+"
        edited="? jnumber()"
        isbn  ="  jnumber()"
        title ="  jstring()">
     <author xd:script="*"> jstring() </author>
   </book>
</books>
```

Note the algorithm prefers to generate for a unique named value an attribute rather than an element with a text value. Note also, that they were removed from model the items "js:array".

We here described the idea of XML/JSON processing. The tool is still in the development stage. We have now the experimental "alpha" version and we are opened to discuss it.

5. Processing of errors

1. The result of the methods used for parsing of text values of attributes or text nodes is an object containing the parsed value or an error report. The instruction of what to do when an error was detected can be added to the script of an attribute or a text node.
2. The processor of the X-definition is storing error messages to MessageReporter. Each message is an object containing the type of message (e.g. error), the message identifier, the template text and the parameter list (used to modify the template text). The reporter contains information counters of the reported message types. In the script of the X-definition you can get the number of error messages reported during the model processing. This enables to process messages reported during processing of any model and to invoke a method associated to an error.

   ```
   <book xd:script="+; finally
   if (errCount() > 0) {...}" ...
   ```

3. The method used for parsing of text values may be invoked in an external Java environment. So it is possible to check if the value is in a database or a code list. The external method can be declared in the heading of the X-definition.

   ```
   <xd:def ...
     xd:methods="XDParedResult project.books.checkISBN(XXData);
                 void project.books.errorISBN(XXData);">
   ```

```
...
<book isbn="checkISBN()"; onFalse errorISBN(); ...
```

In the applications designed in Syntea company most of the errors are processed in the external Java methods.

6. Connection to database

The X-definition provides a connection to different kinds of databases. Therefore three kinds of objects are implemented there:

1. Service. The Service assures the connection to a database. The database engine may be a relational database or XML database. In a relational databases it is `java.sql.Connection` (in a XML database eg. in the eXist project see `org.xmldb.api.modules.XQueryService`). Since Service provides access to the database according to the rights assigned to a user, the service object is recommended to be passed to the X-definition processor from an external environment as an external variable.
2. Statement. The Statement contains a prepared database statement. The processor of the X-definitions executes a database statement via this object. You can get the Statement object when you invoke the method `prepareStatement` on the object Service. The object Statement has the method `execute` which provides required database instruction and returns the result in the object ResultSet. Also there are available methods `prepareStatement`, `query` or `queryItem` etc..
3. Resultset. This object behaves as an iterator and returns the rows from the table which was returned as a list of rows. In relational database each row of the table is interpreted as an element containing attributes with the names of table columns and its values. In the case of XML database it may be a sequence containing more complex structures created by the statement execution.

Let us remember that the X-definition can be executed in two modes. So far we have described the mode we call "processing" mode. In this mode the X-definition parses input data, and returns the processed data as a result. In this mode we can use the database for checking process data (eg. to check if a value is in the database or to store it to a database etc.). The second mode that we call "construction" mode works in inverse way: it creates the result according to the X-definition. In the X-definition we can describe how to cconstruct the result. We simply add to the script the section `create`. In the processing mode this section is ignored. However, in the cconstruction mode this section is executed. Let s show it in the example of the model:

```
<date> dateTime(); create now();
</date>
```

In the processing mode it is checked if the element `date` contains the text value with the valid date and time value. On the contrary in the construction mode the text value is created from the string generated by implemented value `now`.

We can create data from an outside source. Eg. let's create a li t of titles of books we have in XML data from our example above and we pass it to the X-definition from an outside environment as the XML element. The mehod `xpath` returns a list of nodes which is in the X-definition as an iterator. The items of such iterator are used as the context data from which the relevant items of the required result will be created:

```
<xd:def xmlns:xd="http://www.syntea.cz/xdef/3.1"
           xd:name="inventory">
  <xd:declaration>
    external Element source;
  </xd:declaration>

  <Inventory>
    <Book xd:script="*; create xpath('//Book', source)"
          isbn="int()">
      <Title>string();</Title>
    </Book>
  </Inventory>
</xd:def>
```

If we have the database with books we can create the a of titles of books with the database query statement (the password is passed on to the X-definition from an outside environment):

```
<xd:def xmlns:xd="http://www.syntea.cz/xdef/3.1"
           xd:name="inventory" >
  <xd:declaration>
    String database = "jdbc";
    String url = "jdbc:derby://localhost:1527/sample;";
    String user = "app";
    external String password;
    Service service = new Service(database, url, user, password);
    Statement statement = service.prepareStatement(
      "SELECT TITLE,ISBN FROM MYTEST.TITLE ORDER BY TITLE ASC");
  </xd:declaration>
  <Inventory xd:script="">
    <Book xd:script="occurs *; create statement.query()" isbn="int()">
     <Title> string();</Title>
    </Book>
  </Inventory>
</xd:def>
```

Note that the items "isbn" and "Title" are created from columns "ISBN" and "TITLE" from the database table. The names of the columns are processed as case insensitive.

The complete example is available on http://xdef.sy ntea.cz/tutorial/en/user-doc/DBExample.pdf[1]

7. X-components

X-components are an extension of the X-definition for simplification of Marshalling/Unmarshalling without the need any external library. The X-component is a standard java class implementing interface `cz.syntea.xc.XComponent` with a structure that corresponds to the one element in the X-definition. Every element in the X-definition has one X-component as an image and its attributes are expressed as class fields. The value of attributes, elements and text values are accessible by `getName()` and `setName()` where `Name` is the one of the attribute or the element in the X-definition.

Example:

```
<xd:def  xmlns:xd="http://www.syntea.cz/xdef/3.1"
         xd:root     = "Insurer"
         xd:name     = "Insurer">
    <Insurer
          Company       = "required string()">
        <Contract      xd:script="*"
              ContractID    = "required int()"
              ValidFrom     = "required datetime('yyyyMMdd')">
            <AdditionalInformation xd:script="?">
                string()
            </AdditionalInformation>
        </Contract>
    </Insurer>
</xd:def>
```

Corresponding X-components

```
package
cz.syntea.xmlprague.example;
public class Insurer implements cz.syntea.xc.XComponent{
  public String getCompany() {return _Company;}
  public java.util.List<Insurer.Contract> listOfContract() {
    return _Contract;
  }

  public void setCompany(String x) {_Company = x;}
```

[1] http://xdef.syntea.cz/tutorial/en/userdoc/DBExample.pdf

```
  public void addContract(Insurer.Contract x) {
    if (x!=null) _Contract.add(x);
  }

  private String _Company;
  // Constructors and implementation of cz.syntea.xc.XComponent

  public static class Contract implements cz.syntea.xc.XComponent{
    public Long getContractID() {return _ContractID;}
    public cz.syntea.common.sys.SDatetime getValidFrom() {return
_ValidFrom;}
    public
cz.syntea.xmlprague.example.Insurer.Contract.AdditionalInformation
      getAdditionalInformation() {return _AdditionalInformation;}

    public void setContractID(Long x) {_ContractID = x;}
    public void setValidFrom(cz.syntea.common.sys.SDatetime x) {
      _ValidFrom = x;
    }
    public void setValidFrom(java.util.Date x) {
      _ValidFrom=x==null ? null : new cz.syntea.common.sys.SDatetime(x);
    }
    public void setValidFrom(java.sql.Timestamp x) {
      _ValidFrom=x==null ? null : new cz.syntea.common.sys.SDatetime(x);
    }
    public void setValidFrom(java.util.Calendar x) {
      _ValidFrom=x==null ? null : new cz.syntea.common.sys.SDatetime(x);
    }
    public void setAdditionalInformation(
      cz.syntea.xmlprague.example.Insurer.Contract.AdditionalInformation ▶
x)
{
        _AdditionalInformation = x;
      }

    private Long _ContractID;
    private cz.syntea.common.sys.SDatetime _ValidFrom;
    private
cz.syntea.xmlprague.example.Insurer.Contract.AdditionalInformation
      _AdditionalInformation;

    // Constructors and implementation of cz.syntea.xc.XComponent

    public static class AdditionalInformation implements
                                        cz.syntea.xc.XComponent{
      public String get$value() {return _$value;}
```

```
        public void set$value(String x) {_$value = x;}
        private String _$value;

        // Constructors and implementation of cz.syntea.xc.XComponent

      }
    }
  }
```

The interface X-component contains a method for conversion to and from XML. When parsing XML instances of X-components are created and filled data. Every element and attribute has its XPath to the source XML it was created from. The X-component has a method toXML() for conversion the whole subtree to `org.w3c.dom.Element`. With non-parametric constructor the instance can be created by hand filled data programmatically (using this method the source XPath is null).

7.1. Generating X-componets

X-components are generated automatically from the X-definition. Commands for creating X-components are written in the section `<xd:component>` as a part of the source X-definition. The most important command is `%class` which creates the X-component itself. This command contains a fully-qualified name of the resulting object and XPath to the source X-definition pointing to the element it should be created from. Every nested element is created as an inner class of the generated X-component unless it has its own `%class` command. The `%class` command for the above example looks like:

```
<xd:component>
    %class cz.syntea.xmlprague.example.Insurer %link Insurer#Insurer;
</xd:component>
```

Using this command it is also possible to pass information about interfaces and superclass generated class should implement and extend.

Sometimes is necessary to use getter and setter from superclass and do not to generate it. For this case it exists a command `%bind` which suppresses creation of getters and setters used by another method. The name of the method can be arbitrary. Using this command you can also use a different name in the X-definition and in the X-component.

8. The X-definition and the X-components in real-world project

The X-definition technology is used in large systems in the insurance industry in the Czech Republic and the Slovak Republic. Due to usage of the X-definition all interfaces are defined, validated and processed with external entities, which have

the form of web services or the form of a file. The X-definition and X-components are also used for all internal communication of program modules. The number of different types of messages used in the application is nearly 300. The total number of external messages processed by these systems are hundreds of millions per year.The X-definition is also used in these systems for validating and processing of metadata that describe the internal properties of program modules. An example of such a X-definition for the module layer of web services is:

```
<xd:def xmlns:xd="http://www.syntea.cz/xdef/3.1"
        impl-version ="1.0.1_3"
        impl-date="29.9.2016"
        xd:name        ="DefModul_VVR"
        xd:root        ="DefModul_VVR" >

 <xd:declaration>
   type Activity           string(1,30);
   type Bool               list('Y','N');
   type ChannelType        string(1,20);
   type ChannelTypeVersion string(1,30);
     :
   type Variant            list('SYN','ASY');
   type Version            string(1,10);

   uniqueSet subsystemSet    {subsystem:   Subsystem()};
   uniqueSet channelTypeSet  {channelType: ChannelType()};
   uniqueSet versionSet      {channelType: ChannelType();
                                 version: Version()
                             };
   uniqueSet msgNameSet      {channelType: ChannelType();
                                 version: Version();
                                 msgName: MsgName()
                             };
   uniqueSet partnerTypeSet  {partnerType: PartnerType()};
    :
 </xd:declaration>

 <DefSystem SystemCode          ="  SystemCode()"
            SystemLocation      ="  SystemLocation()"
            Version             ="  Version()"
            LastVersionMin      ="  Version()"
            LastVersionMax      ="  Version()"
            ImplTime            ="  ImplTime()"
            Description         ="? Description()"
            Mode                ="  Mode()"
            PartnerCode         ="  PartnerCode()">
    <DefSubsystems         xd:script="1;   ref DefSubsystems"/>
```

```
    <DefChannelTypes       xd:script="1;   ref DefChannelTypes"/>
    <DefPartnerTypes       xd:script="1;   ref DefPartnerTypes"/>
    <DefPartnerChannels    xd:script="1;   ref DefPartnerChannels"/>
    <DefPartners           xd:script="1;   ref DefPartners"/>
    <DefChannels           xd:script="1;   ref DefChannels"/>
    <DefPrograms           xd:script="1;   ref DefPrograms"/>
    <DefPoints             xd:script="1;   ref DefPoints"/>
    <DefActivities         xd:script="1;   ref DefActivities"/>
    <DefFlows              xd:script="1;   ref DefFlows"/>
 </DefSystem>

 <!-- Definition of subsystems -->
 <DefSubsystems>
    <DefSubsystem     xd:script = "0.."
         Subsystem           ="  subsystemSet.subsystem.ID()"
         Description         ="? Description()"
         Status              ="? Status()"
    />
 </DefSubsystems>

 <!-- Definition of Channel types -->
 <DefChannelTypes>
    <DefChannelType     xd:script="0.."
         ChannelType         =" channelTypeSet.channelType.ID
                                  (versionSet.channelType(
                                     msgNameSet.channelType()));"
         Direction           ="  Direction()"
         HasCertificate      ="  Bool()"
         Description         ="? Description()"
         Status              ="? Status()">
       <DefVersion     xd:script="1..; ref DefVersion"/>
    </DefChannelType>
 </DefChannelTypes>

 <DefVersion
         Version             ="
versionSet.version.ID(msgNameSet.version());"
         ImplTime            ="  ImplTime()"
         Disable             ="? DisableStatus()">
    <DefMsg     xd:script="0.."
         MsgName             ="  msgNameSet.msgName.ID()"
         Method              ="  MsgMethod()"
         ProcessLog          ="  ProcessLog()"
         InputLog            ="  Bool()"
         OutputLog           ="  Bool()"
         TimeLimit           ="  TimeLimit()"
```

```
         Description         ="? Description()"
      />
   </DefVersion>

   <!-- Definition of Partner types -->
   <DefPartnerTypes>
      <DefPartnerType        xd:script="0.."
         PartnerType          ="  partnerTypeSet.partnerType.ID()"
         Description          ="? Description()"
         Status               ="? Status()"
      />
   </DefPartnerTypes>
    :
</xd:def>
```

Relational and Semantic Views over Documents

Normalization Considered Abnormal

John Snelson
MarkLogic Corporation
<john.snelson@marklogic.com>

Abstract

SQL is the norm. The relational model has had 45 years of dominance in database users hearts and minds, and has seeded a vast database tools, BI, and ETL market. Whatever your thoughts on the database market, it's hard to escape the ubiquity of SQL and the relational model.

However increasingly developers are learning to embrace the benefits of document databases, and understand the advantages of hierarchical and ordered data models. Returning to more natural data modelling concepts like entities and relationships, they are rightly beginning to view third normal form as distinctly abnormal.

But if my logical entity is represented by a document, how do I use it from SQL? I may wish to use BI tools on my data, or allow colleagues without a document database background access to it. I may find the uniformity and strong mathematical foundation of querying using relational algebra compelling. Similarly, I may wish to expose some of my data as RDF for use in data integration or Semantic Web projects.

The solution can be found in a declarative live transformation into the target data model (relational or RDF), which is always kept up-to-date - where the data is kept in a logical document model, and exposed through views as more query, domain, or user friendly structures.

1. Introduction

This paper discusses different data modelling techinques, designed with different purposes in mind, and with different strengths and weaknesses. It demonstrates a simple technique for combining the use of these data models, as well as discussing why you might want to do that. Although the techinique is simple, its ramifications for how data can be modelled and used are far-reaching, allowing multi-model and document databases to step into the wider context of DBMSs more effectively.

2. Data Model Overviews

2.1. The Entity-Relationship Model

An entity is defined as a thing that can have independent existence and can be uniquely identified. With this definition, the concepts of entities and relationships between those entities can be used to build information models that are very similar to the way that humans think about information. Entity-relationship models are often used as the first step towards reaching a relational data model. Instances of these natural organisational methods can be seen in many places where data has been modelled.

2.1.1. Hypertext: Documents and Links

The internet became what it is today because at its heart it consists of a way to organise, present, and navigate information that is very natural. The document is an intuitive way of creating knowledge for humans, and the links allow discovery of more depth and related topics as and when that becomes interesting. In many cases this maps well to an entity-relationship view of data modelling.

2.1.2. RDF Enhancement

"Resource Description Framework" was created to describe just the kind of documents or entities that exist on the internet. Adding RDF to web pages enhances them from just being human readable documents to being machine readable as well, exposing the data inside those documents. RDF also creates more value from the links in documents, by allowing those links to be described and classified, so that their semantics are understood.

2.2. The Relational Model

The relational model is a well-used alternate model for describing information, with closer ties to the mathematical concepts of first order predicate logic. Data is represented as tuples (rows), which are themselves grouped into relations (tables).

Although it has a strong mathematical foundation, the relational model is often less natural for expressing information. A lack of sequence or set support makes it unlikely that most entities can be expressed as a single row, forcing the data modeller to introduce new entity boundaries where natural entities do not exist. Heirachy inside entities is particularly hard to describe using a relational data model.

2.2.1. Normalization

Normalization has arisen as a technique to codify how to express more complex entities in the relational model. The techniques of database normalization describe how to avoid multi-valued attrbutes (sequences and sets), and duplicated information. However normalization creates fragmentation, where the information for a single conceptual entity is split over many separate rows in different tables. This can produce a rather unintuituve (even ab-normal) data model, where re-constituting conceptual entities requires costly query-time joins.

2.2.2. Extract, Transform, and Load

Relational modelling can often require the use of ETL tools to perform the shredding of the entity information into the correct tables. This transform process will often result in the loss of data - both because the data has not been modelled in the relational schema, and because the original context of the data is not directly stored.

2.3. The RDF Data Model

The RDF data model breaks all information down into very simple triples, with a subject, predicate, and object. As well as modelling intra-entity relationships, this can also be used to describe attributes of entities using literal values in the object position, or more complex heirachical information using blank nodes and further sub-trees.

RDF does not contain a good way to model sequences, resorting to a linked list design that is inefficient and hard to query. The way that RDF is almost super-normalized in the process of breaking information down into very small "facts" or triples means that queries to reconstitute an entity can involve even more joins, and be even less efficient than the relational data model.

2.4. The Document Model

The document model is a short hand for expressing information in a heirachical, ordered format - usually XML or JSON. Although there are significant differences between the expressivity and capabilities of XML and JSON, they are more like each other than they are like a relational data model or RDF data model.

Documents are often written first for humans, and as a by-product usually create a data model that is more comprehensible. They are also often more suitable for displaying to the end user of an information application built on the data, as the entity boundaries are often chosen for exactly that purpose - making this kind of "human first information modelling" a useful way to design entities for the entity-relationship model. However both XML and JSON lack a native way to

model relationships or document links, falling short of fulfilling the entire entity relationship model.

3. Reaching Data Model Nirvana

Since databases based on an entity-relationship model don't exist, it would be good to see how to get closer to this more natural way of modelling data using a document database like MarkLogic - which is already very flexible and human oriented in the way it can store data.

3.1. Document First Data Modelling

Entities map to documents well, which allow heirachy and sequences present in the properties of real world complex entities. They also allow the schema variability necessary to evolve many applications, and continue to on-board new types of data. Many entities exist in document form already, and a document database allows you to maintain the original knowledge organization. For these reasons, starting with a "document-first" data model gives a good foundation to build out a database capable of more closely approximating the entity-relationship model.

3.2. Document Links

However document databases often don't support document links directly, which is a barrier to utilizing the full entity-relationship data model. In a database like MarkLogic that also supports RDF storage, this best-of-breed relationship modelling can be used on top of the documents.

3.3. Business Intelligence and Analysts

There exist a good many reasons why organizations are reticent or unable to move away from their incumbant relational data modelling. A major factor are the Business Analysts employed in large numbers, who are well versed in using SQL to query company data. In addition the vaste majority of Business Intelligence tools (such as Tableau and Qlik) work best or exclusively by talking SQL over ODBC to the company's databases.

4. Template Driven Extraction: A Simple Solution

MarkLogic 9 will introduce a feature called "Template Driven Extraction", which allows simple templates to be defined that declaratively describe how to convert a document into rows for tables, or into triples. This process happens at document indexing time, so that the tables and triples are populated as soon as the document is committed to the database, in a transactional fashion. In addition,

the rows and triples are treated like any other index - ie: they are removed when the base document is removed, updated when it is updates, etc.

The templates are deliberately kept very simple, as potentially hundreds of templates might exist and need to be considered during document insertion. This is not an XSLT replacement! Indeed they are inspired quite heavily by XSLT, and use XPath as a pattern and query language.

4.1. Relational Lens

A simple template to define a relational view or "lens" over a type of document consists of a context path followed by one of more column definitions. For every instance of a node matching the context path, one or more rows will be created.

```
<template xmlns="http://marklogic.com/xdmp/tde">
  <context>/match</context>
  <rows>
    <row>
      ...
    </row>
  </rows>
</template>
```

Rows are specified with the schema and view (table) name it belongs to, along with one or more column definitions.

```
<row>
  <schema-name>soccer</schema-name>
  <view-name>matches</view-name>
  <columns>
    <column>
      ...
    </column>
  </columns>
</row>
```

A column has a name and a fixed type. The XPath expression in the "val" element uses the node matched by the template's context expression as its XPath context item. The result of the expression is automatically cast to the correct type if possible. If the expression raises an error or the cast fails, then the whole row is not added to the table. Columns can also be specified as nullable - nullable columns are left blank in the event of an error, rather than excluding the entire row from the table.

```
<column>
```

```
    <name>id</name>
    <scalar-type>long</scalar-type>
    <val>id</val>
  </column>
  ...
```

Templates are inserted into the "Schemas" database. The `tde:template-insert()` function performs validation of the template before inserting it into the "Schemas" database, as well as adding it to the TDE collection which makes the template available to the indexer.

```
tde:template-insert(
 "/test/my-first-TDE.xml" ,
 $my-first-TDE)
```

All templates are automatically discovered and used to index the documents in the content database. Existing documents will be reindexed if a new template is inserted that applies to them.

A template implicitly declares the definition of the view (table) it populates. This definition can be seen using the `tde:get-view()` function.

```
tde:get-view ( "soccer", "matches" )
->
{
  "view":{
    "id":"8206293418083457271",
    "name":"matches",
    "schema":"soccer",
    "viewLayout":"identical",
    "columns":[
      {
        "column":{
          "id":"1695951276079343631",
          "name":"id",
          "scalarType":"long",
          "nullable":false
        }
      },
      ...
    ]
  }
}
```

More than one template can contribute to the same view. In this case, the column definitions for the two templates have to be compatible.

Templates also support sub rows - a definition of a row that is dependent on the outer row. This is useful, for instance, for dealing with repeating structure inside an entity - where that repeating structure might populate one or more rows in a dependent table.

4.2. Triples Lens

Templates can also be used to extract RDF statements or triples from documents. In this case the template will still start with a context path that needs to be matched for the triples to be produced.

```
<template xmlns="http://marklogic.com/xdmp/tde">
  <context>/match</context>
  <vars>
    <var>
      <name>EX</name>
      <val>"http://example.org/ex#"</val>
    </var>
  </vars>
  <triples>
    <triple>
      ...
    </triple>
  </triple>
</template>
```

Templates can also contain definitions for variables which can be accessed from inner XPath expressions. The "name" element specifies the name of the variable, and the "val" element specifies an XPath expresion which is evaluated to get the value for the variable.

The definition of a triple to create specifies XPath expressions to return the subject, predicate, and object for the triple. In some cases these can be literal values, whilst in other cases they can depend on information from the indexed document such as the document URI - which is a useful way to link documents into the RDF graph.

```
<triple>
  <subject>
    <val>sem:iri( xdmp:node-uri(.) )</val>
  </subject>
  <predicate>
    <val>sem:iri( $EX || "hasID" )</val>
  </predicate>
  <object>
```

```
      <val>xs:integer( id )</val>
    </object>
  </triple>
  ...
```

5. Discussion of Benefits

5.1. Updating

One benefit of using templates is that although the same information can appear in a document, multiple tables, and a triple, only the document needs to be updated to change that information - and those changes happen transactionally. Compared to an alternative strategy of using distinct document, relational, and triple databases, this significantly simplifies interactions with your data.

5.2. Heterogenous Documents, Homogenous Tables or Triples

MarkLogic users often find they are solving problems by bringing together multiple sources of data. The data may be about the same kinds of entities, but it is almost always in different formats. Templates can be used to aid data integration in this scenario - to integrate a new source of data into an existing application, all that is needed is to write a new template for the new type of data - adding it into common tables or using common RDF predicates that the applications can already understand.

5.3. NoETL

By removing the need for extraction and transformation before loading, the original data can kept rather than thrown away. This brings the possibilities of progressive enhancement of the relational or RDF data model at future stages, by simply adding new template rules for further parts of the original data that there is now a need for in the dependent data model. This enables a lossless modelling of the data - not because the relational or RDF model is rich enough to express everything from the original - but because you still have the original!

Additionally, tracking the provenance of data is particularly important in some use cases, and so keeping the original information artifact is a big step towards maintaining knowledge of your data lineage.

5.4. Silo-Busting

Working with many types of data model often means working with multiple types of database. This introduces a host of problems, including loss of transactional consistency accross your data, loss of a consistent backup, and multiple

databases to host, configure and replicate. It also creates silos where some but not all data is available in different places, and so inhibits the use that the data can be put to.

A database that supports multiple data models, and that can perform transactional declarative transformations between those data models removes all of these problems.

5.5. Data in Context

Maintaining the original document context of relational or RDF data brings many benefits. Queries can be performed that pre-filter the relational or RDF view based on criteria in those documents - or queries can fetch the original documents to retrieve additional information.

For instance the provence of a given RDF statement can be seemlessly traced back to the article it came from in order to check the validity of its source, or attach a confidence to that statement.

As another example, rather than adding temporal validity information (start and end date-times) to every row that constitutes part of the information for a given entity, the temporal information can be added to the source document, and temporal queries against the source document can be used to pre-filter the relational views accessed. This gives a very simple way to provide temporal (or bitemporal) queries over a relational view with no relational schema changes or repeated date entries necessary.

Furthermore, it becomes simple to provide entity based access-control. In MarkLogic the original permissions of the document are applied to all relational or RDF views - so security is bulletproof, and ETL does not result in leaky information.

6. Conclusion

The "Template Driven Extraction" techniques described in this paper bring MarkLogic closer to an entity-relationship data modeling paradigm, while uniting some important features of relational and RDF data models and query capabilities. In addition, this technique enables further important benefits.

On the Descriptions of Data

The Usability of Notations

Steven Pemberton
CWI, Amsterdam
<steven.pemberton@cwi.nl>

Abstract

Usability describes the ease with which you can use something: how long it takes to achieve your aims, how correctly, and whether it is enjoyable in the process.

While this is normally applied to interactions with processes, such as computer programs, or machines, it is also applicable to notations: how easily can you achieve what you are trying to do, does the notation aid you in avoiding errors, and, indeed, is it enjoyable to do? However, surprisingly little attention is paid to designing notations for usability.

Invisible XML (ixml) is a technique for treating any parsable format as if it were XML, and thus allowing any parsable object to be injected into an XML pipeline. It uses a notation for describing data formats that are to be parsed.

Earlier papers on ixml discuss the design of the notation based on functional requirements of the language. This paper discusses changes to the design following experience with using it, giving examples of its use to develop data descriptions, and in passing, suggests other output formats.

Keywords: XML, parsing, notation design, data representation, usability

1. Usability

The study of usability is typically carried out in the field of human-computer interaction [10]. Although there are a small number of definitions of what usability is, all agree on three basic points [7], the degree to which you have:

- Efficiency: the time needed to achieve a purpose
- Effectiveness: being able to achieve the purpose without error
- Enjoyability: being able to enjoy achieving the putrpose.

Some definitions add *learnability* to this, and some add a related concept, *memorability*, which is the ability to return after some time and start using it with less effort than initially; however, others argue that these are orthogonal concepts,

and that something that is inherently hard to learn can still be usable despite the steep learning curve.

2. Notations

That notations can affect what is achievable with them is long known. An obvious example is the notation for numbers: the Roman notation CXXVIII is reasonably good for representing the number 128, and is just about acceptable for addition, since with a few simple rules it is possible to add two numbers represented with Roman numerals together, but it is close to impossible to do multiplication in any general way using the numerals alone.

On the other hand, the indo-arabic system that we now use is good for representing numbers, and makes it easy to both add and multiply.

3. Invisible XML

Invisible XML [4], [5], [6] is a methodology for treating heterogenous data representations so that they can be processed as XML: as long as the external data representation is parsable, it is possible to create an internal representation of the data that can further be treated as if it originated from an XML representation, or that can be serialised so that it really is XML, for input to other tools.

4. Data Descriptions

Ixml works by using context-free grammars [1] to describe the format of documents to be converted. A general parsing algorithm, such as Earley [3], is then used to create a parse-tree for the parsed document. This parse tree is then pruned, and the result can be treated as an internal representation of an XML instance, for processing or serialisation.

Using a general-purpose parsing algorithm means that users don't have to be aware of the special rules for languages that are dictated by some types of parsing algorithm, and additionally widens the class of languages that can be described.

In the initial design of ixml [4], focus was put on the functional requirements of the data description language: it was necessary to be able to specify the input format, and describe how to prune the resulting parse-tree to only deliver the useful parts for extracting the enclosed data.

There are two parts to a grammar: the *non-terminal* symbols, roughly speaking describing the semantic concepts represented in a language, and the *terminal symbols* used to express the language.

Terminal symbols can have *intrinsic* role, an *extrinsic* role, or a *structural* role. For instance, to represent "the sum of a and b", you might write `(a+b)`. The characters "a" and "b" are intrinsic, change them and you change the meaning; "+" is

extrinsic, since it only identifies that addition is involved; indeed if you change it to "×" the meaning changes, but in a different way; finally the brackets add nothing essential to the meaning, and are only used as punctuation and to disambiguate similar forms.

Nonterminals also have three roles. *Intrinsic*, where they represent a semantic concept in a language, such as "statement" in a programming language; *structural*, where they are used for disambiguation and structuring, such as "factor" in an expression, and finally *refining*, where a name is given largely for convenience to a syntactic structure than can occur in different places in a language.

In the original design, since terminal symbols were almost never intrinsic, it was decided to exclude them by default from the parse-tree, and required specific marking to include a symbol. On the other hand since nonterminals were the *sine qua non* of a parse tree, they were included by default, and required specific marking to exclude them.

Exclusions were always marked at the *use* of a symbol rather than its definition, since that meant that symbols generally were more useful in their use in the descriptions, and since inclusions were by definition only possible at the point of use (since terminal symbols are only used, and have no further definition).

However, while this produced a technically sufficient notation, it turned out in use that nonterminals need to be excluded very often, making for descriptions that were not easy to read. So in the original paper it was described how *all* symbols were excluded by default from the parse tree, and any symbol had to be explicitely marked for inclusion.

5. User Testing

Part of usability-based design is user testing. As Nielsen points out [9], user testing with only a small number of users can reveal large amounts of data about a design.

What we learned after experience of ixml in use was:

1. It is easier to design the data description by starting from the full parse tree, and incrementally pruning the parts that are not needed.
2. Very many non-terminals are refinements or extrinsic, and it is more sensible to prune these at the definition rather than the use-point.
3. Occasionally you want to prune all uses of a nonterminal but one, so it is useful to be able to mark a definition as deleted, but mark it as inserted at a use-point.
4. There are occasions where you need to say "any character *except* this small list is acceptable at this position" (this had as consequence that a notation for character sets was necessary, something that was rejected in the initial design).
5. It is useful to have an explicit notation for something that is optional.

6. It is useful to be able to use Unicode character classes.

6. An Example

To show the process of interatively building an ixml grammar, let's take an example. In order to keep the example manageable we will look at the grammar for small expressions, such as `pi×(10+b)`.

The raw grammar for expressions could look like this:

```
expr: term; sum.
sum: expr, "+", term.
term: factor; prod.
prod: term, "×", factor.
factor: id; number; "(", expr, ")".
id: letter+.
number: digit+.
letter: ["a"-"z"].
digit: ["0"-"9"].
```

The parse of the string `"pi×(10+b)"` using this grammar and then serialised as XML looks like this:

```
<expr>
   <term>
      <prod>
         <term>
            <factor>
               <id>
                  <letter>p</letter>
                  <letter>i</letter>
               </id>
            </factor>
         </term>×
         <factor>(
            <expr>
               <sum>
                  <expr>
                     <term>
                        <factor>
                           <number>
                              <digit>1</digit>
                              <digit>0</digit>
                           </number>
                        </factor>
                     </term>
                  </expr>+
                  <term>
```

```
                        <factor>
                           <id>
                              <letter>b</letter>
                           </id>
                        </factor>
                     </term>
                  </sum>
               </expr>)
            </factor>
         </prod>
      </term>
   </expr>
   <expr>
      <term>
         <prod>
            <term>
               <factor>
                  <id>
                     <letter>p</letter>
                     <letter>i</letter>
                  </id>
               </factor>
            </term>×
            <factor>(
               <expr>
                  <sum>
                     <expr>
                        <term>
                           <factor>
                              <number>
                                 <digit>1</digit>
                                 <digit>0</digit>
                              </number>
                           </factor>
                        </term>
                     </expr>+
                     <term>
                        <factor>
                           <id>
                              <letter>b</letter>
                           </id>
                        </factor>
                     </term>
                  </sum>
               </expr>)
            </factor>
```

```
      </prod>
   </term>
</expr>
```

As you can see, for such a small input string, this is a surprisingly long XML document, since it records every little part of the parse.

The first thing that is noticeable is the large numbers of `term`s and `factor`s in the tree. Since we don't need these at all in the parse tree, we mark them for exclusion at their definitions, by prepending a "-" sign:

```
-term: factor; prod.
-factor: id; number; "(", expr, ")".
```

(This only excludes the elements for `term` and `factor` from the serialisation, but not their children.)

The parse tree has already become *considerably* smaller:

```
<expr>
   <prod>
      <id>
         <letter>p</letter>
         <letter>i</letter>
      </id>×(
      <expr>
         <sum>
            <expr>
               <number>
                  <digit>1</digit>
                  <digit>0</digit>
               </number>
            </expr>+
            <id>
               <letter>b</letter>
            </id>
         </sum>
      </expr>)
   </prod>
</expr>
```

Now we exclude the letters and digits (the nonterminals, not the terminals):

```
-letter: ["a"-"z"].
-digit: ["0"-"9"].
```

which gives as parse:

```
<expr>
   <prod>
      <id>pi</id>×(
      <expr>
```

```
            <sum>
               <expr>
                  <number>10</number>
               </expr>+
               <id>b</id>
            </sum>
         </expr>)
      </prod>
   </expr>
```

and finally we get rid of the `expr`s:

```
<prod>
   <id>pi</id>×(
   <sum>
      <number>10</number>+
      <id>b</id>
   </sum>
</prod>
```

to give us a 'minimal' parse: structured, unambiguous, and still with all semantic information.

A consequence of including everything by default in the parse is the presence of 'extraneous' terminals in the tree, such as "+" and "(" above. These are harmless in themselves, and can be ignored in processing (since they are identifiable as characters not immediately surrounded by an element); they also provide an advantage that all input characters are present in the output, making the original document easier to recreate. However, if they are not wanted, they can still be explicitly removed by marking them in the same way:

```
sum: expr, -"+", term.
-factor: id; number; -"(", expr, -")".
```

giving as parse:

```
<prod>
   <id>pi</id>
   <sum>
      <number>10</number>
      <id>b</id>
   </sum>
</prod>
```

7. Attributes

Ixml grammars allow you to express that some nonterminals should be serialised as attributes in the XML, so for instance, changing the rules for `id` and `number` in the above grammar to the following:

```
id: name.
@name: letter+.
number: value.
@value: digit+.
```

gives the serialisation:

```
<prod>
   <id name='pi'/>
   <sum>
      <number value='10'/>
      <id name='b'/>
   </sum>
</prod>
```

Again, non-terminals can be marked like this at the point of definition, or the point of use.

Note that you have to be careful when defining non-terminals as attributes, since while child elements are ordered in XML, and several child elements may have the same name, this is not true of attributes.

8. Adding Nodes

If we change the input parse string from "`pi×(10+b)`" to "`pi+(10×b)`" and process it, we get:

```
<sum>
   <id name='pi'/>
   <prod>
      <number value='10'/>
      <id name='b'/>
   </prod>
</sum>
```

A possible problem here, is that this is the identical parse to what you would get for the string "`pi+10×b`": that is, the brackets in the input do not affect the parse tree. This is understandable, since the brackets add no extra information: the two strings are semantically identical. We can fix this, if required, by adding back the node for `expr` in the case that it is a bracketed expressions:

```
-factor: id; number; -"(", ^expr, -")".
```

giving for the bracketed case:

```
<sum>
   <id name='pi'/>
   <expr>
      <prod>
         <number value='10'/>
```

```
         <id name='b'/>
      </prod>
   </expr>
</sum>
```

Another solution would be to add a rule:

```
-factor: id; number; bracketed.
bracketed: -"(", expr, -")".
```

to give:

```
<sum>
   <id name='pi'/>
   <bracketed>
      <prod>
         <number value='10'/>
         <id name='b'/>
      </prod>
   </bracketed>
</sum>
```

9. Other Examples

9.1. URLs

As another small example, consider this restricted grammar for URLs:

```
url: scheme, ":", authority, path.
scheme: name.
@name: letter+.
authority: "//", host.
host: sub+".".
sub: name.
path: ("/", seg)+.
seg: sname.
@sname: fletter*.
-letter: ["a"-"z"]; ["A"-"Z"]; ["0"-"9"].
-fletter: letter; ".".
```

Here you can see the use of repetitions with separators (`sub+"."` means one or more `sub`s separated by points), as well as a grouped repetition: `("/", seg)+`

Given the input string `"http://www.w3.org/TR/1999/xhtml.html"` you get the parse:

```
<url>
   <scheme name='http'/>:
   <authority>//
      <host>
```

```
            <sub name='www'/>.
            <sub name='w3'/>.
            <sub name='org'/>
         </host>
      </authority>
      <path>/
         <seg sname='TR'/>/
         <seg sname='1999'/>/
         <seg sname='xhtml.html'/>
      </path>
   </url>
```

This illustrates a point about attributes and elements: if they have a different syntax in the input, they have to have a different name in the output. Here `sub` has an attribute called `name` and `seg` has an attribute called `sname`. The attribute `sname` cannot be called `name` because it has a different syntax to a `name` (an `sname` can contain points ".", whereas a `name` may not).

9.2. Parsing JSON

We can take the grammar for JSON [8], and convert it to ixml:

```
json: S, object.
object: "{", S, members, "}", S.
-members: pair*(",", S).
pair: @string, S, ":", S, value.
array: "[", S, value*(",", S), "]", S.
-value: string, S; number, S; object; array; "true", S; "false", ▶
S; "null", S.
string: -"""", char*, -"""".
-char: ~['"'; "\"; #0-#1F];
  '\', ('"'; "\"; "/"; "b"; "f"; "n"; "r"; "t"; "u", hexdigits).
number: "-"?, int, frac?, exp?.
-int: "0"; digit19, digit*.
-frac: ".", digit+.
-exp: ("e"; "E"), sign?, digit+.
-sign: "+"; "-".
-S: " "*.
-digit: ["0"-"9"].
-digit19: ["1"-"9"].
-hexdigits: hexdigit, hexdigit, hexdigit, hexdigit.
-hexdigit: digit; ["a"-"f"; "A"-"F"].
```

(There are some potential quibbles here: json.org says that whitespace may appear *between* any two tokens, without saying what a token is, or what whitespace is; this means that leading and trailing spaces are not allowed, which this grammar does allow).

Parsing this piece of JSON:

```
{"name": "pi", "value": 3.145926}
```

then gives us this XML:

```
<json>
   <object>{
      <pair string='name'>:
         <string>pi</string>
      </pair>,
      <pair string='value'>:
         <number>3.145926</number>
      </pair>}
   </object>
</json>
```

9.3. Parsing XML

To move to another example, let us consider a simplified grammar for XML itself:

```
xml: element.
element: -"<", name, (-" "+, attribute)*, (-">", content, -"</", ▶
close, -">"; -"/>").
@name: ["a"-"z"; "A"-"Z"]+.
@close: name.
attribute: name, -"=", value.
@value: -'"', dchar*, -'"'; -"'", schar*, -"'".
content: (cchar; element)*.
-dchar: ~['"'; "<"].
-schar: ~["'"; "<"].
-cchar: ~["<"].
```

Note the notation for character exclusions: `~['"'; "<"]` means "any character except a double quote or a less-than".

Using input:

```
<test lang="en" class="test">
  This <em>is</em> a test.
</test>
```

gives as parse:

```
<xml>
   <element name='test' close='test'>
      <attribute name='lang' value='en'/>
      <attribute name='class' value='test'/>
      <content>  This
         <element name='em' close='em'>
            <content>is</content>
```

```
          </element> a test.</content>
    </element>
</xml>
```

Note XML is not context-free, since you can't check on parsing that open and closing tags match. It is for this reason that the serialisation contains both the opening and the closing tag names, so that they can be checked on later processing.

10. Parsing ixml

Of course, an ixml grammar is itself expressible in ixml. That is to say, we can write a grammar that expresses ixml, and then process it with ixml to give an XML serialisation of the grammar.

So here is ixml expressed in itself. "S" represents a string of spaces or comments. Comments are enclosed in curly braces { and }. The notation `mark?` means that `mark` is optional at that point.

```
ixml: S, rule+.
rule: mark?, name, S, ":", S, def, ".", S.
def: alt+(";", S).
alt: term*(",", S).
-term: factor; repeat0; repeat1; option.
repeat0: factor, "*", S, sep?.
sep: factor.
repeat1: factor, "+", S, sep?.
option: factor, "?", S.
-factor: nonterminal; terminal; "(", S, def, ")", S.
nonterminal: mark?, name, S.
terminal: mark?, (quoted; hex; charset; exclude).

charset: "[", S,  element+(";", S), "]", S.
exclude: "~", S, -charset.

-element: range; character, S; class, S.
range: from, S, "-", S, to, S.
@from: character.
@to: character.
class: letter, letter. {One of the Unicode character classes}

@name: letgit, xletter*.
-letgit: letter; digit.
-letter: ["a"-"z"; "A"-"Z"].
-digit: ["0"-"9"].
-xletter: letgit; "-".

@mark: "@", S; "^", S; "-", S.
```

```
quoted: -'"', dstring, -'"', S; -"'", sstring, -"'", S.
@dstring: dchar+.
@sstring: schar+.
dchar: ~['"']; '""'. {all characters, dquotes must be doubled}
schar: ~["'"]; "''". {all characters, squotes must be doubled}
-character: '"', dchar, '"'; "'", schar, "'"; hex.

hex: "#", number.
number: hexit+.
-hexit: digit; ["a"-"f"; "A"-"F"].

-S: (" "; comment)*.
comment: "{", cchar*, "}".
-cchar: ~["}"].
{the end}
```

Parsing this *with itself* gives 479 lines of output, so only the first couple of rules will be shown:

```
<ixml>
   <rule name='ixml'>:
      <def>
         <alt>
            <nonterminal name='S'/>,
            <repeat1>
               <nonterminal name='rule'/>+
            </repeat1>
         </alt>
      </def>.</rule>
   <rule name='rule'>:
      <def>
         <alt>
            <option>
               <nonterminal name='mark'/>?
            </option>,
            <nonterminal name='name'/>,
            <nonterminal name='S'/>,
            <terminal>
               <quoted dstring=':'/>
            </terminal>,
            <nonterminal name='S'/>,
            <nonterminal name='def'/>,
            <terminal>
               <quoted dstring='.'/>
            </terminal>,
            <nonterminal name='S'/>
```

```
         </alt>
      </def>.</rule>
```

The upshot of this is that *any* grammar that produces a similar output (ignoring extraneous terminals) can be used to parse a document. That means that the grammar format for ixml is not set in stone, but alternatives can be offered. For instance, here is ixml expressed in a different grammar format, BNF [2], representing itself:

```
<ixml>::= <S> <rules>
-<rules>::= <rule> | <rule> <rules>
<rule>::= <mark> <name> "::=" <S> <def> |
      <name> "::=" <S> <def>
<def>::= <alts>
-<alts>::= <alt> | <alt> "|" <S> <alts>
<alt>::= <terms> | <empty>
-<terms>::= <term> | <term> <S> <terms>
<empty>::=
<term>::= <mark> <name> | <name> | <string> | <range>
@<name>::= "<" <letters> ">" <S>
@<mark>::= "@" <S> | "^" <S> | "-" <S>
<letters>::= <letter> <more-letters>
<letter>::= ["a"-"z"] | ["A"-"Z"] | ["0"-"9"]
<more-letters>::= <letter> <more-letters> | "-" <more-letters> | <empty>
@<string>::= """" <chars> """" <S>
<chars>::= <char> <chars> | <char>
<char>::= [" "-"!"] | ["#"-"~"] | """""" {all characters, quotes must be ▶
doubled}
<range>::= "[" <S>  <character> <S> "-" <S> <character> <S> "]" <S>
-<character>::= """" <char> """" | """" """" """" """"
-<S>::= " " <S> | <comment> <S> |
<comment>::= "{" <schars> "}"
-<schars>::= <schar> <schars> |
-<schar>::= [" "-"|"] | "~" {Everything except: }
```

which parsed with itself gives (again, only showing the first few lines):

```
<ixml>
   <rule name='ixml'>::=
      <def>
         <alt>
            <term name='S'/>
            <term name='rules'/>
         </alt>
      </def>
   </rule>
   <rule mark='-' name='rules'>::=
```

```
      <def>
         <alt>
            <term name='rule'/>
         </alt>|
         <alt>
            <term name='rule'/>
            <term name='rules'/>
         </alt>
      </def>
   </rule>
```

11. Other Output Formats

In the true spirit of "data wants to be format neutral", there is strictly speaking no reason why the parse tree need be in XML, but could be equally well serialised in some other form, such as JSON. Taking an example like this:

```
<expr>
    <prod>
        <letter>a</letter>
        <sum>
            <digit>3</digit>
            <letter>b</letter>
        </sum>
    </prod>
</expr>
```

you might be tempted to try to express this as:

```
{"expr":
    {"prod":
        {"letter": "a";
         "sum": {"digit":"3"; "letter":"b"}
        }
    }
}
```

However, JSON object members are more like XML attributes than child elements, because they are not ordered, and (probably) member names may not be duplicated.

Therefore you have to use arrays, which *are* ordered, where each array element is a single-member group:

```
{"expr":
    [{"prod":
        [{"letter": "a"}],
        [{"sum":
            [{"digit":"3"}],
```

```
            [{"letter":"b"}]
        }]
    }]
}
```

There still remains the question of what to do with extraneous terminals, such as here:

```
<expr>
    <prod>
        <letter>a</letter>×(
        <sum>
            <digit>3</digit>+
            <letter>b</letter>
        </sum>)
    </prod>
</expr>
```

The best approach is to treat them as members without a name, like this:

```
{"expr":
    [{"prod":
        [{"letter": "a"}],
        [{"": "×("}],
        [{"sum":
            [{"digit":"3"}],
            [{"":"+"}]
            [{"letter":"b"}]
        }],
        [{"":")"}]
    }]
}
```

12. Conclusion

Notations can have a profound effect on what we are able to express. In the end notations are a human artifact, for use by people, and we should not lose sight of the fact that what is suitable for an automaton is not necessarily ideal for a person. We can learn from the techniques of usability, such as user testing and iterative design, to make notations that are more suitable for human use.

Bibliography

[1] AV Aho, and JD Ullman, The Theory of Parsing, Translation, and Compiling, Prentice-Hall, 1972, ISBN 0139145567

[2] Backus-Naur Form, http://en.wikipedia.org/wiki/Backus-Naur_Form

[3] Earley Parser, https://en.wikipedia.org/wiki/Earley_parser

[4] Steven Pemberton. Invisible XML, Presented at Balisage: The Markup Conference 2013, Montréal, Canada, August 6 - 9, 2013. In Proceedings of Balisage: The Markup Conference 2013. Balisage Series on Markup Technologies, vol. 10 (2013). doi:10.4242/BalisageVol10.Pemberton01. http://www.cwi.nl/~steven/Talks/2013/08-07-invisible-xml/invisible-xml-3.html

[5] Steven Pemberton. Data Just Wants to Be Format-Neutral, Proc. XML Prague, 2016, Prague, Czech Republic, pp109-120. http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf

[6] Steven Pemberton. Parse Earley, Parse Often: How to Parse Anything to XML, Proc. XML London 2016, London, UK, pp120-126. http://xmllondon.com/2016/xmllondon-2016-proceedings.pdf#page=120

[7] International Standards Organisation, ISO 9241 - Ergonomic requirements for office work, 1997.

[8] Introducing JSON, http://json.org/

[9] Jakob Nielsen, Why You Only Need to Test with 5 Users, 2000. https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/

[10] Ben Shneiderman *et al*., Designing the User Interface, Pearson, 2013, ISBN 1292023902.

FOXpath navigation of physical, virtual and literal file systems

Hans-Jürgen Rennau
Traveltainment GmbH
<hrennau@yahoo.de>

Abstract

The FOXpath language extends the XPath language by adding support for file system navigation. This paper explores possibilities how to extend file system navigation beyond physical file systems and include logical file systems like jar files, SVN repositories or github projects. The extension is based on a set of simple concepts related to URIs and their processing, and it is implemented as a FOXpath processor which supports the navigation of physical and various types of logical file systems.

Keywords: FOXpath, XPath, XQuery, file systems, file system navigation, BaseX

1. Introduction

Tree-structured information is ubiquitous. A simple shopping list, for instance, which groups items by department or shop, is a tree. Other examples include file systems, jar files, SVN repositories, NOSQL databases, github projects and Java packages. Major advantages of a tree structure are obvious. It favours a clear understanding where to find what and enables an intuitive addressing of single items. It invites to dwell on a chosen level of detail or abstraction. It domesticates complexity and appears natural to human thought. One particular advantage, however, is still well-hidden from broad attention: this is an amazing **conductivity** which information attains when arranged as a tree. The term borrowed from physics should capture the ease of traversing a bulk of information, in order to arrive at points of interest. This quality is revealed by XPath (6), an expression language for navigating the contents of XML node trees. Consider this example:

```
//route/arrival[airport = 'JFK']/../departure[@t > '18:00']/airport
```

The expression returns the departure airports of evening flights to the airport JFK. In order to arrive at the `airport` items of interest, complex information is traversed in a single fell sweep.

The elegance of such expressions is not based on specific qualities of the XML data format – it reflects inherent qualities of tree-structured information. Consider the key concepts of XPath: the combination of *navigation axis*, *node test* and

predicates into a step; the chaining of steps into a *path*. The XPath model of navigation is not tied to XML and would perhaps make sense in many tree-structured collections of information. How about file systems? Like XML documents, they are trees, and finding particular folders and files is as important (at least) as finding particular XML elements and attributes. Can we have something like XPath for file systems?

The FOXpath language (2) turns this idea into reality. It is a superset of XPath 3.0, adding file system navigation – multi-step movement across the hierarchical structure of the file system. Consider this FOXpath expression:

```
\projects\gateway\\routes-*.xml[not(ancestor~::blacklisted)]
```

Its value is the set of documents about flight routes found in the gateway project, but not at any depth under a `blacklisted` folder. The *integration* of XPath-like file system navigation into XPath has two interesting benefits. First, file system navigation can be seamlessly combined with file content navigation – even within a single expression. Example:

```
\projects\gateway\\routes-*.xml[not(ancestor~::blacklisted)]
//route/arrival[airport = 'JFK']/../departure[@t > '18:00']/airport
```

Here the navigation to file resources (\ seperated steps) is continued by navigation *within* them (/ seperated steps). The next example,

```
\projects\gateway\\routes-*.xml[not(ancestor~::test)]
//route/arrival[airport = 'JFK']/../departure[@t > '18:00']/airport
=> distinct-values() => sort()
```

demonstrates the advantages of navigation embedded into a full-blown expression language: navigation can be used as operand of other operations which have nothing to do with navigation.

FOXpath thus creates an interesting experience of information: a tree (file system) whose leaf nodes (files) are themselves trees (XML node trees), merged into a continuous space of information by a single navigation model, which is embedded into a fully composable expression language.

This paper reports my efforts to push things one step further: to extend file system navigation beyond physical file systems, into the realm of logical file systems like jar files or github projects (3). As a preparatory step I provide some information about the FOXpath language.

2. The FOXpath language

FOXpath (2) is a superset of the XPath 3.0 language (6). The purpose of extension is adding support for file system navigation which has the same look and feel as node tree navigation. The extension of XPath can be summarized as follows:

- Addition of a new expression kind, the *URI axis step* (e.g. `descendant~::foo`)

- Addition of a new operand, the *URI path operator* (e.g. `foo\bar`)
- Semantic extensions (backward-compatible)

A *URI axis step* expression mirrors the axis step expression of XPath, being a combination of navigation axis, name test and optional predicates. A URI axis step maps input URIs to output URIs related to the input URIs by a navigation axis (child, descendent, parent, ancestor, …). File system navigation axes are defined in analogy to the navigation axes defined by the XPath language.

The *URI path operator* chains two consecutive steps of URI navigation into a complex operation, echoing the way how the path operator of XPath chains two consecutive steps of node navigation.

Semantic extensions modify the semantics of a few XPath expressions in such a way that

- Any expression which yields a value (rather than raises an error) when evaluated as an XPath expression yields the same value when evaluated as a FOXpath expression
- Some expressions which raise an error when evaluated as an XPath expression yield a value when evaluated as a FOXpath expression

The semantic extensions enable a seamless integration of file system and node navigation. Consider the expression:

```
\projects\\foo.xml//bar
```

which navigates to `foo.xml` files and then to the `bar` elements which they contain: without semantic extension, the node navigation would raise an error as the context item is a resource URI, not a node. However, the extended semantics of node axis steps prescribe automatic conversion of an atomic context item into a node by parsing the document found at the document URI given by the context item. Further semantic extensions concern the definition of the effective boolean value, which in certain cases yields a value in FOXpath, whereas an error is raised in XPath.

Besides file system navigation, the FOXpath language adds to XPath 3.0 a few more extensions:

- Support for external variables *(as defined by XQuery)*
- Support for namespace declarations *(as defined by XQuery)*
- Support for simple FLWOR expressions, containing one or more `let` clauses and/or one or more `for` clauses *(simplified version of XQuery's FLWOR expression)*
- Support for the `arrow` operator *(as defined by XPath 3.1)*
- About 60 extension functions

It is important to note, however, that FOXpath does not extend or modify the data model of XPath (XDM, (8). File system navigation is viewed as operations applied to strings interpreted as URIs.

To summarize, the FOXpath language complements node tree navigation with file system navigation, moving around in a tree of folders and files. The FOXpath model of file system navigation is a faithful copy of the XPath model of node navigation. In particular, navigation is viewed as a sequence of steps which typically combine a navigation axis (child, descendant, parent, ancestor, …) with an item test (name test) and optional predicates.

3. The challenge of generalization

Many systems of resources appear to be organized like file systems, although they are not implemented by a physical file system. Examples include the contents of archive files, the repositories of version control systems, some NOSQL database systems, project repositories like github and the resources exposed by many RESTful web services. If the scope of FOXpath navigation could be extended to include such logical file systems, the usefulness of the FOXpath language might increase considerably.

The model of FOXpath navigation is based on navigation axes. These presuppose tree-structured collections of URIs, of which physical file systems are just a special case. The FOXpath language is therefore in no way tied to physical file systems, and navigation of non-physical and physical file systems should be equally possible. This situation suggests a dual challenge:

1. To provide a *conceptual framework* relating the abstract navigational functionality of the FOXpath language to an implementation-defined set of logical file systems
2. To provide a proof of concept *implementation* of the FOXpath language which supports the navigation of physical as well as various kinds of non-physical file systems.

4. Concepts

This section sketches a conceptual framework relating the navigation model of the FOXpath language – based on navigation axes applicable to resource URIs – and actual systems of resources to which such navigation may be applied. Formal rigour and precision are sacrificed to readability. As an example of this imprecision, the contained-by relationship between a given URI and a file system is not discussed, although it is important for the exact meaning of an operation like "mapping a URI to its child URIs", as such a mapping can only be unambiguously defined in the context of an actual file system.

4.1. File system types

Desiring a generalization of FOXpath navigation, a good starting point is a generalized concept of file systems. Loosely speaking, a *logical file system* is a tree of

URIs which has a single root, whose leaf URIs can be resolved to a chunk of content (sequence of characters or bytes) and whose inner nodes *cannot* be resolved to content, but are perceived as containers of other URIs. The generalized notions of files and folders are thus the outer and inner nodes of such a tree of URIs.

As the notion of a "tree of URIs" is vague, we introduce the concept of **child URI**. A URI U2 is said to be a child URI of another URI U1 if U1 is a prefix of U2, followed by a slash and a non-empty sequence of characters not containing a slash. So, for example, the URI

```
https://github.com/marklogic/semantic
```

is a child URI of

```
https://github.com/marklogic
```

A logical file system can thus be characterized as a tree of abstract nodes representing URIs, where child nodes are child URIs and only leaf nodes are URIs pointing to content. Some examples:

- The contents of a jar file (or some other archive file)
- The contents of a BaseX database (or some other NOSQL database)
- The contents of an SVN repository (or some other version control system)
- The contents of a github project
- The URIs exposed by a typical RESTful web service

A logical file system is usually backed by APIs enabling some form of navigation and content retrieval. Applications based on such APIs may create the semblance of a file system that can be browsed in the usual way. As an example, consider the github web service (4) and the github web site (3). When APIs and applications create the semblance of a physical file system, we speak of a **virtual file system**.

While virtual file systems *behave* like a tree of URIs, they usually do not expose any literal representation of the tree and its nodes, a representation composed of distinct units of information describing. individual URIs. Such a literal representation might for example be an XML document or an RDF graph, representing URIs by XML elements or RDF nodes, respectively. Note however that for any logical file system such a representation can be *constructed* by tools using the available APIs. We propose the notion of a **literal file system** which is a tree of resource URIs described by an artifact composed of distinct units representing single URIs. A literal file system may, but need not be a one-to-one representation of a physical or virtual file system. It may contain selected parts of a single or multiple logical file systems, and also URIs not associated with any file system at all. Such unconstrained compositions can be merged into a logical file system (a tree of URIs) because the URIs actually *exposed* need not be the original URIs. The URIs integrated into a literal file system are *navigation URIs*, which are associated with *resource access URIs* – the original URIs which may be equal to or different from the navigation URI. A literal file system can be thought of as a special kind

of catalog, mapping a set of URIs (resource access URIs) to a tree-structured set of target URIs (navigation URIs) and providing additional information about the resources, like file size or date of last modification).

In summary, we introduce the notion of a **logical file system** which may be a **physical file system**, a **virtual file system** or a **literal file system**. The FOXpath language is tied to the concept of a logical file system, rather than a physical file system. In principle, it should be possible to support navigation of all kinds of logical file system. To realize this potential, it is important to identify key operations which (a) serve as an abstraction hiding the actual type of file system at hand, (b) serve as building blocks from which the complete functionality of FOXpath navigation can be constructed.

4.2. URI operations

URI operations are operations applied to URIs and yielding a result which is valuable in the context of file system navigation and the retrieval of resource contents and resource properties. We reserve the term for basic operations, usable as building blocks from which to construct more complex operations. A crucial aspect is to define these operations in a generic way which is independent of the type of logical file system at hand (physical file system, archive file, …). For instance, an SVN command defined to map the URI of an SVN folder to the URIs of contained SVN folders and files is not a generic URI operation; an operation defined to map a URI to its child URIs is a generic URI operation. Think of URI operations as a basic interface exposed by a logical file system of any type.

4.2.1. Navigation primitives

File system navigation as defined by the FOXpath language is based on navigation axes. These are mappings of an input URI to related URIs, e.g. descendant or ancestor URIs. Upward navigation is a purely syntactical operation (the removal of trailing URI steps) and downward navigation can be achieved by single or recursive mapping of a URI to its child URIs. Therefore the complete implementation of all FOXpath-defined axes requires only a single navigation primitive, which is a mapping of an input URI to its child URIs.

Practical considerations, however, led me to define two navigation primitives designed with the goal of balancing simplicity and the efficiency of operations based on these primitives:

```
uri-to-child-uris      ($uri, $nameFilter?, $kindFilter?) as xs:anyURI*
uri-to-descendant-uris ($uri, $nameFilter?, $kindFilter?) as xs:anyURI*
```

where `$uri` is the input URI, `$nameFilter` is an optional name pattern to be matched by the trailing step of the returned URIs, and `$kindFilter` is an optional filter retaining only files or only folders.

4.2.2. Content retrieval

Files are URIs pointing to content, which is a sequence of characters or bytes. We define the following URI operations for content retrieval:

```
file-text       ($uri, $encoding) as xs:string
file-bytes      ($uri) as xs:base64Binary
file-doc        ($uri) as document-node()
file-json-doc   ($uri) as document-node()
```

4.2.3. Resource property retrieval

A file system resource has a few basic properties:

- resource kind, which is `file` or `folder`
- timestamp of last modification
- size in bytes (only if the resource is a file, not a folder)

The following URI operations can be used to retrieve the property values:

```
resource-kind($uri)    as xs:string        // value is one of 'file' ▶
or 'folder'
resource-date($uri)    as xs:dateTime?
resource-size($uri)    as xs:nonNegativeInteger?
```

4.3. URI processor

A **URI processor** is a program or program module which implements the URI operations for URIs stemming from a particular type of logical file system. URI processors are therefore classified by the type of logical file system they can handle. A few examples:

- file system URI processor
- archive file URI processor
- BaseX database URI processor
- SVN URI processor
- UTREE processor
- UGRAPH processor

UTREE and UGRAPH processors deal with literal file systems defined by XML documents and RDF graphs, respectively. See [Section 5] for details.

4.4. URI dispatchal rules

A FOXpath implementation including a particular type of URI processor may support FOXpath navigation of the corresponding type of file systems. An SVN URI processor, for example, may enable FOXpath navigation of SVN repositories.

Navigation support for *multiple* types of file system requires an identification of the type of file system to which a given URI belongs. This operation I call **URI**

dispatchal, as it amounts to dispatching a URI to the appropriate URI processor. The dispatchal rules of a FOXpath implementation are implementation defined. Obvious possibilities are based on URI schemes and URI prefixes. Rules may reference configuration data, in which case they must define the format and semantics of such data.

Note that dispatchal rules do not identify an *instance* of a file system – e.g. a particular SVN repository. They only identify the *type* of file system.

5. Literal file systems

A literal file system is an association between a collection of resources and a tree of resource URIs. The tree facilitates discovery of the resources via navigation, and it enables retrieval of resource contents and basic resource properties. The tree should be viewed as a tree of logical components, to be distinguished from data encoding the tree. I have defined two data formats, where one is an XML document and another is an RDF graph.

5.1. Logical model

A literal file system is an instance of an information model designed to model a system of folders and files. It is a tree of abstract nodes defined and constrained as follows:

- Every leaf node represents either a file or an empty folder
- Every inner node represents a non-empty folder
- A file node has the following properties:
  - A navigation URI
  - A resource access URI
  - File size (optional)
  - Timestamp of last modification (optional)
- A folder node has the following properties:
  - A navigation URI
  - Timestamp of last modification (optional)
- For any two nodes N1 and N2 where N2 is a child node of N1, the navigation URI of N2 is a child URI of the navigation URI of N1

Navigation and resource access URIs are distinct properties of a file node – they may, but need not be equal. A navigation URI is a URI associated with a resource in order to make it conveniently discoverable. From the file system user's point of view, the navigation URI is also the URI which is resolved to the resource contents. From the file system implementation's point of view, however, resource access is not provided by the navigation URI, but by the associated resource access URI. The tree structure emerging from the collected navigation URIs does in no way constrain the locations of the resources represented by that tree, nor the

protocols used in order to retrieve those resources. The literal file system *may* be a one-to-one representation of some physical or virtual file system. It may also be a projection of such a system, representing some and not representing other resources contained by the original system. A literal file system may also represent a collection of resources belonging to *several* logical file systems, or even not belonging to any known file system at all.

5.2. Data formats - UTREE and UGRAPH

Two data formats representing a literal file system have been defined: an XML based representation (UTREE) and an RDF based representation (UGRAPH).

5.2.1. The XML format (UTREE)

UTREE is an XML format for representing the contents of one or several literal file systems. Element summary:

- Document root element: `<trees>`
- File system root element: `<tree>`
- Inner nodes: `<dir>` elements
- Leaf nodes: `<file>` and/or `<dir>` elements

Attribute summary:

- **On <trees>**
  - @uriPrefix – optional; if used, a URI processor must treat any URI starting with that prefix as either described by a <tree>, <file> or <dir> element in that document or not corresponding to any existent resource
- **On <tree>**
  - @baseURI – the navigation URI of the literal file system root, followed by a slash, unless the navigation URI ends with a slash
- **On <file>**
  - @name – the file name
  - @path – the trailing part of the navigation URI; the URI is obtained by appending the attribute value to the @baseURI value of the containing <tree> element
  - @accessURI – the URI to be used for resource access
  - @lastModified – timestamp of last modification (optional, may be unknown)
  - @size – size of the file, in bytes (optional, may be unknown)
- **On <dir>**
  - @name – the folder name

- @path – the trailing part of the navigation URI; the navigation URI is obtained by appending the attribute value to the @baseURI value of the containing <tree> element
- @lastModified – timestamp of last modification (optional, may be unknown)

Constraints

- A child node of a given node N has a navigation URI which is a child URI of the navigation URI of N

5.2.2. The RDF format (UGRAPH)

UGRAPH is an RDF format for representing the contents of a literal file system. In the following description, the prefix `fs:` represents the URI

```
http://www.foxpath.org/ns/rdf/filesystem/
```

In an instance of UGRAPH, two types (`rdf:type`s) of resources are distinguished:

- `fs:dir` – resource represents a folder
- `fs:file` – resource represents a file

Properties of `fs:dir` resources:

- `fs:navURI` – the navigation URI of a folder
- `fs:parentDir` – resource URI of the parent folder resource (optional, as the root folder has no parent folder)
- `fs:name` – the folder name (optional, as the root folder may have no name)
- `fs:lastModified` – timestamp of last modification (optional, may be unknown)

Properties of `fs:file` resources:

- `fs:navURI` – the navigation URI of a file
- `fs:accessURI` – the resource access URI of a file
- `fs:parentDir` – resource URI of the parent folder resource
- `fs:name` – the file name
- `fs:lastModified` – timestamp of last modification (optional, may be unknown)
- `fs:fileSize` - size of the file, in bytes (optional, may be unknown)

Constraints:

- An UGRAPH has exactly one `fs:dir` resource without `fs:parentDir` property; this resource represents the root folder of the system
- An `fs:dir` resource without `fs:parentDir` property must not be a descendant URI of a resource contained by the graph – any non-root resource is connected to the root folder by a chain of `fs:parentDir` properties
- If a resource R has an `fs:parentDir` D, D is also contained by the UGRAPH

- If a resource R has an `fs:parentDir` D, the `f:navURI` of R is a child URI of the `f:navURI` of D
- A non-root folder MUST have a name

6. Implementation

This section describes the implementation of a FOXpath language processor which supports the navigation of physical, virtual and literal file systems. The language implementation is accompanied by a command-line tool for creating literal file systems represented by UTREE documents or UGRAPH triple sets.

6.1. Overview

A FOXpath language processor can be downloaded from here:

```
https://github.com/hrennau/foxpath
```

The scope of file system navigation includes:

- physical file systems
- archive files (jar, zip, …)
- BaseX databases
- SVN repositories
- github.com repositories (represented by UTREE or UGRAPH)
- UTREE literal file systems
- UGRAPH literal file systems

The processor ships with a command-line tool for creating literal file systems (UTREE or UGRAPH style) ...

- from SVN repositories
- from github.com repositories

The FOXpath processor is implemented as a set of XQuery modules, XQuery version 3.1. As several extension functions of the XQuery processor BaseX (1) are used, FOXpath can currently only be executed using the XQuery processor BaseX. The implementation of the FOXpath language is a set of ordinary XQuery modules – it does not involve any changes of the XQuery processor.

The *XQuery interface* of the processor is a function resolving FOXpath expressions to their value.

A *command-line interface* is provided by shell scripts (`fox.bat`, `fox.sh`). They resolve FOXpath expressions provided either as a command-line parameter or as file contents.

6.2. Interfaces

The *XQuery interface* of the implementation is an XQuery function resolving FOXpath expressions to their value:

```
declare function f:resolveFoxpath(
  $foxpath as xs:string,     (: the expression text :)
  $ebvMode as xs:boolean?,   (: true => return effective boolean value :)
  $context as xs:string?,    (: folder URI providing file system ▶
context :)
  $options as map(*)?,       (: UTREE locations, UGRAPH endpoints, ... :)
  $externalVariableBindings as map(xs:QName, item()*)?)
      as item()*             (: the expression value :)
```

The *command-line interface* is a shell script (`fox.bat`, `fox.sh`) resolving FOXpath expressions to their value:

```
fox [-t utree-dir] [-g ugraph-endpoint] [-v name=value]* expression-text ▶

fox [-t utree-dir] [-g ugraph-endpoint] [-v name=value]* -f expression-
file
```

6.3. Implementation details

The XQuery-based implementation of the FOXpath language relies on several *XQuery extension functions* supported by the XQuery processor BaseX (1). The following tree-structured representation summarizes the dependencies of functional areas. Underlying view:

- FOXpath = XPath + file system navigation
- file system = physical file system | archive file | BaseX database | SVN repo | UTREE | UGRAPH

Leaf nodes of the tree are XQuery extension functions, inner nodes are areas of functionality, the root node is the FOXpath language, as implemented. The notations [expath] and [basex] mark functions as defined either by EXPath (2) or BaseX (1).

```
FOXpath
.  XPath
.  .  [basex] xquery:eval                 // for partial function ▶
applications
.  logical_file_system_navigation
.  .  physical_file_system
.  .  .  [expath] file:list               // downward navigation
.  .  .  [expath] file:is-dir             // retrieval of file properties
.  .  .  [expath] file:last-modified      // retrieval of file properties
.  .  .  [expath] file:size               // retrieval of file properties
.  .  .  [expath] file:read-binary        // resource retrieval
```

```
.  .  archive_file
.  .  .  [expath] file:read-binary        // downward navigation
.  .  .  [basex]  archive:entries         // downward navigation
.  .  .  [basex]  archive:extract-text    // resource retrieval
.  .  .  [basex]  archive:extract-binary  // resource retrieval
.  .  basex_database
.  .  .  [basex]  db:list                 // downward navigation
.  .  svn
.  .  .  [basex] proc:system              // access to SVN CL interface
.  .  utree
.  .  utree_for_github
.  .  .  [expath] http:send-request       // access to github REST API
.  .  .  [basex]  convert:binary-to-string// github API delivers binary
.  .  ugraph
.  .  .  [expath] http:send-request       // access to SPARQL endpoints
.  .  ugraph_for_github
.  .  .  [expath] http:send-request       // access to github REST API
.  .  .  [basex] convert:binary-to-string // github API delivers binary
```

6.4. URI dispatchal

URI dispatchal maps a given URI to at most one of the supported file system types. URI dispatchal rules are *implementation defined*, and they are crucial for supporting the navigation of multiple file system types.

6.4.1. URI dispatchal configuration

Dispatchal is controlled by static rules and configuration data passed to the expression resolver at runtime. The configuration data consist of

- UTREE folders – folders containing one or more UTREE documents
- UGRAPH endpoints - SPARQL endpoints exposing one or more UGRAPH triple sets

UTREE folders and UGRAPH endpoints are passed as parameters to the FOXpath processor. On the command line, the UTREE folders and UGRAPH endpoints are specified using option –t and –g, respectively:

```
fox -t "utree-dirs" ...
fox -g "ugraph-endpoints" ...
```

On the XQuery interface of FOXpath, UTREE folders and UGRAPH endpoints can be specified as entries in a map of options (keys UTREE_DIRS and UGRAPH_ENDPOINTS).

6.4.2. URI dispatchal rules

The following URI dispatchal rules are evaluated in order, and the first rule inferring a file system type provides the final result.

6.4.2.1. Rule #1 - archive rule

A URI containing one or more steps with the text `#archive#` points to contents of an archive file. The *archive file URI* is given by the path prefix preceding the last occurrence of an `#archive#` step. The *within-archive path* is given by the path suffix following the last occurrence of an `#archive#` step.

For example, the URI

```
/apache-jena-3.1.0//jena-tdb-3.1.0.jar/#archive#/org//*.properties
```

references all resources found in the archive file

```
/apache-jena-3.1.0//jena-tdb-3.1.0.jar
```

at locations matching

```
/org//*.properties
```

URI dispatchal rules are applied recursively in order to access the archive file. The archive file may therefore be contained by any of the file system types supported by the FOXpath processor. It may, for example, itself be located in an archive file.

6.4.2.2. Rule #2 - literal file system rule

If the URI does not refer to archive contents, run time inspection of the provided UTREE documents and UGRAPH triples enables the FOXpath processor to infer a mapping of URI prefixes to UTREE documents and UGRAPH endpoints:

- A URI with one of the UTREE-associated prefixes is inferred to belong to a UTREE-based literal file system
- A URI with one of the UGRAPH-associated prefixes is inferred to belong to a UGRAPH-based literal file system

6.4.2.3. Rule #3 - BaseX rule

If the URI starts with `basex:/`, the URI refers to a resource contained by a BaseX database.

6.4.2.4. Rule #4 - SVN rule

If the URI scheme starts with `svn-` (e.g. the URI starts with `svn-file:/`, `svn-http:/` or `svn-https:/`), the URI refers to a resource contained by an SVN repository.

6.4.2.5. Rule #5 - github rule

If the URI starts with `https://api.github.com/repos/`, the URI refers to a resource contained by a github project.

6.4.2.6. Rule #6 - physical file system rule

- If URI starts with `file:/`, the URI refers to a physical file system resource
- If the URI has no URI scheme, the URI refers to a physical file system resource

6.4.2.7. Rule #7 - match failure rule

A URI for which no file system can be determined, does not belong to a (recognized) file system. This implies:

- Attempts at downward navigation (e.g. navigation to child URIs) yield the empty sequence
- Attempts to retrieve file properties yield the empty sequence
- The semantics of content retrieval (functions `fn:doc`, `fn:unparsed-text`, ...) are not affected - content retrieval is controlled by the URI scheme

6.4.3. Proprietary URI schemes

Note the use of **proprietary URI schemes**

- `svn-…/` (… = file | http | https)
- `basex:/`

A *proprietary URI* is derived from a *standard URI* by inserting a prefix (`svn-`, `basex:`) or an intermediate step (`#archive#`), which indicates the type of containing file system. The FOXpath implementation relies on the use of proprietary URIs in order to recognize certain types of file system, like SVN, BaseX or archive, which cannot be inferred from the URI scheme. Proprietary URIs are used by the FOXpath processor, but never passed to the underlying APIs for navigation and retrieval. Note however that the *choice* of underlying API is crucial, as navigation, for instance, must be handled differently according to the file system type. The switching between proprietary and standard URIs is transparent to the FOXpath user, though, who supplies and receives only the proprietary URIs. Consider the following examples:

```
svn-https://svn.apache.org/repos/asf/tomcat/jk/trunk/*
svn-https://svn.apache.org/repos/asf/tomcat/jk/trunk/KEYS/grep('apache')
https://svn.apache.org/repos/asf/tomcat/jk/trunk/*
```

The first expression yields a list of child URIs, which of course have the same URI scheme as the parent URI. These URIs cannot be used outside of FOXpath expressions. Within FOXpath expressions, however, they are used as if they were stand-

ard URIs, as the second expression demonstrates, which retrieves resource contents. The third expression will probably return the empty sequence, as the URI is not recognized as referring to SVN resources.

6.4.4. Tool support for constructing literal file systems

The creation of literal file systems is supported by a command-line tool called `lifis` (literal file system). The tool enables the convenient construction of literal file systems whose structure and contents capture the structure and contents of a physical or virtual file system – e.g. a github or SVN repository. The literal file system can be restricted to a fragment of the original system (rooted in a non-root folder) and to the result of filtering the original system contents (ignoring selected folders or files). The tool user identifies one or several source file systems and the literal file system format (UTREE or UGRAPH). The tool creates a UTREE document or a UGRAPH graph. Additional options can be specified which load the UTREE document into a BaseX database or load the UGRAPH triples into a TDB triple store (5).

7. Examples

Several examples will demonstrate FOXpath navigation of various types of logical file systems. The examples are provided as calls of the `fox` script, a command-line tool for resolving FOXpath expressions. The examples use a syntax variant of the FOXpath language called **friendly syntax**, in which the roles of slash and backslash are swapped. The *axis step operator* inherited from XPath is represented by a *backslash*, not by a slash as in XPath. The *slash* represents the URI axis step operator, which separates file system navigation steps.

7.1. Navigating the physical file system

We explore an installation of the application server Wildfly (version 10.1.0). To get started, we list the top-level folders, together with the number of contained XML files:

```
fox "/wildfly101/*[is-dir()]
     /concat(rpad(file-name(), 25, '.'), '   ', count(.//*.xml))"

=>
.installation ...........   0
appclient ...............   1
bin .....................   1
docs ....................   15
domain ..................   4
modules .................   362
```

```
standalone ..............   4
welcome-content .........   0
```

Next, we list the paths of all XML files not contained in the `modules` or the `docs` folder:

```
fox "/wildfly101//*.xml[not(ancestor~::modules)][not(ancestor~::docs)]"

=>
/wildfly101/appclient/configuration/appclient.xml
/wildfly101/bin/jboss-cli.xml
/wildfly101/domain/configuration/domain.xml
…
```

Finally, we list all XML files which are not described by any of the (278) XSDs contained in the installation:

```
fox "let $xsdnames :=
          /wildfly101//*.xsd
          \xs:schema\xs:element\@name\QName(..\..\@targetNamespace, .)
     return /wildfly101//*.xml[not(node-name(\*) = $xsdnames)]"

=>
/wildfly101/docs/licenses/licenses.xml
/wildfly101/modules/system/layers/base/org/jboss/genericjms/main/META-
INF/ra.xml
```

7.2. Navigating the contents of an archive file

We navigate into a .jar file and extract a .properties file defining messages. The .jar file has a name pattern “mod_cluster*”. The .properties file is found by looking for the string “# error messages”:

```
fox "/wildfly101//mod_cluster*.jar
     /#archive#//*.properties[grep('# error messages')]/file-content()"

=>
# Regular messages
modcluster.advertise.start=Listening to proxy advertisements on {0}:{1}
modcluster.context.disable=Undeploy context [{0}] from host [{1}]
modcluster.context.enable=Deploy context [{0}] to host [{1}]
…
```

7.3. Navigating the contents of BaseX databases

We list the names of all BaseX databases containing XSD files:

```
fox "basex://*[.//*.xsd]"
```

7.4. Navigating an SVN repository

We access the public SVN repository of the Apache foundation and extract the names of developers involved in the xerces project:

```
fox "svn-https://svn.apache.org/repos/asf
     /xerces/xml-commons/trunk/status.xml
     \\developers\person\@name
     => string-join(', ')"
=>
Shane Curcuru, David Crossley, Neil Graham, Ilene Seelemann, Norman ▶
Walsh,
Michael Glavassevich, Morris Kwan, Jeremias Maerki, Cameron McCormack,
Volunteer needed
```

7.5. Navigating a UTREE file system

In a preparatory step, we create a UTREE file system of the github.com projects published by the organisation MarkLogic. We use the `lifis` tool:

```
lifis "github?org=marklogic,format=utree" > /utree/github/ml/utree-ml.xml
```

Now we can navigate this system, specifying the UTREE folder via `fox` option –t. We create a report of dependencies defined by any of the POM files in any of the MarkLogic projects:

```
fox -t /utree/github/ml
  "https://github.com/marklogic//pom.xml
   \\*:dependency
   \concat(rpad(*:groupId, 35, '.'), ' ',
           rpad(*:artifactId, 30, '.'), ' ',
           *:version)
   => distinct-values() => sort()"

=>
...
log4j ............................ log4j ....................... 1.2.17
net.sf.opencsv ................... opencsv ..................... 2.3
net.sourceforge.htmlcleaner ...... htmlcleaner ................. 2.4
net.sourceforge.openutils ........ openutils-log4j .............
org.apache.avro .................. avro-tools .................. 1.7.4
org.apache.commons ............... commons-csv ................. 1.2
org.apache.derby ................. derby .......................
org.apache.hadoop ................ hadoop-annotations ........... 2.6.0
...
```

In order to improve performance, we load the document into a BaseX database which we call `utreebase`. We repeat our reporting, now specifying the UTREE *database*, rather than the UTREE folder:

```
fox -t basex://utreebase "https://github.com/marklogic//pom.xml …"
```

7.6. Navigating a UGRAPH file system

In the previous section we created a literal file system of the UTREE type in order to navigate the github projects of MarkLogic. We may alternatively create a literal file system of the UGRAPH type:

```
lifis "github?org=marklogic,format=ugraph" > /ugraph/github/ml/ugraph-
ml.ttl
```

After loading the triples into a TDB database (5):

```
tdbloader --loc=/tdb/ugraph-github-ml /ugraph/github/ml/ugraph-ml.ttl
```

we start a Fuseki server (5), which exposes the database as a SPARQL endpoint:

```
fuseki-server --loc=/tdb/ugraph-github-ml /marklogic
```

We repeat our reporting step, now specifying the UGRAPH endpoint via option -g:

```
fox -g http://localhost:3030/marklogic
  "https://github.com/marklogic//pom.xml
   \\*:dependency
   \concat(rpad(*:groupId, 35, '.'), ' ',
           rpad(*:artifactId, 30, '.'), ' ',
           *:version)
   => distinct-values() => sort()"
```

8. Discussion

The FOXpath language extends the XPath language by adding support for file system navigation. Navigation is modelled in terms of new expressions which map a URI to other URIs, based on structural relationships within a tree of URIs (e.g. asserting a URI to be a descendant or the parent of another URI). Expression semantics make no assumptions about the rationale of these relationships. In particular, they may be established by coexistence within a physical file system, but also by coexistence within a virtual file system, which attains the appearance of a file system due to APIs or applications. Literal file systems, finally, enable a complete decoupling of the original resource URIs and the URIs used for navigation, so that the relationships driving navigation can be constructed independently of coexistence and relationships within a physical or virtual file system.

What has been called "file system navigation" throughout this paper might therefore more aptly be called URI navigation. The generic nature of this added functionality points to a certain lack of concepts as how to bind potential use to actual, implementation-defined uses. This paper should be seen as a tentative

step towards closing the gap and also towards proving the practical value of the new potential.

URI navigation is about discovering resources of interest. The discovery is based on relationships between URIs, which are not established by explicit hyperlinks. The relationships are implied by the structure of the URIs themselves, which conveys a "place" occupied within a system of URIs. A navigation language like FOXpath, which lets us take advantage of URI structures in an unprecedented way - in the same way as XPath lets us take advantage of XML structure - might encourage a broader interest in a potential hitherto by and large ignored.

Bibliography

[1] BaseX: XML database and XQuery processor. http://basex.org

[2] EXPath Community Group. Homepage. https://www.w3.org/community/expath/

[3] Rennau, Hans-Jürgen. "FOXpath - an expression language for selecting files and folders." Presented at Balisage: The Markup Conference 2016, Washington, DC, August 2 - 5, 2016. In Proceedings of Balisage: The Markup Conference 2016. Balisage Series on Markup Technologies, vol. 17 (2016). doi: 10.4242/BalisageVol17.Rennau01. http://www.balisage.net/Proceedings/vol17/html/Rennau01/BalisageVol17-Rennau01.html.

[4] github - how people build software. Homepage.https://github.com/

[5] Github Developer - APIhttps://developer.github.com/v3/

[6] Apache Jena - A free and open source Java framework for building Semantic Web and Linked Data applications. https://jena.apache.org/

[7] Robie, Jonathan et al, eds. XML Path Language (XPath) 3.0 W3C Recommendation 8 April 2014. http://www.w3.org/TR/xpath-30/

[8] Robie, Jonathan et al, eds. XQuery 3.0: An XML Query Language. W3C Recommendation 8 April 2014. http://www.w3.org/TR/xquery-30/

[9] Walsh, Norman et al, eds. XQuery and XPath Data Model (3.0). W3C Recommendation 8 April 2014. http://www.w3.org/TR/xpath-datamodel/

DHW: An online introductory toolset for XML encoding

Alejandro Bia
Miguel Hernández University (UMH), Spain
<abia@umh.es>

Abstract

In this paper we will describe a set of online tools built initially for teaching XML encoding, though they can be used for production as well.
This set of tools comprises:

- *An online platform with tools to validate, pretty-print, edit and transform XML documents.*
- *Tools for automatic XML-TEI markup from a lightweight markup language.*
- *Tools to graphically visualize and design markup vocabularies and XML document instances.*
- *XSLT processing.*
- *XPATH evaluation.*

These tools will be briefly showcased during the conference presentation.

Keywords: XML-TEI, text encoding tools, automatic markup, visualization

Acknowledgements: This work has been developed within the TRACE project: Software Tools for Contrastive Analysis of Texts in Parallel Bilingual Corpora, and has been parcially financed with aid FFI2012-39012-C04-02 from the MINECO (Ministry of Economy and Competitiveness of Spain).

1. The DHW platform

The DHW (Digital Humanities Workbench) platform integrates a wide collection of useful XML-TEI related tools, starting at the creation of the Miguel de Cervantes Digital Library in 1999 and finishing with the more recent TRACE project which served to integrate and evolve these tools to make working with XML and TEI faster and easier, while favouring mobility and portability by offering them as online services.

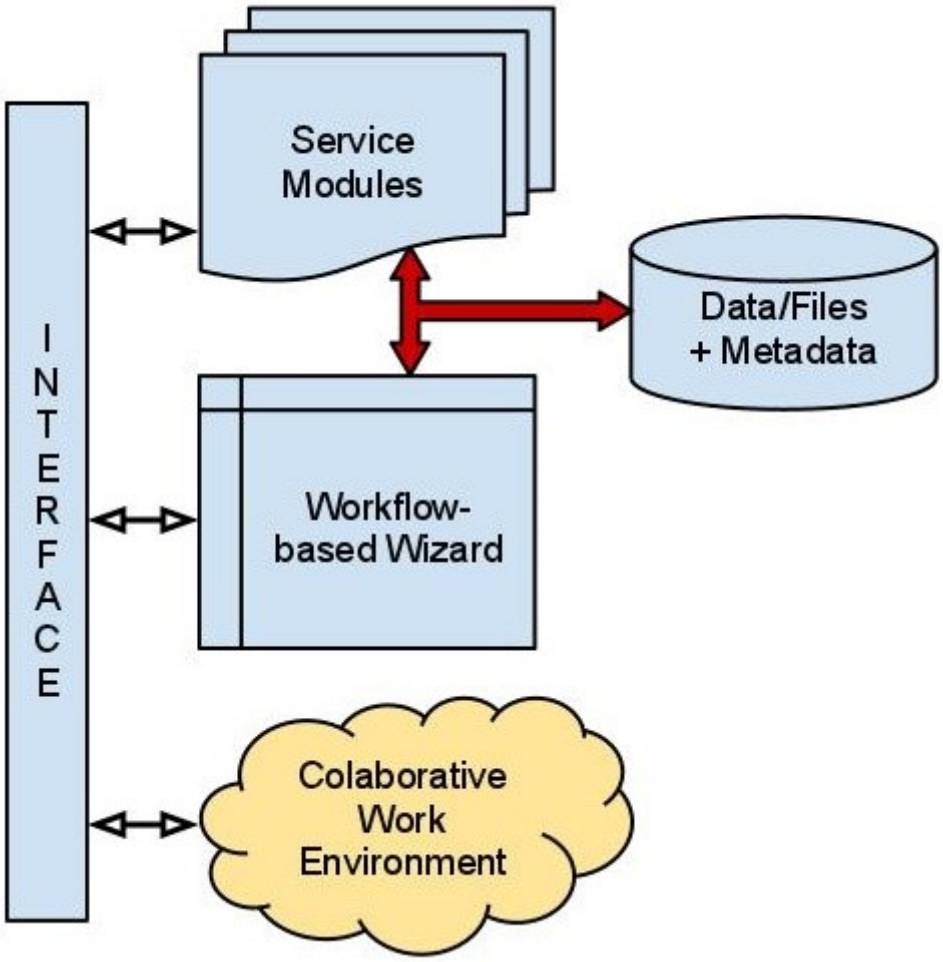


Figure 1. DHW: An integrated online collaborative working environment

This set of tools comprise:

- Tools related to XML documents: TEI, DTDs, Schemas, XSLT processing, Xpath, HTML, etc.
- Tools to graphically visualize and design markup vocabularies and XML document instances.
- Tools for automatic TEI markup from a Markdown-style lightweight markup language.
- Tools to validate, pretty-print, improve and transform XML documents.
- Tools to render XML documents
- Tools related to image processing.

Some of the tools used in this project already exist, and the rest are of our own breed. Not all of them are currently available, neither as online services, nor for download. The tools that are available for download are not generally available as web services, so they need to be installed according to different requirements

and be configured properly. For examples of some of these tools see: DiRT , Cover Pages , Garshol's XML Tools , TEI Tools . In some cases, there was some re-engineering of the tools to adapt them for client-server operation. We realized we could do this with tools that performed in an extended filter fashion. Apart from tool integration as web services, a virtual desktop with file management capabilities has been developed to provide a complete integrated work environment.

The idea of having the tools adapted to run as web services (see figure 1), allows for anyone to use them from anywhere, anytime, without the need of installation. This is particularly useful for Digital Humanities hands-on courses, where one of the main problems is to get all the necessary tools installed before the course takes place. This set of online services will also suffice for emerging or small digitization projects, and as a display and workbench for DH scholars looking for tools.

Furthermore, some intelligent piping of the tools can be arranged to follow certain processing workflows required for digital libraries or DH document processing. The platform can serve as a playground for workflow and task pipelining experiments.

Following the Software as a Service delivery model (SaaS), the purpose was to set up a web-server able to run different types of tools, and to develop a user friendly front end to allow beginners to operate the tools through a web browser (thin client) without the need for installation and configuration of myriad computer programs. In this sense, the DH-Workbench would serve as a teaching aid for beginners, and as an entry level solution for emerging DH projects, allowing for savings in software and installation costs.

The implementation is based on the popular and reliable XAMPP platform (Apache HTTP Server, MySQL, PHP, Perl), plus the Java Runtime Environment (JRE) and eventually any other runtime processor required by the tools to run (Saxon, Xalan, Phyton, etc.). Only a Web browser is needed on the client side.

The following is a non-exhaustive list of the services and tools included in the DH Workbench:

- dtd2xs : Dtd2Xs allows conversion of complex, modularized XML DTDs and DTDs with namespaces to W3C XML Schemas.
- DTDinst : Converts DTDs to XML-DTD, i.e. DTD structure represented in XML format. Useful for XML processing of DTDs.
- DTDprune: A DTD simplification tool (Bia and Carrasco 2001).
- Mindmap diagrams generated from XML document instances to visualize and analyze the document structure (Bia, Munoz and Gómez 2010).
- Mindmap diagrams generated from DTDs and Schemas to visualize and analyze the structuring rules (Bia, Munoz and Gómez 2010).

- Multilingual Markup Translator (previously called "Multilingual Markup Website") (Bia, Malonda and Gómez 2006)

- rxp : RXP is a validating XML parser written in C.

- Tidy : HTML Tidy is a computer program and a library the purpose of which is to fix invalid HTML and to improve the layout and indent style of the resulting markup. It was developed by Dave Raggett of W3C, then passed on to become a Sourceforge project.

- TEIdown: automatic markup tool that converts a document with a very simple and easy to apply lightweight markup to a valid TEI document (Bia 2015).

- Trang : Receives DTDs, Relax-NG and Relax-NC Schemas, and XML document instances as input, to produce DTDs, XML Schemas (XSD), and Relax-NG and NC Schemas. Trang can also infer a schema from one or more example XML documents.

- xmllint : The xmllint program parses one or more XML files, specified on the command line as xmlfile. It prints various types of output, depending upon the options selected. It is useful for detecting errors in XML documents.

- XSLT ready-made transformations: Several standard ready-made transformations (e.g. tei2html).

- XSLT online transformations: This is an online service providing transformations of XML document instances by user-provided XSLT scripts, using one of several parsers offered: csxslt, MSXSL, Saxon (versions 6, 7 and 8), Xalan, Xerces, xml4j, xsltproc and XT

All these tools have been integrated in the DH workbench using a bilingual (English/Spanish) web interface that includes a file manager (see figure 2) that resembles a conventional file explorer, and a basic online editor (figures 4 to 6).

The file management area allows operations like: file upload and download, copy, move, rename, delete and selection of files for editing or processing (see figure 3). The editing view, from which transformations and processing functions can be launched, offers basic editing facilities to work with up to three files in parallel folders: e.g., input, transformation and output files (see figures 4, 5 and 6 respectively).

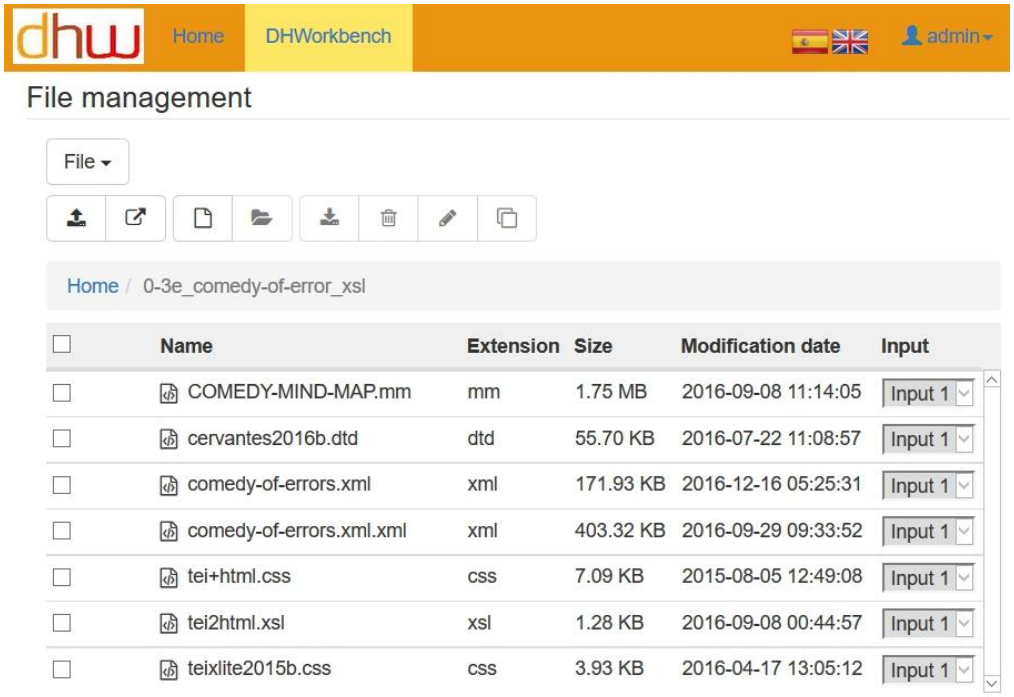


Figure 2. DHW file manager

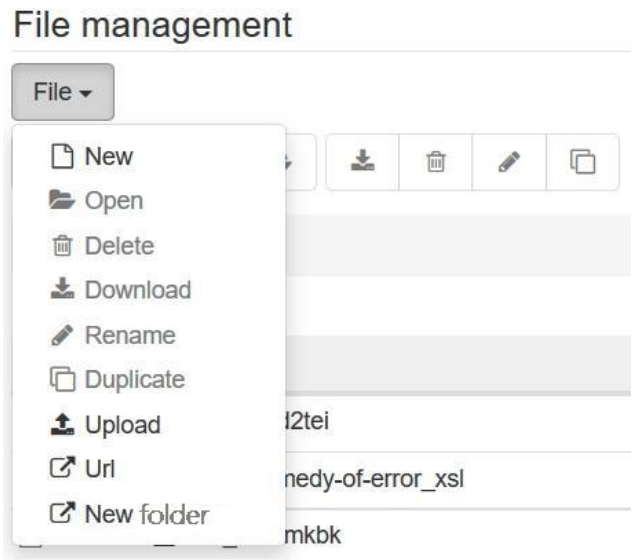


Figure 3. File options

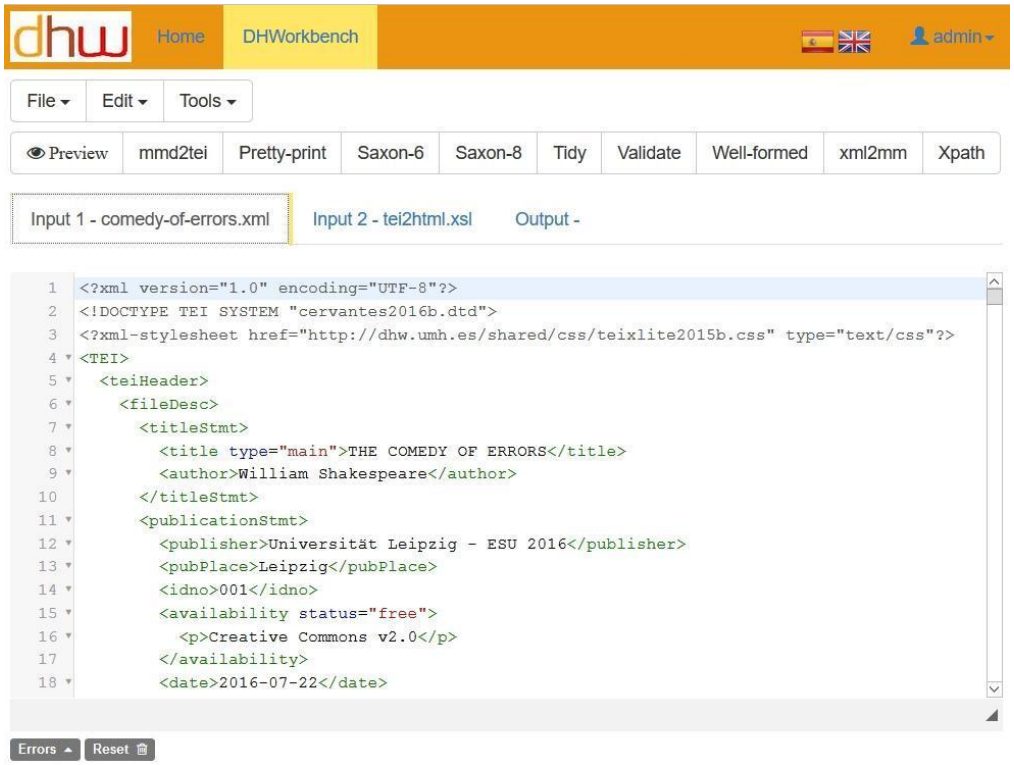


Figure 4. XML Editor: Input 1 (XML document instance)

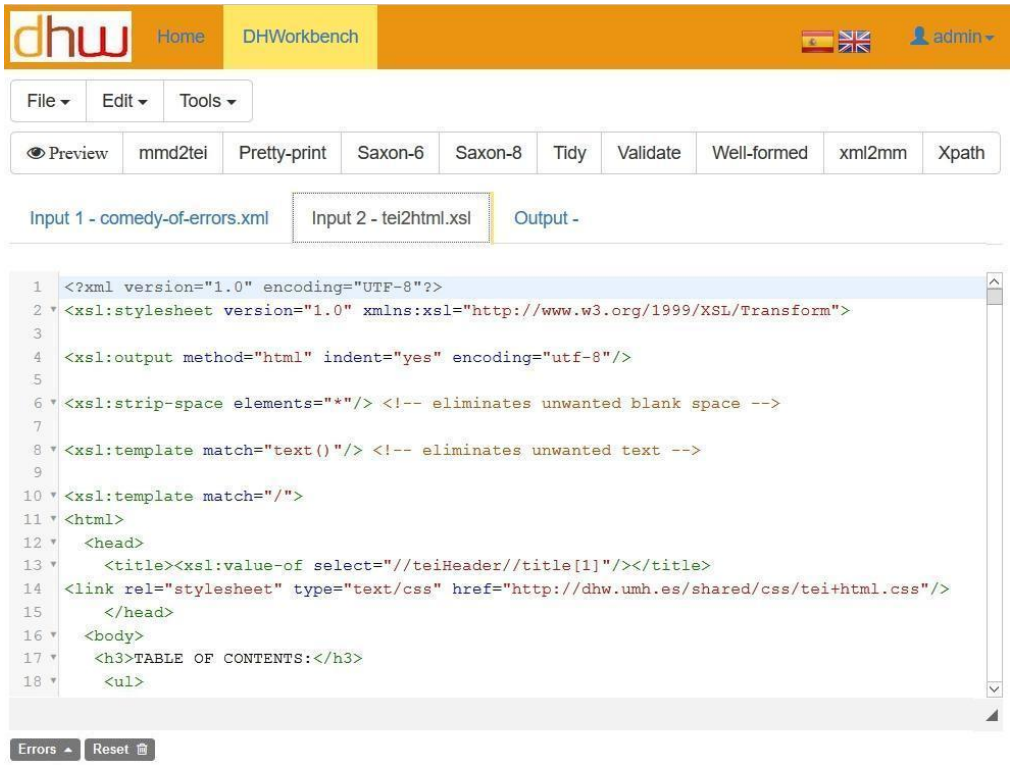


Figure 5. XML Editor: Input 2 (XSLT script)

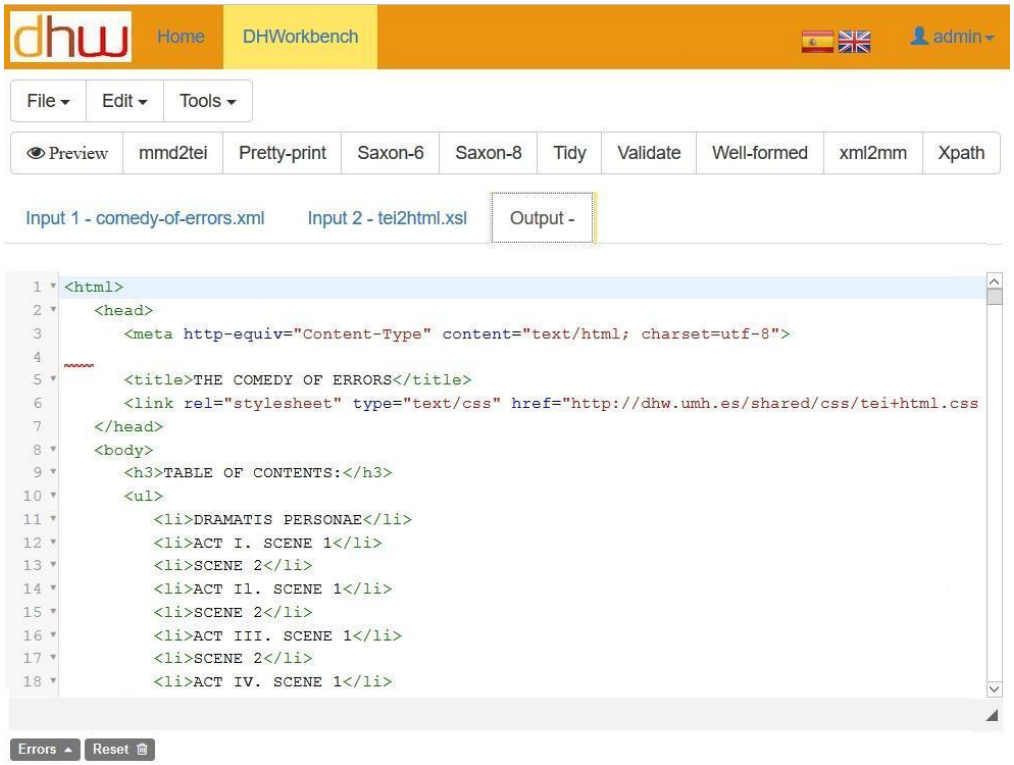


Figure 6. XML Editor: Output (HTML file)

The online application allows for three types of users: not registered (can use the online tools but cannot store documents in the cloud), registered users (have their own work area where documents can be left to be used in future work sessions), and administrators (can perform application management tasks).

We hope that this online platform will serve to test and share new tools in the future. Hence, easy maintenance and upgradeability were amongst the main design goals of the project.

Although power tools like these can enhance training and production activities, we have to agree with Schreibman and Hanlon that 'tool development is indeed considered a scholarly activity by developers, but recognition of this work and rewards for it lag behind rewards for traditional scholarly pursuits (such as journal articles and book publication)' (Schreibman and Hanlon 2010). In this sense is not always easy to find support and recognition for this type of projects.

2. Tools for automatic markup

Creating new XML (eXtensible Markup Language) documents, from scratch or from plain-text, can be a difficult, time consuming and error prone task, especially when the markup vocabulary used is rich and complex, as is the case of the TEI (Text Encoding Initiative) with a vocabulary of more than 500 different tags. It usually takes a good amount of time to make the document validate for the first

time, and the errors that appear in the process can be many and hard to locate and repair.

A couple of decades ago, SGML (Structured Generalized Markup Language) allowed certain freedom to document encoders, meant to save time and effort, like leaving certain tags open, or omitting the quotes of attribute values. In this sense, SGML was more permissive than XML. This was good for document encoders, but made it difficult for programmers to create parsers and applications that fully complied to SGML's permissive rules and inferences. On the contrary, XML was meant to be very restrictive, and hence, more predictable, which makes parsing and processing easier and contributes to the fast popularity that XML gained soon after its introduction.

On the other hand, in the Wiki world (web sites where the reader can edit and add contents directly from the web browser), a myriad of Wiki languages emerged with the purpose of simplifying, or completely avoid, HTML markup. Among these lightweight markup languages, Markdown (Gruber, 2004) is a recent and very successful shorthand notation, adopted, in some cases with variants, by several successful projects like GitHub, reddit, Diaspora, Stack Exchange, OpenStreetMap, and SourceForge, to avoid writing HTML tags while still keeping text legibility intact.

Merging the spirit of SGML with the principles of Markdown, we came to the idea of TEIdown (Bia 2015), which consists of an extension of the markdown syntax meant for the easy creation of TEI-XML documents. We implemented this idea by creating the corresponding parsers needed to perform such conversion (see figure 7). The parser generates TEI from extended multimarkdown (mmd): '.dtei' files. In the prototype, we separate metadata from body text for separate processing (this will not be done in the final version). In this way, we obtain an intermediate file that is not yet TEI, but 'protoTEI'. Then an XSLT transformation puts everything in place and performs the last fixes. We obtain a valid document, although it may require further markup.

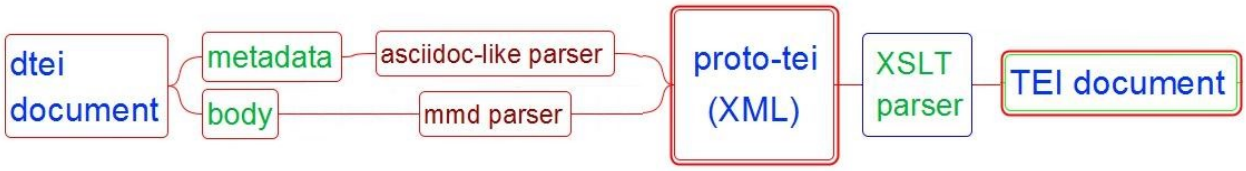


Figure 7. Prototype parser built as proof-of-concept

With this approach, it is easy to obtain a valid TEI document in a very short time, avoiding going through a long list of validation errors (see figures 8 and 9). The approach, however, has some limitations. It is meant to process the most common tags, like the ones used for prose and verse, and the most commonly used for metadata (within the teiHeader section of the document). For specialized applications (like manuscripts, for instance), further tagging is necessary after the initial

conversion, but even in such cases a significant amount of time is saved in the process.

```
# Carta abierta #
docAuthor: Ricardo Güiraldes

Amigos:

He leído hoy el último número de MARTíN FIERRO. Patria chica en el papel, grande en el anhelo. Vuestra juventud sube hacia mi rostro, como un aliento de pampa, cuando sobre la gramilla iluminada de rocío (emoción de la madrugada, que vuelve a encontrar su mundo) me aferro al optimismo ascendente de los nuevos crecimientos. El hombre se siente pequeño ante la infinita transmutación que anuncia lo porvenir, pero crece con sentirse capaz de comprenderla.

En vuestra valentía se produce una vez más el eterno amanecer del espíritu. ¿No es ése el misterio de la anunciación?

Vienen y vendrán los ataques. Inútil sorprenderse. Caminos sin pantanos no son caminos de hombres libres y los más duros son los que más espolean el deseo de llegar. Caer no es nada. Las osamentas sirven de mojón a los que después de uno sienten el vértigo del desierto. Así se conquistan horizontes. Así se regala el bien habido a los timoratos.

Uno de los nuestros ha pedido piedad. Y es que dio el rostro a lo que siempre debió volver la espalda. ¿Cómo la cobardía momentánea del fuerte puede pedir ayuda a la cobardía constante de los débiles? Dice un refrán gaucho: "No hay que mudar caballo en medio 'el río'".

En el camino de las ideas la duda equivale a mudar de caballo.

Además,¿qué puede esperar el que cargó sobre sus hombros con la responsabilidad de partir, de aquellos que se aferraron a la inmovilidad? Solamente un reproche, una acusación de soberbia, de
```

Figure 8. Example of input text encoded using dtei extended markdown.

```
<text lang="es">
  <front>
    <titlePage>
      <docTitle>
        <titlePart>
          <title type="main">Carta abierta</title>
        </titlePart>
      </docTitle>
      <docAuthor>Ricardo Güiraldes</docAuthor>
    </titlePage>
  </front>
  <body>
    <div>
      <p>Amigos:</p>
      <p>He leído hoy el último
número de MARTíN FIERRO. Patria chica en el papel, grande en el
anhelo. Vuestra juventud sube hacia mi rostro, como un aliento de pampa, cuando
sobre la gramilla iluminada de rocío (emoción de la madrugada,
que vuelve a encontrar su mundo) me aferro al optimismo ascendente de los
nuevos crecimientos. El hombre se siente pequeño ante la infinita
transmutación que anuncia lo porvenir, pero crece con sentirse capaz de
comprenderla.</p>
      <p>En vuestra valentía se produce una
vez más el eterno amanecer del espíritu. ¿No es ése
el misterio de la anunciación?</p>
      <p>Vienen y vendrán los ataques.
Inútil sorprenderse. Caminos sin pantanos no son caminos de hombres
```

Figure 9. Example of output XML-TEI encoded text.

3. Visualization

Visual modelling: is the graphic representation of objects and systems of interest using graphical languages. Creating information models for use in SGML or XML applications is not like designing a data model for a conventional database application. A typical information model for an SGML or XML application will be much less constraining about the structure of information. Whereas a data model for a classical database application will be very precise about the number and order of fields, and the length of the content of fields, a typical SGML or XML information model will allow greater variation in the number, sequence and content of the structural components (elements)

The OHCO model (Ordered Hierarchy of Content Objects), is a way to define text in computer terms. [Steven J. DeRose, David G. Durand, Elli Mylonas, and Allen Renear, 'What is Text, Really?']

Each content object is a chunk of text marked at its beginning and end with opening and closing tags. Content objects can contain both text and other content objects. The nesting of these contained text chunks convert the document into a tree of hierarchically contained objects. The OHCO model is inherent in SGML,

and its descendants: HTML and XML. OHCO is a useful, but imperfect model. It is more powerful than a model of text as a stream of characters and formatting instructions, but one of its limitations is that it does not allow for overlapping content objects. A book is a good example of what can be represented in this way.

A good model must be capable of hiding unnecessary detail. This is precisely one of the problems that we had when we used available UML tools with stereotyped Class diagrams:

- Whole DTD/Schema diagrams were too complex to handle and display,
- There was no way to selectively fold parts of them.
- Lack of efficient automatic arranging of visual objects while importing a DTD/Schema

All this made our attempts impractical for real-application purposes. So, it is not only the type of diagram, but the tool is also important.

Concerning visualization, we adapted mind maps to successfully visualize, model, design, modify, import and export XML-TEI schemas, including DTDs (see figure 10), as well as TEI document instances (see figure 11). Using Freemind, a software tool for drawing mind maps, and XSLT transformation scripts, we get very manageable, easily comprehensible, folding diagrams from XML sources, which in turn can be edited in a graphical environment and converted back to their original XML format. In this way, we adapted a general purpose mind-mapping tool, into a visual tool for XML vocabulary design and simplification. This approach is also very useful for teaching and presentation purposes.

It is frequently said that a good model must be capable of hiding unnecessary detail. The ability to interactively hide/unhide branches of a mind map diagram, and the automatic allocation of nodes around a central point is what makes Freemind so attractive to our purposes of representing semistructured document structures. User friendly features for copying, pasting, moving, dragging-and-dropping subtrees make it ideal for visual structure design. For this we needed a way to import and export several types of schemes. So we implemented transformations for the most popular notations: DTDs, W3C XML Schema and RelaxNG.

Freemind uses an XML file format, which can be generated using XSLT scripts. We have written several scripts to translate DTDs, XSD and Relax NG Schemas to and from Freemind's file format. In the case of DTDs, whose syntax is not actually XML, we used some additional pre/post processing.

4. Conclusions

- ONLINE TOOL INTEGRATION: We have built an online service to provide a wide collection of useful text processing tools.
- AUTOMATIC MARKUP: The proposed method significantly reduces the time required to markup an XML-TEI document. It is good both, for expert encod-

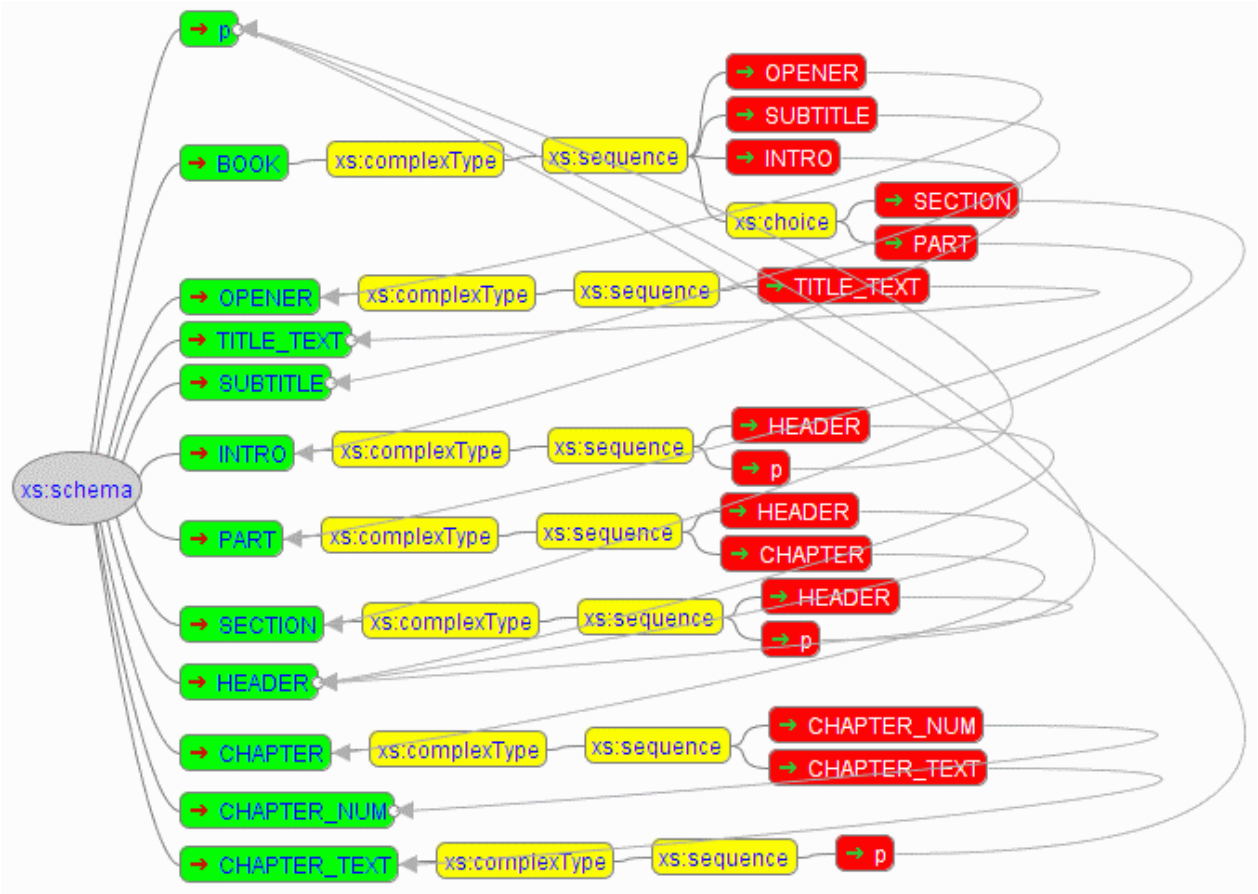


Figure 10. A simple example of XML Schema represented as a Mind Map. (the root node `BOOK' is the only node not pointed from any other content model)

ers, and, particularly, for beginners, reducing frustration, since the generated document validates immediately. It may also speed-up the learning curve.

- VISUALIZATION: We managed to use mind maps and the Freeplane tool to successfully import, model and generate DTDs, Schemas and also XML instances, getting very manageable, easily comprehensible, folding diagrams. In this way, we converted a general purpose mind-mapping tool, into a handy tool for XML vocabulary design and visualization (for teaching XML-TEI markup, or just for presentation purposes)

- STUDENT INVOLVEMENT: Involving advanced students proved to be a good experience of merging research and teaching, where several groups participated in a creative environment building prototypes to test different aspects of the problem.

- PROTOTYPING: was useful as a means to gather design information from future users, and to discuss with other designers.

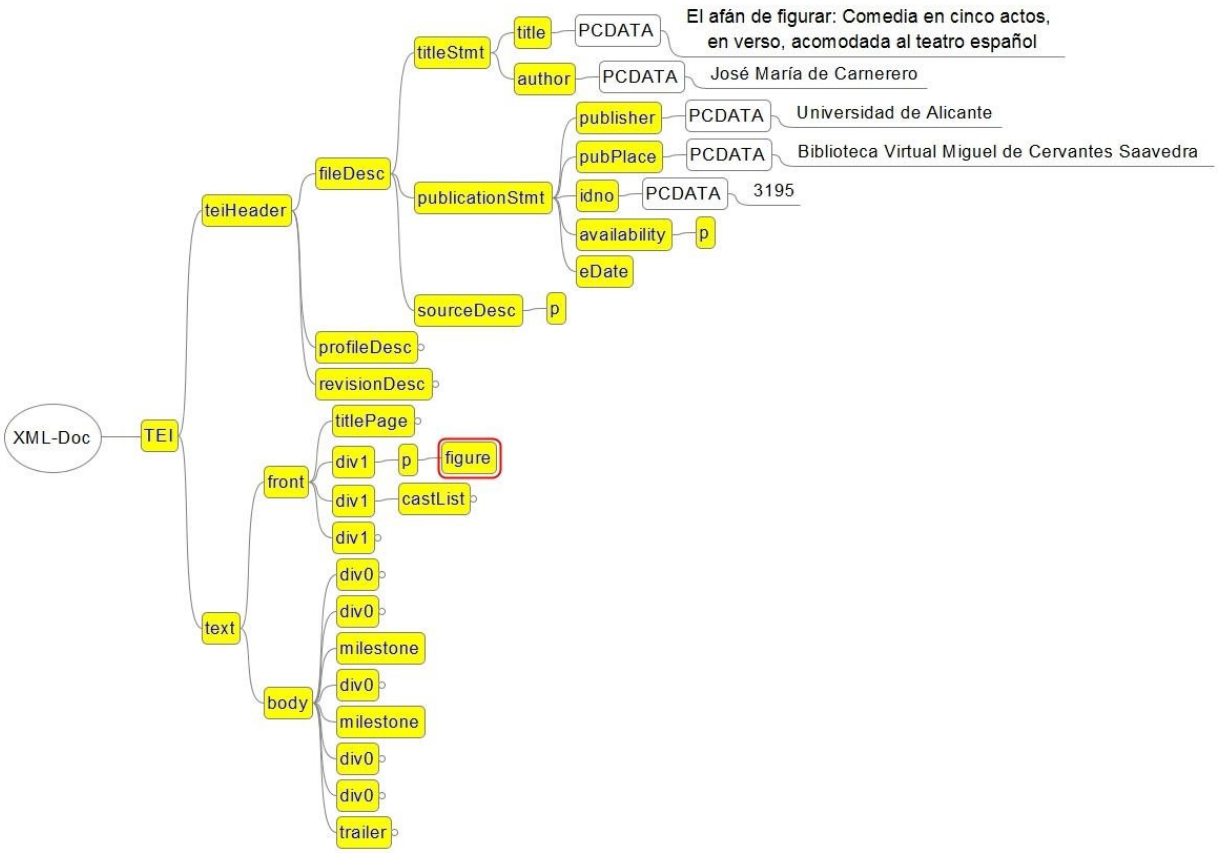


Figure 11. An example of an XML-TEI document instance.

References

[1] Bia, A., and Carrasco, R.: Automatic DTD simplification by examples. In ACH/ ALLC 12001. The Association for Computers and the Humanities, The Association for Literary and Linguistic Computing, The 2001 Joint International Conference New York City, New York University, pages 7–91, 2001.

[2] Bia, A., Malonda, J. and, Gómez, J.: The Multilingual Markup Website. In Digital Humanities 2006: The First ADHO International Conference, C. Sun, S. Menasri 1 and J. Ventura, editors Paris, Universite Paris-Sorbonne, pages 26–31, 2006.

[3] Bia, A., Munoz, R., and Gómez, J.: Using Mind Maps to Model Semistructured Documents. Lecture Notes in Computer Science, 6273:421–424, 6–10 Sept. 2010.

[4] Bia, A: Down to TEI: use of extended markdown to speed-up the creation of TEI documents. In 15th TEI Conference and Member's Meeting Universite Lumiere Lyon 2, Lyon, France, 2015.

[5] Gruber, J.: Markdown. (accessed on 15/09/2016) http://daringfireball.net/projects/markdown/

[6] Schreibman, S., and Hanlon, A. M.: Determining Value for Digital Humanities Tools: Report on a Survey of Tool Developers. Digital Humanities Quarterly, 4(2), 2010.

A Text Structure "Epischema" for TEI

Constraining a generic Relax NG schema with an additional Relax NG schema

Gerrit Imsieke
le-tex publishing services GmbH
<gerrit.imsieke@le-tex.de>

Abstract

This paper presents an underutilized mechanism for XML document grammar customization. Instead of altering the base schema or adding Schematron constraints, a second grammar-implementing schema is associated with the document. This second schema will enforce structural constraints where the basic schema is liberal. This second schema is lightweight in that it allows anything anywhere except for a certain aspect for which it adds grammatical constraints over the permissive base schema. We call this additional, sparse, aspect-oriented schema an Epischema. An example to which this concept is applied is TEI's notoriously generic `div` *hierarchy, where the* `div/@type` *attribute can assume arbitrary values. Generic vocabularies such as TEI and HTML are increasingly used by publishers as the primary source format. These publishers ask for prescriptive constraints to be imposed on top of basic schema conformance. An advantage of epischemas over the commonplace Schematron constraints is that they allow better context-aware markup completion in authoring systems.*

Keywords: XML Schema, Relax NG, Schematron, Schema customization

1. Introduction

It is common in publishing, digital humanities, or technical writing that XML schemas are subject to customization. Popular XML vocabularies such as DITA [1] and TEI [2] are designed to be customized.

Often a customization is derived from a library of modules – such as for drama, dictionaries, or critical apparatus – by omitting modules entirely or by omitting or adding module members, to wit elements, attributes, and attribute values.

A common approach is to have a comprehensive schema for XML experts and derivative schemas that reduce choice for authors or copy editors. For example, JATS comes in three flavors, archive and interchange, publishing, and authoring [3], ordered by increasing prescriptiveness.

For publishers, it is difficult to find the right balance between flexibility and prescription. On the one hand, it is important to be able to find adequate structures in order to best model a given piece of content, and content can be quite diverse within a single publishing house. On the other hand, even for experts there is often not consensus which markup to apply to a given content. You can find examples for these kinds of discussions in the TEI-L archives for any given month [4]. Therefore, in order to contain complexity of both rendering and authoring systems, and to make a content repository homogeneous enough for meaningful analysis, additional constraints are to be exerted on top of generic schemas. This can either be done by creating derived schemas by means of several extension mechanisms or by applying independent rules, typically in the Schematron language [5] (see Section 3). A zoo of derived Schemas raises concerns of maintainability and configuration management, favoring schema generation from configuration files and composable constraint libraries over "handcrufted" customizations.

1.1. Motivation

The current work was motivated by a group of German-language publishers of fiction, non-fiction, and of scientific literature in the humanities.[1] They wanted to use a common XML vocabulary for their publication workflows. Of the widespread and well-maintained vocabularies DocBook, JATS (specifically, the BITS customization [7]), TEI, and HTML (specifically HTMLBook [6]) they selected TEI as the XML schema for integrated print/electronic book production. What tipped the scales in favor of TEI was the consideration that a small but important percentage of their book projects is funded by Deutsche Forschungsgemeinschaft (DFG). DFG usually requires projects in the humanities to deliver their content as TEI XML [8], preferably according to the TEI-Simple customization [9].

The problem with TEI is that it is quite non-prescriptive when it comes to modeling the document structure (see Section 2). In order to be able to use a common basis of conversion pipeline adaptations [10], and in order to be able to produce EPUBs in the publisher's layout from TEI that was prepared by other means, we looked into harmonizing the use of div types across the TEI-adopting transpect [11] users. In order to produce reliable TEI input, it is not only necessary to limit the div attribute values to a fixed set. It is rather necessary to define which typed divs may occur in which elements, at which position, and in which cardinality. This very much sounds like a grammar problem that can be expressed by a schema.

Another motivation came from a request on TEI-L [12] that was about restricting the top-level div's model in a TEI body and make oXygen take this model change into account when issuing completion suggestions. Restricting the model

[1] C.H. Beck, Beltz, Campus, Hanser, Klett-Cotta, Suhrkamp, Unionsverlag

seemed like a task for Schematron. However, for content completion, inferring the allowed elements and attributes at a given position is only implemented for Schemas, not for Schematron constraints on top of a Schema.

In addition, it turned out that context-specific schema customizations are impossible with the TEI ODD mechanisms and utterly cumbersome with Relax NG customization mechanisms, which made the author explore the epischema approach to this problem, too.

2. Generic Sectional Hierarchy Elements in TEI

TEI offers two approaches for modeling the hierarchic organization of a work, by unnumbered (`div`) and by numbered (`div1`, `div2`, …) element names [13]. The `@type` attribute may be used to name each structural unit (chapter, part, section, …) according to a community's or a publisher's conventions.

As an example, think of a textbook publisher whose books all follow the same structure:

- Frontmatter **?**
  - User Guide **?**
  - Preface **+**
    - Section *
- Mainmatter
  - Unit **+**
    - Intro **?**
    - Part **+**
      - Intro **?**
      - Lesson **+**
        - Section *
        - Exercise *
- Backmatter
  - (Glossary **?** | Vocabulary **?**)

When textbook content is authored as XML or when printed works are converted to XML, the publisher wants to make sure that the structure complies with these rules. XML schemas are well suited for checking these structural constraints and for giving document authors meaningful suggestions. Except when everything is a `div` and any `div` may contain any type of `div`.

TEI's div, like many other hierarchization methods in other grammars, do not allow paragraph-like content after a div has ended.[2] To circumvent this restric-

[2]This restriction has recently been questioned on TEI-L [14].

tion, a generic element, `floatingText`, was introduced. This is a device designed to hold embedded structures such as sidebar boxes or letters. It can appear anywhere where paragraphs are permitted, and said German publishers need a fixed set of these floatingText types. In principle, both these floatingTexts and the structural divisions that they may contain enjoy the freedom of arbitrary typing. This is certainly a good thing for scholars who can select a type vocabulary that they deem appropriate to mark up a given source text. It is, however, overly permissive when the goal ist to provide guidance to authors and transformation tool developers. Therefore we can state that TEI's sectional division approach is *Behlendorf-flexible* [15]:

The advantage of TEI's document structuring approach is its flexibility, while the drawback is its flexibility.

Let's suppose though that the textbook publisher or the abovementioned transpect/TEI-adopting publishers have good reasons to select TEI as their XML format, and that they want to adapt it to their needs by adding detailed and prescriptive hierarchy modeling.

3. Approaches for Augmenting the Model

Several approaches seem feasible:

1. HyTime Architectural Forms [16]. The idea is to have a front-end DTD that maps its custom elements (`textbook-unit`, `textbook-exercise`, etc.) to TEI typed divs. Then a more refined grammar can be defined for the custom elements, while after HyTime processing, they appear as TEI divs.

   On second thought, SGML is as dead as many Web pundits consider XML to be (and as the author wishes XML DTDs were), so please regard this suggestion as a joke.

   In a sense, this paper presents the antithesis to an architectural forms approach. Architectural forms: Different tags, same underlying metamodel. Epischema: Same tags, different models.

2. Use the numbered element names (`div1` etc.) and implicitly know that in the frontmatter, `div1` is a Preface or a User Guide and in the main matter, it's a Unit.

3. Use one of the ODD customization mechanisms that are suggested in the guidelines [17], in particular:

   a. limiting the permissible values of the `div/@type` attribute

   b. adding new elememts (such as `unit` or `part`) to a TEI content model class (such as `model.divLike` [3] or `model.div1Like` [4]

[3] http://www.tei-c.org/release/doc/tei-p5-doc/en/html/ref-model.divLike.html
[4] http://www.tei-c.org/release/doc/tei-p5-doc/en/html/ref-model.div1Like.html

   c. changing the content model
4. Import an existing TEI schema (in RNG or XSD, DTD schema and redefine the content models). This is equivalent to the previous content model change, although it will be applied to an existing schema rather than to the ODD source.
5. Impose Schematron rules for these nesting rules: which `div`/ `@type` may appear in which context, for the types of `div`s that may precede or follow a `div` of a given type, for the maximum number of certain `div`s in a given context, etc.
6. Create an additional schema that will allow anything everywhere outside of `front`, `body`, `back`, and `div` contexts. In these contexts, it will apply the complex, context-dependent nesting rules for the given publication type.

Approach 3a allows only the `@type` attributes to be taken from a finite list. No hierarchical order whatsoever among typed `div`s may be prescribed by this kind of customizing.

Approach 2 doesn't really cut it, either: if any two elements that are allowed as `div1` are supposed to have differing content models, you can't use the same `div1` element. In the example above, the User Guide might only contain lists, paragraphs, and tables, while the preface may have a section substructure. Or suppose that a publisher wants a book to contain either parts or chapters as immediate children. Then you wouldn't know whether your `div1` elements are supposed to be chapters or parts. This may be regarded as nitpicking about arbitrary terms, but as soon as there are different content models involved (for example, only part, not a chapter, should be allowed to contain a minitoc `divGen`), this becomes unwieldy.

The same holds true for Approach 3b—if both `unit` and `part` belong to the same content model class, they share the same content model, which cannot be context-dependent.

The only approaches that seem suitable for context-aware content modeling are 3c/4, 5, and 6.

3.1. Changing the model (3c/4)

In order to specify what type of div may appear at which location in a book, we'd have to redefine the models for `div`, `front`, `body`, and `back`. Changing a model in Relax NG (which used to be ODD's mechanism, too) is only easy if the basic schema already provides all the hooks for redefining attributes and models at the required level of granularity (see Sect. 12.1.1 of Eric van der Vlist's Relax NG book [18]). Otherwise you'd have to define these constituents in first place, or live with unmanageable redundancy of redefining large portions of the original schema. This is the issue here. TEI's Relax NG schema granularity corresponds to

the granularity of its class and element definitions, which is not sufficiently fine-grained for permitting the addition of context-aware div nesting rules.

Apart from embedded Schematron, modern ODD [2] offers less flexibility than pure Relax NG in defining context-dependent models. This is probably due to the requirement that ODD should also be convertible to DTD, which lacks context-dependent models, too. This excludes approach 3c.

In any case, approaches 3c and 4 replace existing models with new ones. You'll always have to pay attention not to damage the existing model when deriving the new models, take the changes into account that other people may have made upstream, etc. In other words: it's a daunting task to redesign a model that wasn't designed to be redesigned at the required granularity.

3.2. Schematron (5)

It is feasible to convert a grammar-oriented schema (XSD, RNG, DTD) to Schematron. At least to an extent that is sufficient for the context-aware div nesting check. There's particularly Rick Jelliffe's long running XSD2SCH project for automatically converting a grammar to Schematron rules [19].

Everybody in XML processing should be able to come up with the Schematron rules for the context-aware typed `div` constraints. Everybody should have a Schematron checker at their disposal. Everybody may include these rules in an existing Relax NG schema without the need to tamper with the preexisting (and sometimes poorly extensible, see above) grammar. We have in fact used Schematron for these kind of additional constraints. So why shouldn't we?

We think that Schematron is the answer to many validation issues and is quite well suited to complement grammar-based checks. But if you have a grammar-based checker at your disposal and the task at hand looks grammatical—why shouldn't you just add your grammar like you'd add your Schematron, on top of an unaltered preexisting grammar?

The practical benefit of this approach might be context-aware content completion.

4. Add an Independent Schema (6)

4.1. Example 1: Constraining the Top-Level `div` Model

In order to demonstrate the feasibility of the epischema approach, we will inspect the (simpler) solution for whitelisting top-level div children of [12] first. See Section 4.3 for details of associating the epischema with the document.

The task at hand is to require a single `head` and to only allow `p`, `figure`, `list`, and `div` below a `div` that is immediately below `body`. Otherwise, the default `tei_all` model should prevail.

So in the following example, `ab` as a child of `body/div` should be invalid while `ab` and other elements are valid below `body/div/div` etc.

```
<body>
  <div>
    <head>Top-Level div</head>
    <p>allowed</p>
    <ab>not allowed (only p, figure, list, or div)</ab>①
    <div>
      <head>2nd-level div</head>
      <ab>allowed</ab>
    </div>
    <div>
      <ab>allowed</ab>
      <div>
        <head>3rd-level div</head>
        <ab>allowed</ab>
      </div>
      <ab>Not allowed here because of base schema</ab>②
    </div>
  </div>
</body>
```

① element "ab" not allowed here; expected the element end-tag or element "div", "figure", "list" or "p"

② element "ab" not allowed here; expected the element end-tag or element "addSpan", "alt", "altGrp", "anchor", "argument", "byline", "cb", "certainty", "closer", "damageSpan", "dateline", "delSpan", "div", "divGen", "docAuthor", "docDate", "epigraph", "fLib", "figure", "fs", "fvLib", "fw", "gap", "gb", "incident", "index", "interp", "interpGrp", "join", "joinGrp", "kinesic", "lb", "link", "linkGrp", "listTranspose", "meeting", "metamark", "milestone", "notatedMusic", "note", "pause", "pb", "postscript", "precision", "respons", "salute", "shift", "signed", "space", "span", "spanGrp", "substJoin", "timeline", "trailer", "vocal", "witDetail" or "writing"

There is no precise recipe for creating epischemas, but it usually starts with a named pattern that may be called `almost-anything`:

```
<define name="almost-anything">
  <choice>
    <element>
      <anyName>
        <except>
          <choice>
            <name>div</name>
            <name>body</name>
          </choice>
```

```
        </except>
      </anyName>
      <ref name="any-atts"/>
      <zeroOrMore>
        <choice>
          <text/>
          <choice>
            <ref name="body"/>
            <ref name="almost-anything"/>
          </choice>
        </choice>
      </zeroOrMore>
    </element>
  </choice>
</define>
```

When validation starts at the top-level element, it will accept any element that is not called `body` or `div`. As children, this any-element may have other any-elements, text, or `body`. If you think that this allows construction of all kinds of invalid TEI documents, you are right. It is important to note that we are not creating a modified TEI schema here but an epischema that is meant to be used in conjunction with a TEI schema proper.

When validation encounters a body element, it will check it against the following patten:

```
<define name="body">
  <element name="body">
    <ref name="any-atts"/>
    <choice>
      <zeroOrMore>
        <choice>
          <ref name="top-div"/>
          <ref name="almost-anything"/>
        </choice>
      </zeroOrMore>
    </choice>
  </element>
</define>
```

If there is a `div` below `body`, it must conform to the `top-div` pattern, which is defined as

```
<define name="top-div">
  <element name="div">
    <ref name="any-atts"/>
    <element name="head">
      <ref name="any-mixed"/>
```

```
      </element>
      <oneOrMore>
        <choice>
          <element name="p">
            <ref name="any-mixed"/>
          </element>
          <element name="list">
            <ref name="any-mixed"/>
          </element>
          <element name="figure">
            <ref name="any-mixed"/>
          </element>
          <ref name="regular-div"/>
        </choice>
      </oneOrMore>
    </element>
  </define>
```

with

```
  <define name="any-mixed">
    <ref name="any-atts"/>
    <zeroOrMore>
      <choice>
        <text/>
        <ref name="almost-anything"/>
        <ref name="regular-div"/>
      </choice>
    </zeroOrMore>
  </define>
```

and

```
  <define name="regular-div">
    <element name="div">
      <ref name="any-mixed"/>
    </element>
  </define>
```

It's important to note that `any-mixed`, the pattern that becomes ubiquitous just when we're past the top-level `div`, holds a reference to `regular-div`, which is a div that can contain virtually anything. This means that from this point on, the epischema will not stand in the way of what `tei_all` permits.

The epischema is available at [20] and an NVDL bundling of this epischema with the tei_all.rng "standard" TEI customization is at [21]. A sample document that is associated with the NVDL version is available at [22].

4.2. Example 2: Docbook-like `divs`

This epischema is a bit more complex than the first example. This is due to several factors:

- the number of individual context-dependent models
- the inclusion of `section`/`sect1` alternatives
- modeling of typed `floatingText`

The epischema's Relax NG file size is 22 kB large. However, trying to impose these restrictions on the base TEI schema by conventional extension mechanisms would have necessitated many times more of schema code.

We selected a TEI base schema with some extras here, SVG, MathML, CSSa [23], RDFa. An important advantage of an independent, on-top schema is that we don't need to care which base customization we use. This orthogonality is certainly an advantage that epischema shares with Schematron.

We encourage you to explore the epischema [24] and the sample document [25]. You will see one advantage over Schematron and that is content completion for section types.

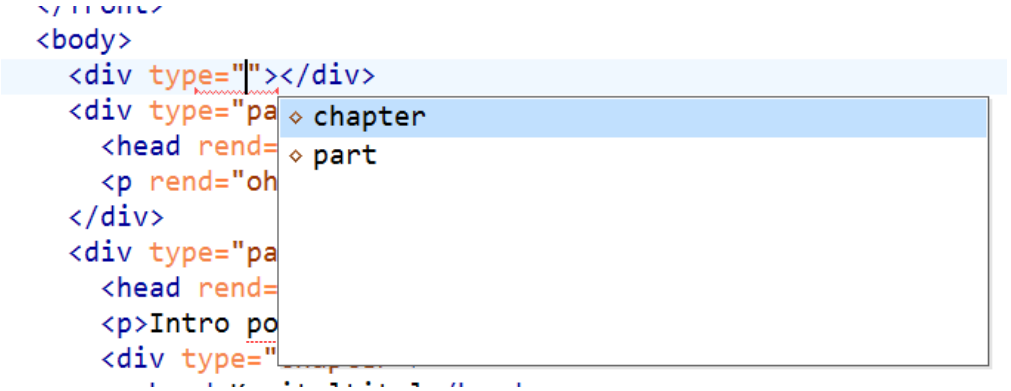


Figure 1. Content completion for Docbook-like `divs`

4.3. Using it in practice

4.3.1. Multiple Relax NG Associations or a single NVDL Association?

It has already been mentioned in Section 4.1 that the epischema can be bundled with the main schema in an NVDL schema. The NVDL of the whitelist example looks like

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"
  startMode="tei">

  <mode name="tei">
    <namespace ns="http://www.tei-c.org/ns/1.0">
```

```
      <validate schema="tei/tei_all.rng"/>
      <validate schema="whitelisted-body-div-children.rng"
        useMode="allow"/>
    </namespace>
  </mode>

  <mode name="allow">
    <anyNamespace>
      <allow/>
    </anyNamespace>
  </mode>

</rules>
```

When associated with the document via an `xml-model` processing instruction, an oXygen 19 beta version[5] will give this restricted content completion list:

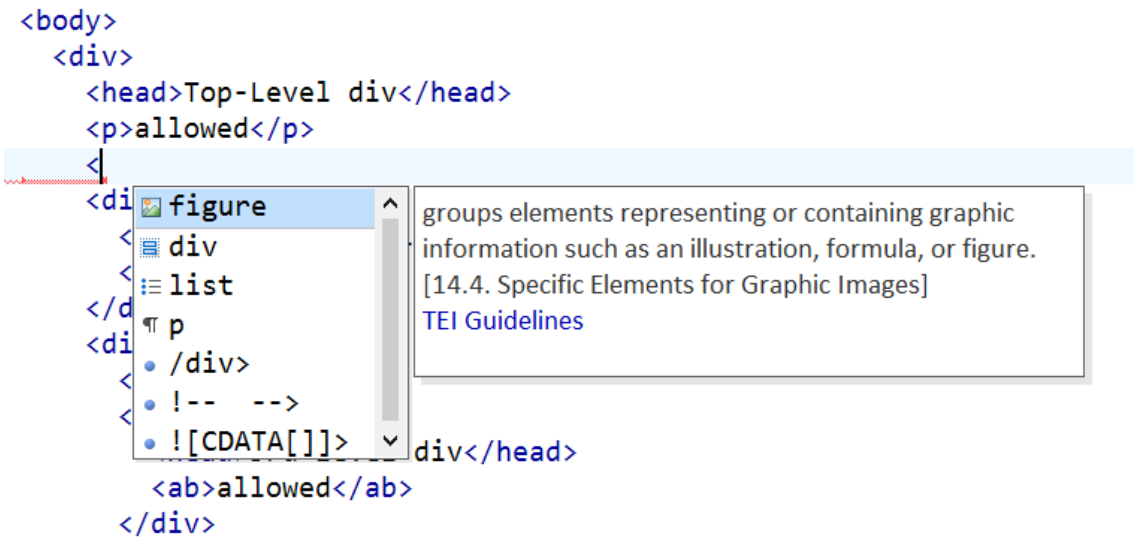


Figure 2. Content completion for whitelisted top-level `div` children

Wrapping the Docbook-like divs and the SVG/MathML-enhanced TEI in an NVDL looks like

```
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"
  startMode="tei">
  <mode name="tei">
    <namespace ns="http://www.tei-c.org/ns/1.0">
      <validate useMode="allow"
schema="http://www.le-tex.de/resource/schema/tei-cssa/docbook-like-
divs.rng"/>
      <validate useMode="extensions"
```

[5]Many thanks to George Bina for *personally* improving the schema-based content completion list calculation in oXygen XML Editor when multiple schemas apply in a given context.

```
schema="http://www.le-tex.de/resource/schema/tei-cssa/tei_allPlus-
cssa.rng"/>
    </namespace>
  </mode>

  <mode name="extensions">
    <namespace ns="http://www.w3.org/1996/css">
      <attach/>
    </namespace>
    <namespace ns="http://www.w3.org/1998/Math/MathML">
      <attach/>
    </namespace>
    <namespace ns="http://www.w3.org/2000/svg">
      <attach/>
    </namespace>
  </mode>

  <mode name="allow">
    <anyNamespace>
      <allow/>
    </anyNamespace>
  </mode>

</rules>
```

As an alternative to wrapping the schemas in an NVDL, they can be associated by adding a second `<?xml-model?>` processing instruction:

```
<?xml-model type="application/xml"
schematypens="http://relaxng.org/ns/structure/1.0"
href="https://www.tei-c.org/release/xml/tei/custom/schema/relaxng/▶
tei_all.rng"?>
<?xml-model type="application/xml"
schematypens="http://relaxng.org/ns/structure/1.0"
href="http://www.le-tex.de/resource/schema/tei-cssa/whitelisted-body-div-
children.rng"?>
```

4.3.2. Effects on oXygen Content Completion

It is currently recommended to use it as NVDL in oXygen because of content completion. When associating multiple Relax NG schemas by xml-model PIs, oXygen derives content completion suggestions only from the first one. There can be no optimal choice on which xml-model to place first: You'll end up with either sparse epischema suggestions or with only the standard TEI completion options. On the plus side, immediately after insertion, the document will be validated against both schemas and you will see in the validation messages which elements are actually valid at the given location.

When using NVDL in oXygen prior to version 19, the completion suggestions that each Schema generates will be merged. At any document location, you will have the full TEI completion choices plus the additional choices of the epischema.

This is ok for the div type values that haven't been present in the base schema, but it is not ideal. Until now, oXygen's NVDL implementation does not remove suggestions from the list that are prohibited by the other schema.

This is due to a heuristic that they apply. In terms of computing costs and perceived lag, they are currently not pursuing to pre-validate every scenario "would it still be valid according to all schemas if this element/attribute/attribute value was inserted at this position?" for each possible insertion item. This pre-validation would be necessary in order to filter suggestions according to Schematron constraints.

The author of this paper managed to convince oXygen XML Editor's George Bina that they can improve suggestion list generation in the case of multiple concurrent schemas. Instead of computing the union of all suggestions, they need to calculate the intersection. This has been implemented and will hopefully find its way into the forthcoming version 19 of oXygen XML Editor.

George Bina mentioned that there is another, explicit method of configuring the content completion that also works with Schematron constraints [26]. While this is useful, it requires additional configuration. The elegance of the epischema approach is that only a single (NVDL) schema association is necessary for both validation and content completion.

4.4. Complementing Other Base Schema Languages

It should be noted that although epischemas are probably best written in Relax NG, they work on top of any other schema mechanism, including DTD.

This demonstrates the limitations of a monistic `DOCTYPE` declaration as well as `xml-model`[6]'s and NVDL's utility for positively associating schemas with documents.

5. Outlook

It was already mentioned that epischema-based content completion will probably be supported in future versions of oXygen XML Editor.

It will also be interesting to explore how this concept can be applied to Web-based XML editors that often derive their context-dependent completion rules from translating schemas into program code or into configuration settings.

And it will also be interesting to apply this approach to other type- or class-heavy vocabularies with nested models, such as DITA `topics` or HTML `divs`/

[6] http://www.w3.org/TR/xml-model/

`sections`. Their users might also benefit from typed-section grammars for a more prescriptive authoring experience.

6. Conclusion

Although other approaches are feasible, additional structural constraints can be exerted elegantly by adding an epischema (or multiple epischemas, if you want to extend a basic schema in multiple dimensions). Epischemas offer aspect-based *orthogonality*, a separation of concerns, while they allow you to stick to the finest tool for defining XML document grammars (which is Relax NG, of course).

Bibliography

[1] http://docs.oasis-open.org/dita/v1.2/os/spec/archSpec/configuration-specialization-and-constraints.html#configuration-specialization-and-constraints. Accessed 2017-01-28.

[2] http://www.tei-c.org/Guidelines/Customization/. Accessed 2017-01-28.

[3] https://jats.nlm.nih.gov/. Accessed 2017-01-28.

[4] See for example https://listserv.brown.edu/archives/cgi-bin/wa?A1=ind1612&L=TEI-L#11 (accessed 2017-01-28) and subsequent threads.

[5] http://schematron.com/. Accessed 2017-01-28.

[6] https://oreillymedia.github.io/HTMLBook/. Accessed 2017-01-28.

[7] https://jats.nlm.nih.gov/extensions/bits/. Accessed 2017-01-28.

[8] Deutsche Forschungsgemeinschaft: Praxisregeln Digitalisierung (12/2016). http://www.dfg.de/formulare/12_151/12_151_de.pdf

[9] TEI Simple ODD: https://github.com/TEIC/TEI/blob/dev/P5/Exemplars/tei_simplePrint.odd#L448. Accessed 2017-01-18.

[10] https://subversion.le-tex.de/common/transpect-adaptions/docx-idml-tei-epub_github/. Accessed 2017-01-28.

[11] http://transpect.io/. Accessed 2017-01-28.

[12] https://listserv.brown.edu/archives/cgi-bin/wa?A1=ind1606&L=TEI-L#35. Accessed 2017-01-28.

[13] http://www.tei-c.org/release/doc/tei-p5-doc/en/html/DS.html#DSDIV. Accessed 2017-01-28.

[14] https://listserv.brown.edu/archives/cgi-bin/wa?A1=ind1701&L=TEI-L#10. Accessed 2017-01-28.

[15] http://httpd.apache.org/docs/2.0/rewrite/#preamble. Accessed 2017-01-28.

[16] Gary F. Simons, C. M. Sperberg-McQueen, and David G. Durand. 1999. *Rethinking TEI markup in the light of SGML architectures*. ACH-ALLC '99 International Humanities Computing Conference. June 9–13, 1999. Charlottesville, Virginia. http://www2.iath.virginia.edu/ach-allc.99/proceedings/simons.html. Accessed 2017-01-28.

[17] http://www.tei-c.org/release/doc/tei-p5-doc/en/html/USE.html#MD. Accessed 2017-01-28.

[18] Eric van der Vlist. 2003. *RELAX NG*. O'Reilly & Associates. http://books.xmlschemata.org/relaxng/relax-CHP-12-SECT-1.html#relax-CHP-12-SECT-1.1. Accessed 2017-01-28.

[19] http://broadcast.oreilly.com/2009/03/post-1.html. Accessed 2017-01-28.

[20] https://subversion.le-tex.de/common/schema/tei-cssa/whitelisted-body-div-children.rng. Accessed 2017-01-28.

[21] https://subversion.le-tex.de/common/schema/tei-cssa/tei_all_whitelisted-body-div-children.nvdl. Accessed 2017-01-28.

[22] https://subversion.le-tex.de/common/schema/tei-cssa/sample/tei-epischema_sample_whitelist.xml. Accessed 2017-01-28.

[23] http://archive.xmlprague.cz/2013/presentations/Conveying_Layout_Information_with_CSSa/CSSa_xmlprague_gimsieke.html .

[24] https://subversion.le-tex.de/common/schema/tei-cssa/docbook-like-divs.rng. Accessed 2017-01-28.

[25] https://subversion.le-tex.de/common/schema/tei-cssa/sample/tei-epischema_sample.xml. Accessed 2017-01-28.

[26] https://www.oxygenxml.com/doc/versions/18.1/ug-editor/topics/configuring-content-completion-proposals.html. Accessed 2017-01-28.

CSS for Print via XSL-FO

George Bina
Syncro Soft
<george@oxygenxml.com>

Dan Caprioara
Syncro Soft
<dan@sync.ro>

Abstract

The problem with XSL-FO is that it is complex to create and modify, and people prefer to customize PDF using CSS rather than creating/modifying an XSLT stylesheet that generates XSL-FO. Thus, CSS for print has received more traction lately. There are many initiatives for various XML vocabularies to provide support for CSS for print (for example, there are two open-source projects to generate PDF from DITA using CSS).

On the other hand, there are a number of XSL-FO engines available (including the open-source Apache FOP engine) that provide reasonable support for XSL-FO to produce PDF.

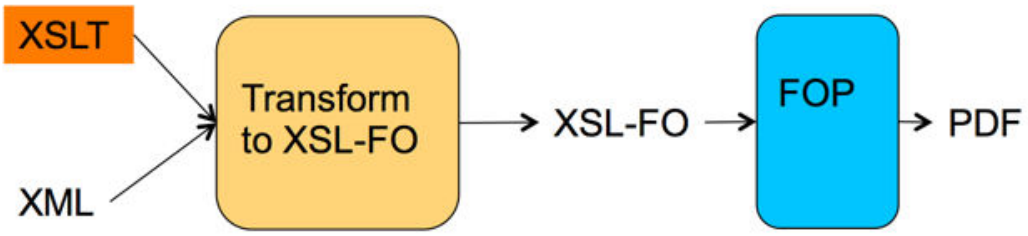
In order to leverage the existing FO processors for CSS-based PDF, we can support CSS for print by implementing a conversion from XML+CSS to XSL-FO and then apply an FO processor to get the actual PDF output.

We will show the anatomy of such an engine that implements CSS for print using XSL-FO as an intermediary format. We will focus on the advantages of such an approach as well as discussing challenges we encountered during implementation. Some advanced CSS level 3 and level 4 functions are essential for more advanced layouts or rendering of information. We also propose a few CSS extensions that can be very useful, such as a function to provide XPath evaluation support.

Keywords: Print, XML, CSS, PDF, XSL-FO

1. Motivation

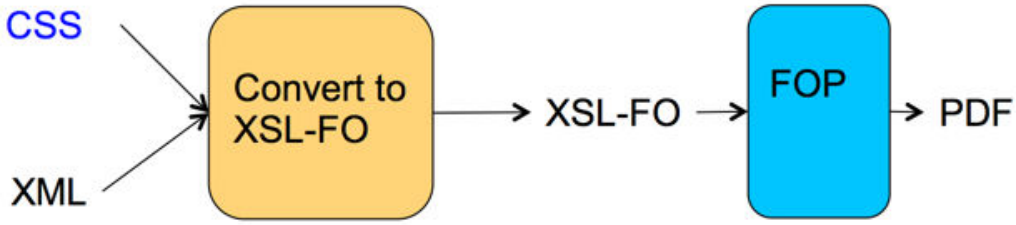
XSL-FO is a very good option to generate PDF documents and it has already been used for a number of years in production systems, so it is a mature and powerful technology. The only issue with it relates to customizations. Usually the process involves converting XML content to XSL-FO via XSLT stylesheets and the customization means applying changes to the XSLT code that generates the XSL-FO.



Thus, someone needs to possess very good knowledge of both XSLT and XSL-FO as a minimum, since in most cases one needs to also understand how the current XSLT stylesheets are designed and how to integrate new XSLT scripts into the existing processing workflow. This makes any customization of XSL-FO based systems a rather difficult task and many people lack part of the required knowledge, so only a few are actually able to make such customizations.

As an alternative approach to XSL-FO, in the last few years we have seen a lot of advancements to CSS-based processors that basically use CSS and specific page extensions to describe how a document should be formatted for print and they generate PDF based on that information. These have the very important advantage of being easy to customize, taking advantage of the simple CSS syntax and the fact that many potential users already know CSS. Also, a very important fact is that the existing CSS design is not very important, thus you can just import the existing CSS and add new rules in the new CSS file.

So, if we come back to XSL-FO, it is a good and mature option and there is at least one free and open-source product that converts XSL-FO to PDF, the Apache FOP processor. We can try to solve the customization problem by providing a simple customization layer to control the XSL-FO generation from there. Now, if we choose this customization layer to be CSS, we basically obtain a CSS to PDF processor that internally goes through XSL-FO as an intermediary stage and reuses the available formatting objects processors.



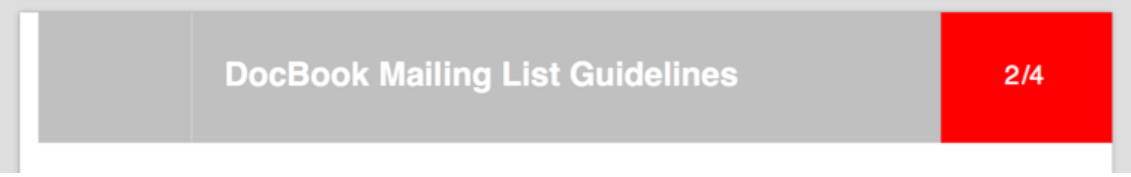
2. CSS customizations

To understand why CSS is a good choice as the XSL-FO customization layer, we can look at a few examples. To define page properties, we can use the `@page` CSS at-rule and define page size, margins, background, and which static content appears in various areas of the page (such as page number, title, and so on).

Example 1. CSS for defining default page properties

```
@page {
    size:a4;
    margin: 1.3in;
    @top-left {
        background-color:silver;
        font-size: 1.5em;
        font-weight:bold;
        font-family: sans-serif;
        content: string(article-title);
        padding-left:1em;
        color:white;
    }
    @top-right-corner{
        background-color:red;
        color:white;
        font-size:large;
        content: counter(page) " / " counter(pages);
        text-align: center;
        vertical-align: middle;
    }
}
```

Such a CSS will generate the document title and the current page number from the total pages in the page header:



In order to define how a specific element should appear, or to customize its rendering, we do not need a lot of knowledge about the existing CSS. If we want titles to be centered, then all we need is a rule like:

```
title {
    text-align:center;
}
```

and titles will be centered:

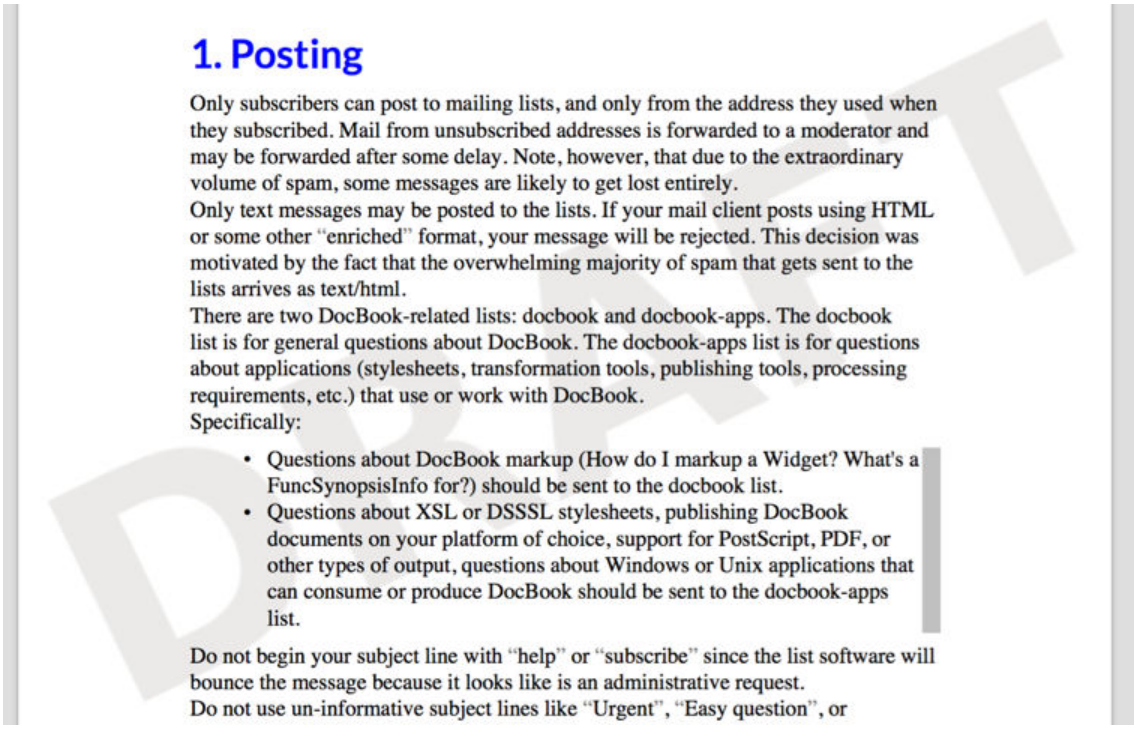
Replies Go to the Author

The DocBook lists don't set the Reply-to: header on messages (see "Reply-To" Munging Considered Harmful) so replies go to the author by default. If you want to reply to the list, make sure you include the list mailing address in your reply.

Marking a document as draft, with a watermark image can be done with a few lines:

Example 2. Marking as draft

```
@page {
...
    background-image:url('draft.png');
    background-repeat:no-repeat;
    background-position:center;
...
}
```



1. Posting

Only subscribers can post to mailing lists, and only from the address they used when they subscribed. Mail from unsubscribed addresses is forwarded to a moderator and may be forwarded after some delay. Note, however, that due to the extraordinary volume of spam, some messages are likely to get lost entirely.
Only text messages may be posted to the lists. If your mail client posts using HTML or some other "enriched" format, your message will be rejected. This decision was motivated by the fact that the overwhelming majority of spam that gets sent to the lists arrives as text/html.
There are two DocBook-related lists: docbook and docbook-apps. The docbook list is for general questions about DocBook. The docbook-apps list is for questions about applications (stylesheets, transformation tools, publishing tools, processing requirements, etc.) that use or work with DocBook.
Specifically:

- Questions about DocBook markup (How do I markup a Widget? What's a FuncSynopsisInfo for?) should be sent to the docbook list.
- Questions about XSL or DSSSL stylesheets, publishing DocBook documents on your platform of choice, support for PostScript, PDF, or other types of output, questions about Windows or Unix applications that can consume or produce DocBook should be sent to the docbook-apps list.

Do not begin your subject line with "help" or "subscribe" since the list software will bounce the message because it looks like is an administrative request.
Do not use un-informative subject lines like "Urgent", "Easy question", or

Rendering the content on two columns is as easy as:

Example 3. Split content on two columns

```
@page {
...
    column-count : 2;
    column-gap : 2em;
```

```
...
}
```

4. Use Short Quotes of Previous Messages in Replies

Please do not quote entire messages just to add a few lines at the beginning or end. Instead, quote the parts to which you are directly replying or quote enough to establish the context. Everybody on the list has already received the message that you are quoting, and anyone searching the archive will find your message and the previous message listed under the same thread.
Subscribers to the mailing list will just ignore most of the quoted messages and move to the next post, but subscribers to the mailing list Digest will mostly have to page past the quoted messages to reach the next material in which they are interested in reading.

6. Subscribing and Unsubscribing

Subscriptions to the DocBook lists and their digest forms are managed by OASIS: http://www.oasis-open.org/mlmanage/index.php .

7. Archive

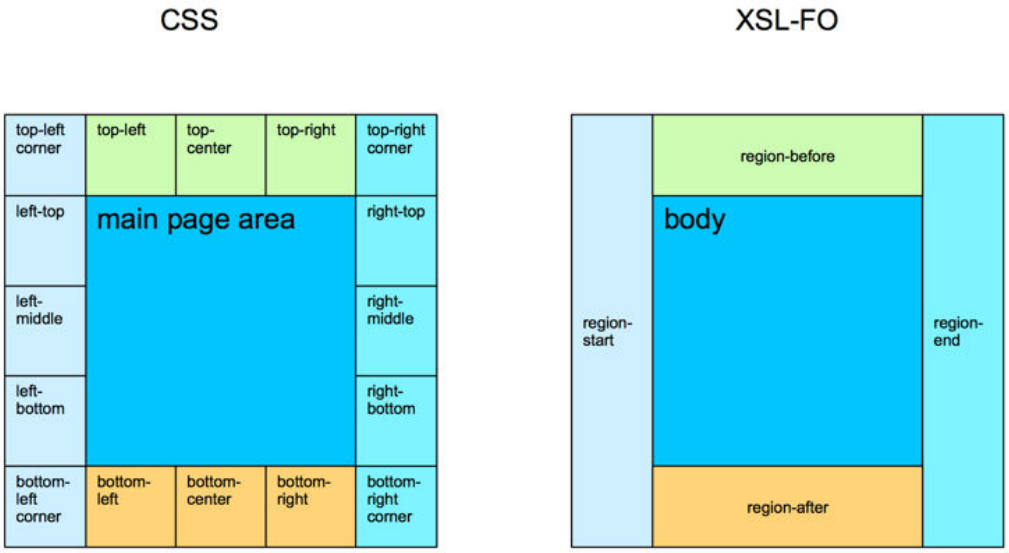
The DocBook mailing list messages are archived at http://lists.oasis-open.org/archives/docbook/.
The DocBook Applications mailing list messages are archived at http://lists.oasis-open.org/archives/docbook-apps/.

3. CSS and XSL-FO

While CSS and XSL-FO share many common properties, there are also some differences between them that need to be taken into account. We can explore a few cases where we have an impedance mismatch between CSS and XSL-FO.

3.1. Different page regions

For example, the page regions are different. In CSS we have 17 regions, but in XSL-FO only 5 regions, as can be seen in the following figure:



This means that in order to convert from CSS to XSL-FO, we need to define additional structure inside XSL-FO to accommodate all of these regions (for example, we can add a table in the `region-before` and `region-after` areas with one row and up to three columns, or a set of block container elements in the `region-start` and `region-end` areas, controlling the width and height to store the corresponding content specified in CSS).

3.2. Nested elements on separate page types

Another difference is that an XML document can contain nested structures that the CSS styles can place in different page types and nested page types are not supported in XSL-FO. Thus, another important thing to consider is splitting and slicing the document to avoid nesting content that goes into different page types.

Example 4. Sections in section specific page types

```
section {
    /* Each section is in its own page. */
    page:sectionPage;
}
```

A document like:

```
<article>
…
    <section>
  …
    </section>
    <section>
```

```
  …
    </section>
…
</article>
```

will need to have the article content in the default page type and each section content in section-specific page types and while sections are within an article element in the original XML document, we need to split/slice the document to avoid nesting of content that belongs to different page types, like:

```
<article>
…
</article>
<article>
    <section>
    …
    </section>
</article>
<article>
    <section>
    …
    </section>
</article>
<article>
…
</article>
```

3.3. Start and end indent

If the CSS marks an element as `block` then it can be translated to an XSL-FO block, but if the CSS specifies a `width` for a block element, then we need to translate it to an XSL-FO block container to be able to also specify the `width` property in XSL-FO. This does not seem to generate any issues at first glance, but in XSL-FO, text indent in block containers is relative to the nearest ancestor block container, ignoring block ancestors, while in CSS, this will be relative to the parent block. Thus, we get this difference that we need to take into account in order to make sure the generated XSL-FO behaves the same as the original CSS.

3.4. Font management

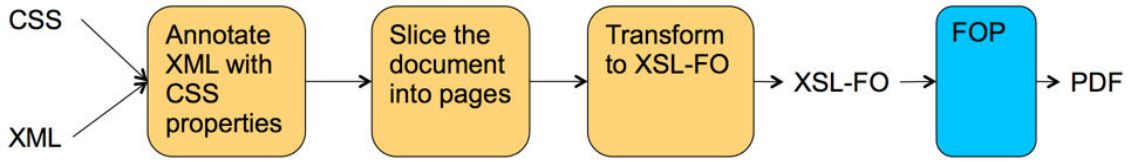
Sometimes, parts of the CSS cannot be fully translated to XSL-FO, but they need to go into some other type of resources. For example, the CSS fonts may also result in an FO processor-specific configuration file that loads or defines those fonts for the FO processor.

3.5. XSL-FO versus FOP

Another thing to take into account, from a practical point of view, is the fact that a specific FO processor may not fully support the XSL-FO standard, so even if some CSS functionality translates to XSL-FO, we may not be able to use that in practice with the specific FOP we use. As an example, the `relative-align="baseline"` is not supported by Apache FOP, so we need to either improve the FOP to add the missing support or we need to find some other way to obtain the same behavior by generating a different XSL-FO code.

4. Implementation

We need a system that takes XML as input and generates PDF taking into account the styles specified in a CSS file. This conversion can be split in a number of processing stages:



- annotate XML with CSS information
- slice the annotated document into pages
- transform to XSL-FO
- apply FOP processing

4.1. Annotate XML with CSS information

The first processing stage, after we parse the document resolving entities, default values, XInclude includes, etc. is to annotate the document with CSS information. That is, we need to find what styles apply to each node and make that information available within the XML document so that it can be processed further by the following processing stages.

While the annotations that represent the CSS information are mostly attributes, there are a few cases where we need to use XML elements (for example, the static content generated from CSS will appear as XML elements in the annotated document).

All annotations are stored in a particular namepsace (we used `http://www.w3.org/1998/CSS` mapped to the `css` prefix).

Example 5. Annotate with CSS properties

If we have an XML document, for example an article:

```
<article>
...
</article>
```

and we specify it as a block in the CSS file:

```
article {
    display:block;
}
```

then the annotated document will contain this information using a `css:display` attribute:

```
<article css:display="block">
...
</article>
```

For static content, we generate annotation elements, because we can have multiple static blocks of content and that can also be composed from multiple sources, static text, attribute values, functions results, etc.

Example 6. Annotation elements for static content

Let's say we want to precede section titles with a “Title:” prefix. Then, if we have an XML document:

```
<section>
  <title>Processor</title>
</section>
```

and a CSS that generates that prefix:

```
section, title {
    display:block;
}
title:before {
    color:green;
    content: "Title:";
}
```

the annotated XML will contain a `css:before` element with the generated static content inside:

```
<section xmlns:css="http://www.w3.org/1998/CSS" css:display="block"
    css:font-family="sansserif" css:font-size="12pt">
  <title css:display="block">
    <css:before css:display="inline" css:color="green">
    <css:text>Title:</css:text>
     </css:before>
     Processor
```

```
  </title>
</section>
```

In order to further process page information, we also add that as annotation elements in the `css` namespace.

Example 7. Page information:

```
@page {
    size: A4;
    margin: 0.5in;
}
```

will result in an XML fragment that expands the CSS properties:

```
<css:pages>
    <css:page>
        <css:property name="width" value="8.27in"/>
        <css:property name="height" value="11.69in"/>
        <css:property name="margin-top" value="0.5in"/>
        <css:property name="margin-right" value="0.5in"/>
        <css:property name="margin-left" value="0.5in"/>
        <css:property name="margin-bottom" value="0.5in"/>
    </css:page>
</css:pages>
```

We also need to take into account application defaults, such as font size, page size, and so on, as well as implicit CSS rules and inheritance. Also, the CSS may be more generic and specify properties that do not apply to print media, so we need to collect only the general ones and the ones marked with print media and ignore others that are specified for other media, (screen for instance).

4.2. Slice into page sequences

At this stage, we have each element that should be placed on a specific page type annotated with a `css:page` attribute.

```
<article xmlns="http://docbook.org/ns/docbook" version="5.0"
    xmlns:css="http://www.w3.org/1998/CSS"
    css:display="block" css:font-family="sansserif"
    css:font-size="12pt">
  <title css:display="block" css:font-size="20pt">
    Sample
  </title>
  <section css:display="block" css:page="section_page">
    <title css:display="block" css:font-size="20pt">
    Introduction
    </title>
```

```
  </section>
  <section css:display="block" css:page="section_page">
    <title css:display="block" css:font-size="20pt">
      Content
    </title>
  </section>
  <acknowledgements css:display="block">
    <para css:display="block">Thanks!</para>
  </acknowledgements>
</article>
```

Notice here that for some elements there is no page information. This means they do not change the page type and if no ancestor contains page information, the default page type will be used. This stage tries to flatten the page structure, making the document ready for the following transformation to XSL-FO. That means we need to slice the document into separate page sequences, each containing content that should be rendered in pages of the same type.

We generate a wrapper element to contain page sequences, and a `page-sequence` element to mark each page sequence. Thus, the above example will result in:

```
<css:root xmlns:css="http://www.w3.org/1998/CSS"
    xmlns="http://docbook.org/ns/docbook">
  <css:pages>
    <css:page name="section_page">
      <css:property name="margin-left" value="2cm"/>
    </css:page>
  </css:pages>
  <css:page-sequence page="css2fo-default">
    <article version="5.0" css:display="block"
        css:font-family="sansserif" css:font-size="12pt">
      <title css:display="block" css:font-size="20pt">Sample</title>
    </article>
  </css:page-sequence>
  <css:page-sequence page="section_page">
    <article version="5.0" css:display="block"
        css:font-family="sansserif" css:font-size="12pt">
      <section css:display="block">
        <title css:display="block" css:font-size="20pt">Introduction</▶
title>
      </section>
    </article>
  </css:page-sequence>
  <css:page-sequence page="section_page">
    <article version="5.0" css:display="block"
        css:font-family="sansserif" css:font-size="12pt">
```

```
      <section css:display="block">
        <title css:display="block" css:font-size="20pt">Content</title>
      </section>
    </article>
  </css:page-sequence>
  <css:page-sequence page="css2fo-default">
    <article version="5.0" css:display="block"
        css:font-family="sansserif" css:font-size="12pt">
      <acknowledgements css:display="block">
        <para css:display="block">Thanks!</para>
      </acknowledgements>
    </article>
  </css:page-sequence>
</css:root>
```

The `css:root` wrapper element will also include the page meta-information.

4.3. Transform to XSL-FO

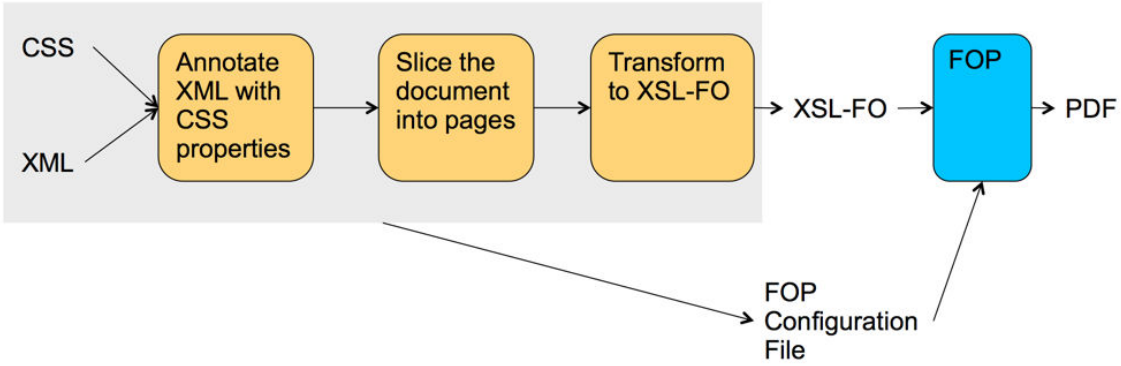
Now we have an annotated document that is ready for XSL-FO conversion. At this stage, we apply an XSLT script that will transform the annotated document into XSL-FO. This transformation does not depend on the element names from the original XML document. We basically match on the CSS annotations and drive the conversion to XSL-FO based on that.

Example 8. Processing inline elements

```
<xsl:template match="*[@css:display = 'inline'">
    <fo:inline>
        <xsl:apply-templates select="@* except @css:display | node()"/>
    </fo:inline>
</xsl:template>
```

4.4. Apply FOP to obtain PDF

At this stage, we have the XSL-FO file and we apply the Apache FOP processor to obtain the print format, the PDF file. There is one additional thing that changes the overview diagram a bit and that is the fact that some of the CSS properties translate to entries in the FOP configuration file, such as the font information, so the FOP stage also receives that configuration file as an input.



5. Related work

CSSToXSLFO[1] is an open-source project that was started to implement this idea of converting to XSL-FO from XML and CSS. Unfortunately, there has been no activity on this project for the past few years.

Then we have a number of commercial *CSS to print* engines, but they do not advertise whether or not they go through XSL-FO as an intermediary format:

- PrinceXML[2]
- AntennaHouse[3]
- PDF Reactor[4]
- Vivlyostyle[5]

6. Final thoughts

The processing pipeline can include some additional steps, such as a pre-processing step for the XML document before it is matched against the CSS file, or a post processing of the generated XSL-FO before it goes through the FOP stage. These extension points should allow for more advanced functionality or provide workarounds for solving specific problems.

With the print extensions, CSS can be the easy customization layer for generating XSL-FO, its main advantages being that it is a widely used and rather easy technology and in order to customize it, you do not necessarily need to know how the existing CSS is developed.

CSS may not solve all the customization requirements in general, but for technical documentation it should be enough. Although there are a number of impe-

[1] http://www.re.be/css2xslfo/
[2] https://www.princexml.com
[3] http://www.antennahouse.com/
[4] http://www.pdfreactor.com/
[5] http://vivliostyle.com/

dance mismatches between CSS and XSL-FO, there are ways to solve them and we solve them once, taking advantage of this work every time we transform XML to print using CSS.

Jiří Kosek (ed.)

XML Prague 2017
Conference Proceedings

Published by
Ing. Jiří Kosek
Filipka 326
463 23 Oldřichov v Hájích
Czech Republic

PDF was produced from DocBook XML sources
using XSL-FO and AH Formatter.

1st edition

Prague 2017

ISBN 978-80-906259-2-1 (pdf)
ISBN 978-80-906259-3-8 (ePub)