XML Prague 2020

Conference Proceedings

University of Economics, Prague
Prague, Czech Republic

February 13–15, 2020

XML Prague 2020 – Conference Proceedings

ISBN 978-80-906259-8-3 (pdf)
ISBN 978-80-906259-9-0 (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 13th, 14th and 15th, 2020

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, *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, *The New York Times*
Petr Cimprich, *Wunderman*
Jim Fuller, *MarkLogic*
Michael Kay, *Saxonica*
Jirka Kosek (chair), *University of Economics, Prague*
Ari Nordström, *Creative Words*
Uche Ogbuji, *Zepheira LLC*
Adam Retter, *Evolved Binary*
Andrew Sales, *Bloomsbury Publishing plc*
Felix Sasaki, *Cornelsen GmbH*
John Snelson, *MarkLogic*
Jeni Tennison, *Open Data Institute*
Eric van der Vlist, *Dyomedea*
Priscilla Walmsley, *Datypic*
Norman Tovey-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 (https://www.oxygenxml.com)
Antenna House (https://www.antennahouse.com/)
le-tex publishing services (https://www.le-tex.de/en/)
Saxonica (https://www.saxonica.com/)
print-css.rock (https://print-css.rock/)
Czech Association for Digital Humanities (https://www.czadh.cz)
speedata (https://www.speedata.de/)
schematronist.org (https://schematronist.org/)
Mercator IT Solutions Ltd (http://www.mercatorit.com)

Preface

This publication contains papers presented during the XML Prague 2020 conference.

In its 15th 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 13–15 February 2020 at the campus of University of Economics in Prague. XML Prague 2020 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 https://xmlprague.cz)—allowing XML fans, from around the world, to participate on-line.

The Thursday runs in an un-conference 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.

This year we put special focus on CSS and publishing. On the un-conference day there will be introductory tutorial about producing print output using CSS followed by the workshop where future of CSS Print should be discussed. Friday opening keynote by Rachel Andrew *Refactoring (the way we talk about) CSS* will hopefully give you a new perspective about how to perceive CSS.

We hope that you enjoy XML Prague 2020!

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

A note on Editor performance

A story on how the performance of Fonto came to be what it is, and how we will further improve it

Stef Busking
FontoXML
<stef.busking@fontoxml.com>

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

Abstract

This paper will discuss a number of key performance optimizations made during the development of Fonto, a web-based WYSIWYM XML editor. It describes how the configuration layer of Fonto works and what we did to make it faster. It will also describe how the indexing layer of Fonto works and how we improve it in the future.

1. Introduction

1.1. How does Fonto work?

Fonto is a browser-based WYSIWYM[1] editor for XML documents. It can be configured for any schema, including many DITA specializations, JATS, the TEI, docbook and more. Fonto configuration consists of three parts:

1. How do elements look and feel (the *schema experience*)
2. How can they be mutated (the *operations*)
3. The encompassing *user interface* of Fonto

The schema experience is specified as a set of rules that assign specific properties to all nodes matching a corresponding selector. These selectors are expressed in XPath.

Operations also make use of XPath in order to query the documents. Effects are defined either as JavaScript code, or using XQuery Update Facility 3.0.

The user interface of Fonto has several areas (e.g., the toolbar, sidebar and custom dialog boxes) in which custom UI can be composed from React components. These can observe XPath expressions to access the current state of the documents

[1] What You See Is What You Mean

and be updated when it changes. The documents themselves are rendered recursively by querying the schema experience for each node and generating HTML appropriate for the resulting configuration.

1.2. What is performance?

When a single key is pressed, Fonto needs to update the XML and then update all related UI. This includes updating the HTML representation of the documents, recomputing the state of all toolbar buttons based on the applicability of their operation in the new state, and updating any other UI as necessary.

Typically, such updates involve looking up the values of various configured properties for a number of nodes (by re-evaluating the associated XPath selectors against those nodes) and/or executing other types of XPath / XQuery queries. In order to keep the editor responsive, these updates need to be implemented in a way that scales well with respect to both the complexity of the configuration as well as the sizes of the documents being edited. In order to keep Fonto easy to configure, we should not place too many requirements on the shape of this configuration. This means Fonto has to deal with a wide range of possibilities regarding the number of selectors etc.

When we started Fonto, we considered documents of around 100KB to be 'pretty big', and these could be pretty slow to work with. After heavy optimization, we now have workable editors that load documents of multiple megabytes[2], using (automatically updating) cross references, (automatic) numbering of sections and more. This paper details a few of the most significant optimizations we have applied in order to get to that point.

2. Accessing schema experience configuration

As described in the introduction, Fonto uses XPath selectors to apply a set of properties to nodes. We call the combination of a selector and a value a declaration.

Example of the look and feel configuration of the 'p' element:

```
configureAsBlock(sxModule, 'self::p');
```

This configuration does the following internally:

[2]Using just-in-time loading to only load a small subset, this even scales to working in collections totaling in the hundreds of megabytes, but that could be considered cheating.

Table 1. Summary of properties set for a paragraph

Property	Value
Automergable	false
Closed	false
Detached	false
Ignored for navigation	false
Removable if empty	true
Splittable	true
Select before delete	false
Default Text Container	none
Layout type	block
Inner layout type	inline
… (a total of 23 properties, plus optionally up to 35 more that are not set automatically)	...

There are about 23 properties being configured for a single paragraph, each specifying whether the paragraph may be split, how it should interact with the arrow keys, how it behaves when pressing enter in and around it, etcetera.

2.1. Orthogonal configuration

A number of these properties can be set individually, such as the background color or the text alignment of an element. This allows for a drastic reduction in the amount of selectors. Previously, when configuring some paragraphs to have a different background color compared to the 'generic' paragraph, all of the 'the same' properties also needed to be configured. By adding a way to configure single properties, reductions of more than three quarters of the configuration were seen.

Table 2. Orthogonal configuration

Without using Orthogonal Configuration	With Orthogonal configuration

`configureAsBlock(`
`  sxModule,`
`  'self::p',`
`  'paragraph'`
`);`
 	
`configureAsBlock(`
`  sxModule,`
`  'self::p[@align="right"]',`
`  'paragraph with right alignment',`
`  {align: 'right'}`
`);`
	`configureAsBlock(`
	`  sxModule,`
	`  'self::p',`
	`  'paragraph'`
	`);`
	
	`configureProperties(`
	`  sxModule,`
	`  'self::p[@align="right"]',`
	`  {`
	`  markupLabel: 'paragraph with right alignment',`
	`  align: 'right'`
	`  }`
	`);`

2 x 23 properties, plus one, for the alignment = 47	23 properties, plus one, for the alignment, makes 24

For a property like how an element behaves when computing the plain text value from it, the registry may look like this, for the p element. Note that multiple of these selectors are automatically generated.

Table 3. Properties defined for a paragraph

Selector	Plain text behavior	Priority

`self::p and`
`parent::*[(`
`  self::list-item`
`  ) and`
`parent::* [`
`  self::list[`
`  @list-type="simple"]]]`
	interruption	2

`self::p and parent::*[(self::list-item) and parent::*[ self::list[ @list-type="roman-upper" and @continued-from ] ]]`	interruption	2
<18 rows omitted for clarity>		
`self::p[parent::def]`	interruption	0
`self::p`	interruption	0
`self::*[parent::graphic]`	removed	0

2.2. Selector buckets

As shown earlier, selectors are used extensively in the configuration layer. For some selectors, it is quite obvious to see that a given node will never match a given selector. For example, the selector `self::p` may never match the `<div />` element.

We leverage this knowledge by indexing the selectors that are used in configuration by a hash of the kind of nodes they may 'match' their 'buckets'. We currently use node type buckets - derived from the nodeType values defined in the DOM spec[8] - and node name buckets derived from the qualified names of elements.

Table 4. Buckets

Selector	Bucket
`self::p`	name-p
`self::element()`	type-1
`@class`	type-1 (only elements may have attributes)
`self::p or self::div`	type-1
`self::comment()`	type-8
`self::*`	No bucket: both attributes and elements may match to this selector

Note that some of these selectors could also be expressed as a list of more specific buckets. For example, `self::*` could be stored under both the bucket for `type-1` as well as the one for `type-2` For simplicity, and to keep lookups by bucket as efficient as possible, we have currently limited our implementation to a single bucket per selector. We may revisit this decision in the future.

We then group the selectors that configure a certain property by their bucket. By computing the same hash(es) that may apply for a node, we drastically reduce the amount of selectors that need to be tested against any given node.

2.3. Selector priority / optimal order of execution

2.3.1. Conceptual Approach

An application may have the following configuration for the 'italic font' property:

Table 5. Italic font per selector

Selector	Value
`self::cursive`	true
`self::quote`	true
`self::plain-text`	false
<default>	none

The ordering of selectors is defined using a *specificity* system inspired by CSS: We group and count the amount of 'tests' in an selector: a selector with two attribute tests is more important than one with a single attribute test. Additionally, we allow applications to define explicit *priorities*. Specificity is used only if priorities are omitted or to break ties when priorities are equal.

The selectors defined by this piece of configuration will be evaluated in order and the value of the first match will be returned. In this example, a <p /> element will have no configuration for the 'slant' property, while the <quote /> will set it to 'true' and the <plain-text /> will set it to 'false'.

2.3.2. Optimization

The ordering of declarations does not mean all of these selectors have to be executed in that specific order. In the table defining the properties set for a paragraph, all of the high-priority selectors have a very low probability of matching. The much simpler `self::p` selector is more likely to match. To generalize this problem, we use a Bayesian predictor for the likeliness of whether a selector will match a given node.

The hypothesis (H) is that this selector matches. Evidence (E) is the hash assigned to the node. This is configurable, but usually the name of the element we input. We want to compute the probability of H given E: the selector matches for this hash. Bayes theorem gives us that $P\left(H \middle| E\right) = \frac{P(E|H) \quad P(H)}{P(E)}$ where $P(E|H)$ is the percentage of matches of this selector that match this hash. Basically, this is the amount of times the selector matched a similar element, continuously approximated based on previous results. $P(H)$ is the percentage of matches of this selector overall, and $P(E)$ is the percentage of results of any selector for a node with this hash. Because we will compare these scores for the same hash, the $P(E)$ part is constant and can be omitted.

We use the statistical probability of the selectors we will evaluate to determine an optimal order of execution of selectors. If we evaluate all selectors in order of decreasing likeliness, we only need to check selectors with higher priority *but a different value* in case of a match. In pseudocode, this becomes:

```
Let declarations be all declarations that may match the input, based on
buckets.
Sort declarations based on their priority, their specificity and lastly
on order of declaration.
Let skippedDeclarations be an empty list.
Let declarationsInOrderOfLikeliness be declarations, sorted using the
Bayesian predictor from most likely to least.
For likelyDeclaration of declarationsInOrderOfLikeliness do:
  If (likelyDeclaration.selector does not match input) continue;
  // We have a likely match, see whether it was the 'good' one
  For declaration of declarations do:
    If (declaration.selector is equal to likelyDeclaration.selector)
      // The likely declaration is the most matching one
      Return likelyDeclaration.value;
    If (declaration.value is equal to likelyDeclaration.value)
      // No need to evaluate this selector now,
      // it would result in the same value
      Add the declaration to skippedDeclarations, continue;
    // This higher-priority declaration would result in a different value
    If (declaration.selector does not match the input) continue;
    // This declaration applies, unless one of the skipped declarations
(with higher priority) matches as well
    For skippedDeclaration of skippedDeclarations do:
      If (skippedDeclaration.selector matches input)
        Return likelyDeclaration.value
    // We have no declaration that is deemed more important
    Return declaration.value
```

Fonto ends up querying a large number of declarations for all nodes in the loaded documents as a result of rendering and other initial processing. This means that the initial set-up will make sure that the Bayesian predictor is sufficiently trained by the time the user starts editing.

2.3.3. Performance impact

Worst case: This algorithm has the same worst-case performance as the implementation without it. The worst case will be triggered when the most likely match is also the least important one, and all preceding declarations point to another value. In this case, the algorithm will be forced to evaluate every preceding selector.

Best case: The most likely selector is preceded by a large amount of more complex selectors, which point to the same value. The algorithm will only evaluate a single selector: the most likely one. Because these selectors are prefiltered by their bucket, this is the more likely case: it is more common to configure a number of paragraphs to have the same declared value in for instance enter behaviour than all having different values.

2.3.3.1. Measurement

We conducted a performance test of the initial render of a JATS document of 721KB, containing 18826 nodes in the configuration that was highlighted in the table describing the properties set for a paragraph. These performance tests measured how long it took to render all of the content to html elements using Chrome 81 in Fonto 7.9.0. Tests were repeated 4 times.

Table 6. Performance of the Bayesian predictor

	Amount of XPaths evaluated
Without the Bayesian predictor	121575
With the Bayesian predictor	109964

With the optimization, we see a 9.5% reduction in the amount of XPaths that are being evaluated. The total load time is reduced by three seconds. This is a significant improvement over the old situation.

Furthermore, we measured how many times certain XPath expressions were executed. The following expressions stood out:

Table 7. XPaths with a fewer executions with the Bayesian predictor:

Selector	Execution count without predictor	Total time spent executing this expression	Execution count with predictor	Total time spent executing this expression
`self::*[ parent::*[ self::term-sec[ not(ancestor::abstract or ancestor::boxed-text)]]] and not ( self::node()[ not(self::sec or self::term-sec)] )`	1797	93 ms	900	42 ms
`self::label`	79	2ms	1662	47 ms
`self::label[ parent::abstract]`	79	3 ms	1	(Too low to measure)
`self::label[parent::fn]`	1733	60 ms	1091	42 ms
`self::named-content[ @vocab="unit-category"]`	2173	160 ms	1174	96 ms

`self::named-content[ @vocab="specification"]`	325	22 ms	538	56 ms

From this table, the label selectors stand out the most: the `self::label` selector grew both in execution count and in the total spent. This effect is explained by the next selector: `self::label[parent::abstract]`. This selector is part of a set of twelve similar selectors that went for 79 executions to a single one. The Bayesian predictor learned that the `self::label` select is more likely to match than the `self::label[parent:abstract]` and prevents executing it.

2.3.3.2. Comparison to another approach

In order to verify the results of the Bayesian predictor, we compared it to another, similar approach. Instead of using the predictor as the main sorting function, use the 'complexity of a selector. In other words, consider 'simpler' expressions to be more likely to match than 'complex' selectors. In order to approximate the 'complexity' of a selector, use the specificity algorithm as described in an earlier section.

This gave us the following results for the selectors mentioned in the previous chapter:

Total amount of XPaths executed: 111864.

Table 8. Performance metrics of using selector specificity as likeliness

Selector	Execution count without predictor	Total time spent executing this expression
`self::*[ parent::*[ self::term-sec[ not(ancestor::abstract or ancestor::boxed-text)]]] and not ( self::node()[ not(self::sec or self::term-sec)] )`	899	43 ms
`self::label`	1684	46 ms
`self::label[ parent:abstract]`	0	-
`self::named-content[ @vocab="unit-category"]`	2165	175 ms

The table gives interesting results: the `self::label` selector is executed many times, but the `self::label[parent:abstract]` selector is never executed at all.

However, the moderately complex `self::named-content[@vocab="unit-category"]` selector is evaluated way more often than when using the Bayesian predictor.

When going through the configuration of this editor, this can be explained. The 'normal' `<named-content />` element is expected to never occur in the editor in question. It is configured to never be rendered. However, the more special `<named-content />` elements that have additional attributes set are expected to occur, and are given a number of additional visualization properties, such as widgets, additional options for a context menu etcetera.

In essence, the `self::named-content` occurs few times in the total configuration, while the specific versions occurs many times. However, some specific versions of this element occur more than others; the Bayesian predictor takes advantage of this while this approach can not hold it into account.

2.4. Deduplication of duplicate property values

This best case is further leveraged by deduplicating duplicate values. In some cases, the configuration API allows one to input instances of functions. We rewrote these APIs to allow for better memoization: all function factories attempt to return the same function when called multiple times with the same arguments.

2.5. Related work

While the selector-to-value configuration in Fonto looks like how XSLT links up selectors to templates, they differ on a fundamental point: Templates in XSLT are usually unique to a selector; they see little reuse. The value space of a configuration variable in Fonto is usually small: they mostly consist of booleans and any non-discrete data is grouped nonetheless by the deduplication mechanisms described earlier. This makes optimizations like the Naive Bayes optimization work out of the box.

Lumley and Kay present optimizations for the XSLT case. In particular, they highlight the common use of DITA-class-substring selectors in DITA cases. However, such selectors and associated optimizations are not as applicable in Fonto. While Fonto does offer an abstraction[3] over the dita-class infrastructure for DITA-based editors, we advise against using it for configuration. This is because the class hierarchy usually produces unwanted results when used directly in our orthogonal configuration hierarchy and doing so may introduce a lot of complexity in the configuration as specific values frequently need to be overridden for specific sub-classes.

An example of this problem is found in specializing the list item element: not all specializations of the list item should be rendered or behave like list items:

[3] https://documentation.fontoxml.com/api/latest/fonto-dita-class-16324219.html

Take for example the `<consequence />` element in the Dita hazard domain. These elements should not be rendered like lists and they should not be indentable using the tab key, nor splittable using enter.

Because of these reasons, and because the dita inheritance structure does not give any pointers on how to create those elements, the configuration is most often denormalized to simply using node names.

3. Processing XML at interactive speeds

3.1. General XPath performance

The main performance bottleneck of Fonto is the performance of running XPath queries. XPaths is not only used to retrieve the schema experience configuration, but also to run generic queries. In order to speed up most queries, most of the optimizations described in the work of Kay[5] are implemented.

3.1.1. Outermost

Furthermore, a number of specific optimizations are implemented. One of the strongest optimizations regards the 'outermost' function, which returns the 'highest level' nodes from an input set. An example usage in Fonto is find and replace, which runs a query similar to following to determine the searchable text of an element:

```
descendant::node()[
  self::text() or
  (self::paragraph or self::footnote)
] => outermost()
```

This query returns all textnodes that are directly in a 'block', and any elements that are also a 'block'. Consider the following XML:

```
<xml>
  <paragraph>
    A piece of text
    <footnote>text is a string of characters</footnote>
    with a footnote in it
  </paragraph>
</xml>
```

When evaluating this query in a naive way, the path expression will result in a list of all descendants that match its filter, including all of the descendants that will be removed by the 'outermost' function.

A common optimization in functional languages like XPath is to perform lazy evaluation. We implemented this using a generator pattern inspired by LINQ[4]. However, lazy evaluation alone is not enough in this case. To further optimize

outermost, we pass a hint into the generator for the descendant axis, indicating whether it should traverse the descendants of the previous node returned or skip to the next one.

Consider the expression above, which consists of three parts: a descendant part, a filter part and the outermost function. Using lazy evaluation, we start at the outermost function, which requests the first node from the expression that feeds it. To compute this, the filter expression requests nodes from the descendant expression until it finds one that matches the filter, which is returned to the outermost function. The outermost function is not interested in the descendants of this node, so it now passes the "skip descendants" hint when requesting the next node. This hint, passed through the filter expression to the descendant expression, prevents the latter from traversing the subtree of the matching node and instead skips to the following node.

As find and replace recursively applies the query for text nodes and subblocks, this optimization basically changes the performance of that from $O(n\log(n))$ complexity to $O(n)$, as every subtree is now only traversed once instead of for each ancestor.

3.2. Schema validity

Fonto checks the validity of XML to the schema by converting each content model into a nondeterministic finite automaton (NFA), similar to the approach described by Thompson & Tobin[7]. We perform several optimizations to ensure this validation can happen quickly enough to not seriously impact editor performance.

Before a schema is loaded in Fonto, it is pre-processed in an offline compilation step. This converts the schema to a JSON format and simplifies the content model expressions. We first remove any indirections such as substitution groups and redefinitions. We then apply a number of rewrite rules to reduce these content models to equivalent but simpler models. For example, if any item within an `<xs:choice>` is optional, the entire choice can be considered optional, and all items within can be marked as required (`minOccurs="1"`). If multiple items within the choice were optional, this reduces the number of empty transitions that have to be created in the resulting NFA.

For example, the schema structure:

```
<xs:choice minOccurs="1">
  <xs:element name="employee" type="employee" minOccurs="0"/>
  <xs:element name="member" type="member" minOccurs="0"/>
</xs:choice>
```

Is equivalent to:

[4] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

```
<xs:choice minOccurs="0">
  <xs:element name="employee" type="employee" minOccurs="1"/>
  <xs:element name="member" type="member" minOccurs="1"/>
</xs:choice>
```

When compiling the reduced schema to an NFA we apply a few optimizations over the Thompson & Tobin algorithm in order to further reduce the size of the resulting automaton. Firstly, all branches of a choice that each process a single node (such as the "employee" and "member" branches in the example) are represented as a single transition. Large choices between multiple single-element options are a fairly common occurrence in schemata we've seen used in Fonto. This optimization reduces the number of possible paths in the NFA, reducing the memory and execution time costs for computing possible paths during validation. In real-world schemata, such optimizations may be more significant. For example, the content model of paragraphs usually consists of a repeating choice between a number of inline elements.

Secondly, and again to reduce the size of the NFA, any repetition of a term T with `minOccurs="1"` and `maxOccurs="unbounded"` is compiled to the automaton for T followed by an additional empty transition back to the start. The original Thompson & Tobin algorithm would build an NFA containing the automaton for T twice (once required, once optional repeating).

Our implementation for applying the resulting NFAs to XML content makes heavy use of pre-allocated typed arrays to store all state during traversal. Being a garbage-collected language, manual memory management is not commonly considered in JavaScript applications. However, validation being a very hot code path, preventing allocations serves both to avoid the performance overhead associated with them, as well as the later cost of having garbage collection reclaim those allocations. Ignoring the schema and NFA optimizations, manual memory management alone has led to a significant performance improvement compared to our implementation before these changes: applying a test NFA similar[5] to `<xs:any minOccurs="0" maxOccurs="unbounded" />` to a sequence of 10000 children went from around 111ms to just 17ms.

3.3. Indices

Many operations in Fonto applications require traversing parts of the DOM using XPath queries. While most of these traversals are limited to a reasonably local subset of nodes, there are some types of queries that have to traverse large numbers of nodes. In our experience, these most commonly take one of two shapes. One is to find a specific element or set of elements based on the value of some of their attributes, for instance, finding the target of a reference based on its `xml:id`.

[5]This particular test also involves determining all possible minimal traces through the NFA. Fonto can use this information to synthesize[6] missing elements.

Another is to find all descendant nodes of a certain type, often under some ancestor node, for instance, finding all footnotes in the document.

To prevent the full DOM traversal in answering these queries, it can help to perform some of the work ahead of time. To this end, Fonto allows defining specialized indices, which are then made accessible to XPath queries as functions that return associated data given some key. Fonto currently has three types of index:

- The *attribute index* can be defined for any attribute name (local name and namespace URI), and maps a given value to the set of nodes that have the attribute set to that value.
- The *bucket index* can be defined for any bucket, as discussed in an earlier section, and tracks all nodes matching that bucket that are currently part of any loaded document
- The *descendant index* tracks the set of descendant nodes matching a given selector under a specified ancestor. To make updates efficient, this selector is currently severely limited in terms of the parts of the DOM it may refer to.

Internally, Fonto makes heavy use of mutation observers (as defined in the DOM standard) and the resulting mutation records to represent changes in any of the loaded documents. Indices interpret these mutation records to determine which changes affect their data, and then update that data accordingly only if such changes are found.

In our current implementation, all indices should be explicitly defined by the application developer. We have considered automatically generating indices, such as attribute indices for attributes using the xs:ID type, but found that many schemata do not actually assign this type for their identifier attributes.

3.3.1. Indexing arbitrary computations

In addition to these indices, mutation records can be used to invalidate the cached results of any DOM-based computation, including XPath evaluation[4]. This requires tracking that computation's data dependencies in terms similar to the relations described by the mutation records. While not an index in the traditional sense, the similarity in terms of implementation and integration with the indices described above have led us to refer to this system as the *callback index*.

Summarizing from our earlier work, we use a facade between the computation and all DOM access to intercept these events and track corresponding dependencies in terms of the corresponding mutation record type (either *childList*, *attributes* or *characterData*). When mutation records are processed, we match them against these dependencies and signal (potential) invalidation of a computed value when the data depended on has changed. To avoid unnecessary work, re-computation is not performed automatically, but only on demand. This usually happens when the UI using the result is ready to update, instead of updating

these values many more times than could ever be observed by any user. It also avoids work in cases the UI decides not to re-issue the computation, for instance based on the result on another. For instance, the title of some figure in the document outline does not need to be recomputed if the entire section containing that figure is removed from the outline tree.

Both mutation records and raw DOM access operations can sometimes present a rather coarse-grained view of changes / dependencies. For instance, looking for a child element of a specific type may require visiting and examining all children of the parent node. This means that the corresponding computation may be invalidated unnecessarily if a node of a different type is inserted under the same parent. We use two mechanisms to reduce such unwanted invalidations.

First, dependencies registered by the DOM facade can specify a test callback in addition to the mutation record type. This test is evaluated against the changed document if a mutation record is processed with the matching type. If the test does not pass, the mutation record is ignored. We use this, for instance, to check whether a *childList* change affected the "parent node" relation for a given node.

Second, for most axis traversals in XPath we pass the *bucket* of the corresponding node test to the DOM facade. The resulting bucketed dependencies only invalidate the computation result for changes that match the bucket in question. For instance, the *childList* dependency for the selector `child::p` only triggers invalidation if a *childList* mutation record adds or removes `<p>` elements, not when only other nodes are added or removed.

3.3.2. Indexing and overlays

In Fonto, both the DOM and indices use a system of overlays to represent a future state of the DOM without actually mutating the original. For indices, these overlays are only initialized and then updated at the moments when the indices are actually used. As Fonto computes many possible future states at any time (for example, to determine the states of buttons in the toolbar), this avoids a significant amount of work for operations that do not use indices.

Furthermore, the lazy initialization of overlays allows computations based on the unmodified DOM to be re-used across different operations, as long as the value is computed before any modifications are made. In practice, this happens a lot. For example, tables use the callback index to derive a schema-neutral "grid model" representation from the DOM nodes. They then mutate this model, which in turn updates the schema-specific table DOM. As the entire table toolbar uses the same initial state of the DOM to compute the state of its operations, we only need to compute this grid model once. In fact, the same model has likely already been computed and cached in the callback index in order to validate the result of the previous operation, and is also used in rendering the table.

3.3.3. Fonto versus XML databases

In general, XML databases solve similar problems in terms of using indexing to make queries faster. However, the problem space differs in the following ways:

In Fonto, loading multiple megabytes of XML is a lot; we are on the web, so data needs to be small enough to download quickly and as a result will always fit in memory. In XML databases, gigabytes of XML is not rare and to be expected. In Fonto, authors on a bad internet connection don't want to wait ages for their documents to load, so larger documents are usually cut up into smaller chunks, which are loaded just in time for editing.

In Fonto, both queries and changes usually affect the same small part of the document. Furthermore, changes happen frequently. In an XML database, both changes and query subjects are often more spread-out. Because of this, our indexing approach needs to take frequent updates into account, and such updates need to happen quickly enough for users not to notice any slowdown.

In XML databases, it is acceptable to build indices during load time. In Fonto, the editor should be up and running as soon as possible. This means we can not build a large index at start-up if computing that index takes a non-trivial amount of time. Also, because Fonto usually does not run for a long time, it is probable that an index will never be used.

The current cache invalidation approach so far fits that set of requirements. A reusable result is often only computed when necessary, and forgotten when it no longer applies. A larger computation can be spread out over multiple separately indexed entries in order to make recomputation more efficient in cases where only part of these are invalidated.

4. Conclusions

Lots of tricks are possible to make user-friendly authoring of XML fast, even in JavaScript and webbrowsers. When Fonto was two years old, we received a lot of feedback on the performance of documents of 100KB of XML. Currently, we have clients working with single documents ranging into the megabytes, configured using complex schemata like JATS or the TEI standards. Using approaches like JIT loading and chunking, we have clients working with tens of thousands of documents which we are unable to even download and keep in memory simultaneously.

5. Future work

At Fonto we continue to move to declarative formats to specify the configuration, behavior and UI of the editor. We prefer to use existing standards, and continue to improve and extend our XPath, XQuery and XQUF implementations. For configuration, the closest analogue in terms of declarative formats seems to be CSS.

However, we prefer to keep using XPath for our selectors. We also have several property types that go far beyond the property values commonly found in CSS, including the way the appearance of elements is defined as a composition of visual components and widgets. It is likely we will need to develop a custom format to support this combination.

In mutating the XML DOM, moving to XQUF has the additional advantage that we can use the callback index to track the dependencies of an operation, and therefore only recompute its effect (represented as a pending update list) and state (based on the validity of the resulting DOM) when required. In addition to converting current JavaScript-based primitives, this requires allowing other bits of state to be dependency-trackable in the same way as the DOM, including the current selection.

To minimize work even further with minimal impact on the way developers configure Fonto, we wish to further expand indices and the callback index into a framework for general incremental computation. This requires dependencies between index entries (already partially implemented), which allow for memoization by isolating one computation from another. To propagate invalidations caused by DOM changes efficiently, we also need to add a way to stop this propagation when the new result for some computation equals the previous value, as that means results depending on that value can be reused.

Bibliography

[1] XQuery update facility 3.0 https://www.w3.org/TR/xquery-update-30/

[2] A minimalistic XPath 3.1 implementation in pure JavaScript https://github.com/FontoXML/fontoxpath

[3] https://drafts.csswg.org/selectors-4/#specificity-rules

[4] Martin Middel. Soft validation in an editor environment. 2017. http://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf

[5] Michael Kay. XSLT and XPath Optimization http://www.saxonica.com/papers/xslt_xpath.pdf

[6] Martin Middel. How to configure an editor. 2019. https://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf

[7] Thompson, Henry S., and Richard Tobin. "Using finite state automata to implement W3C XML schema content model validation and restriction checking." *Proceedings of XML Europe*. Vol. 2003. 2003.

[8] Various authors. The DOM Living Standard. Last updated 16 January 2020. https://dom.spec.whatwg.org/

XSLWeb: XSLT- and XQuery-only pipelines for the web

Maarten Kroon
<maarten.kroon@armatiek.nl>
Pieter Masereeuw
<pieter@masereeuw.nl>

Abstract

XSLWeb is an open source and free to use web development framework for XSLT and XQuery developers. It is based on concepts similar to frameworks like Cocoon and Servlex, but aims to be more easily accessible and pragmatic.

1. Introduction

When a webbrowser asks information from a webserver, the data sent to the server may look like this:

```
GET /demojam HTTP/1.1
Host: www.xmlprague.cz
User-Agent: Lynx/2.8.9dev.16 libwww-FM/2.14 SSL-MM/1.4.1 GNUTLS/3.5.17
..
```

Now suppose that a request would look like this:

```
<req:request>
    ..
    <req:path>/demojam</req:path>
    <req:request-URI>http://www.xmlprague.cz/demojam</req:request-URI>
    ..
</req:request>[1]
```

If this were true, generating and serving a webpage could easily be done with XSLT, for example:

```
<xsl:template match="/req:request[req:path eq '/demojam']">
   <xsl:apply-templates select="doc('demojam4ever.xml')"/>
</xsl:template>
```

[1]Namespace definitions are omitted from all examples in this document.

2. Why XSLWeb?

XSLWeb was created out of the need to have a pipelining platform that is trivially simple to use for XSLT developers. Of course, we had a look at existing technologies such as Cocoon, XProc and Servlex.

At the time that XSLWeb was created, we had a lot of experience with *Apache Cocoon*. Unfortunately, the Cocoon project lost the interest of its developers - the latest release dates from 2013, while the one before that dates from 2007. Version 3.0, a major rewrite, never made it further than the Alpha version in 2011. Furthermore, we found that Cocoon, albeit very powerful, had a rather steep learning curve, so in the end it proved to be not so easy to use after all.

The Servlex platform does not seem to be actively developed anymore. We have no practical experience with it.

XProc is of course a very serious technology for creating pipelines. Unfortunately, the language requires some time before you have a feel for it[2]. For us, XProc's main drawback was that it is not specifically intended for use in a web service environment. Even in the case of *Piperack* (the web companion program of the XProc processor *Calabash*), you do not have easy access to all information inside the HTTP request, while such information can be vital for many web applications.

Our most important reason for moving away from XProc was that we really wanted a platform that is so straightforward that a person with XSLT knowledge can use it almost at once. Furthermore, we wanted it to be very simple to combine data coming from different sources into one pipeline. Cocoon and XProc require you to set up distinct pipelines that you eventually have to merge. In XSLWeb, you can, in most cases, reference external information, such as a REST services or relational databases, from the XSLT stylesheet itself. It does so by providing a large set of extension functions (such as functions for querying databases and manipulating result sets).

In short, XSLWeb aims to be practical and very easy to use for XSLT (and XQuery) programmers. It has the following characteristics:

- It gives access to the full HTTP request in an XML representation;
- It supports the full HTTP specification - GET, POST, PUT, and all other methods;
- It makes pipelining trivially easy;
- It allows XSLT and XQuery programmers to program things at the moment they need it - i.e. in their stylesheet or XQuery script.
- It offers an XML representation of the HTTP response;
- It allows caching;

[2]Of course, XProc 3.0 makes the programmer's life a lot easier, but the concepts remain the same.

- It allows access to static content (assets);
- It has a large set of extension functions, including, e.g.:
  - Functions for manipulating the request, the session and the response;
  - EXPath file functions and EXPath http functions;
  - Spawning external processes;
  - Sending e-mails;
  - Image processing;
  - ZIP file processing;
  - SQL processing;
  - and even (experimental) server side Javascript.
- Allows the addition of user defined extension functions in Java, using the Saxon API.

3. XSLWeb in a nutshell

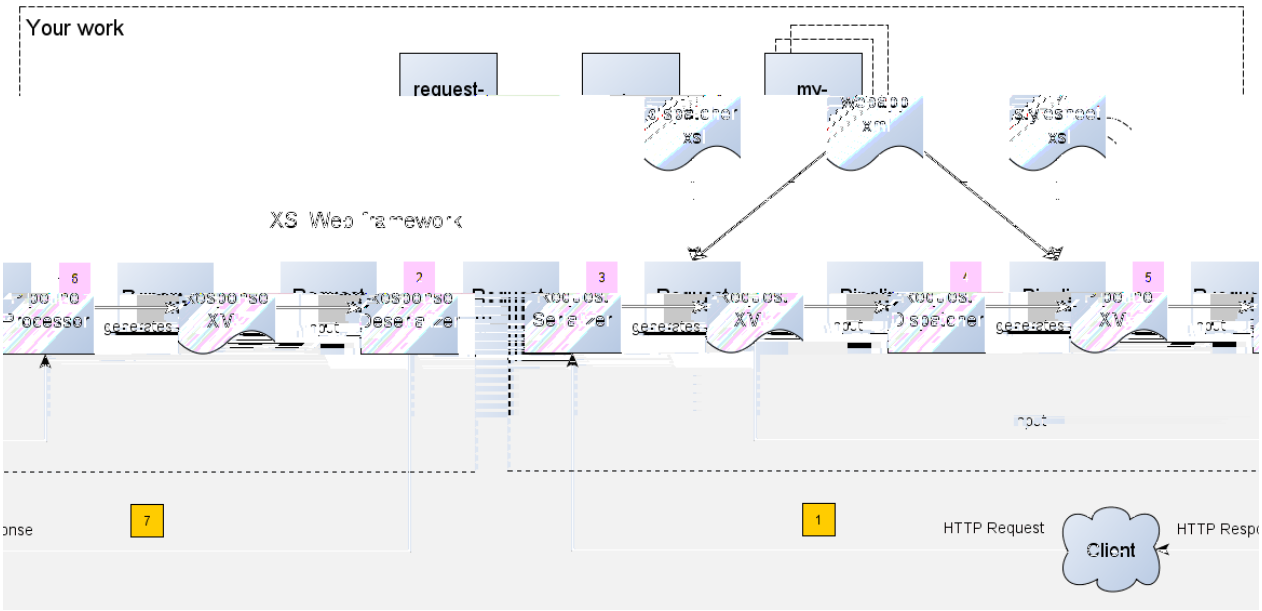
Using XSLWeb, XSLT/XQuery developers can develop both web applications (dynamic websites) and web services. In essence, an XSLWeb web application is one or more XSLT stylesheets (version 1.0, 2.0 or 3.0) or XQueries (version 1.0, 3.0 or 3.1) that transform an XML representation of the HTTP request (the *Request XML*) to an XML representation of the HTTP response (the *Response XML*). Which specific XSLT stylesheet or XQuery (or pipeline of XSLT stylesheets and XQueries) must be executed for a particular HTTP request is governed by another XSLT stylesheet, the *request dispatcher stylesheet*, which is a normal stylesheet that dynamically generates a pipeline, represented by an XML <pipeline> element.

During transformations, data sources can be accessed using a library of built-in extension functions that provide HTTP communication (for example to consume REST or SOAP based web services), file and directory access, relational database access, and so on.

The result of a transformation pipeline can be serialized to XML, (X)HTML or plain text format and using specific serializer pipeline steps to JSON, ZIP files, PDF, Postscript or RTF (using XSL-FO and Apache FOP).

The configuration of an XSLWeb web application can be specified in an XML configuration document called *webapp.xml*. An XSLWeb server can contain multiple separate web applications.

The Figure 1 illustrates the flow of control within XSLWeb.



1. A HTTP request is sent from a client (a web browser or webservice client).
2. The HTTP request is serialized by the Request Serializer to a Request XML document. All information of the request is preserved in the XML representation.
3. The Request XML is the input of the Request Dispatcher, which transform the it using the webapp-specific XSLT stylesheet *request-dispatcher.xsl*. The output of this transformation is a pipeline specification, in the simplest form only specifying the path to an XSLT stylesheet that will be used to transforming the Request XML to the Response XML. This specification could also contain a pipeline of multiple XSLT transformations and XML Schema or Schematron validations.
4. The pipeline specification is the input for the Pipeline Processor, which reads the Pipeline XML and executes the pipeline transformation and validation steps. The input for the first transformation in the pipeline is the same Request XML as was used as input for the Request Dispatcher.
5. The Pipeline Processor executes the pipeline of XSLT stylesheets, XQueries and validations. The last transformation in the pipeline must generate a Response XML document.
6. The Response XML is then passed on to the Response Deserializer, which interprets your Response XML and converts it to a HTTP response, which is sent back to the client, a web browser of webservice client (7).

Figure 1. The flow of a HTTP request to a HTTP response within XSLWeb

3.1. The Request XML and the Response XML

The Request XML is an XML representation (or XML serialization) of the HTTP Request. It contains all information of the raw request, including normal headers, request parameters, request body, file uploads, session information and cookies.

The Response XML is an XML representation (or XML serialization) of the HTTP Response. It contains the HTTP headers, the response body, session information and cookies.

Both the Request XML and the Response XML are formally described in an XML Schema, to which they must conform.

3.2. The Request dispatcher XSLT stylesheet

The task of the XSLT stylesheet *request-dispatcher.xsl* is to dynamically generate the pipeline specification that is then used to process the Request XML and convert it to the Response XML. The input of the request dispatcher transformation is the Request XML which implies it has all information available to generate the correct pipeline. The output of the request dispatcher transformation is a pipeline specification.

The resulting pipeline specification contains one or more transformation, query, validation or serialization steps. The input of the first stylesheet or query in the pipeline is the Request XML, the output of the last stylesheet in the pipeline must conform to the Response XML schema[3].

The pipleline specification is formally described in an XML Schema, to which it must conform.

3.2.1. Example pipelines

Below is an example of a very basic request dispatcher stylesheet that generates a valid pipeline for the HTTP request *http://my-domain/my-webapp/hello-world.html*:

```
<xsl:stylesheet ..>

 <xsl:template match="/req:request[req:path eq '/hello-world.html']">
   <pipeline:pipeline>
     <pipeline:transformer name="hello-world"
          xsl-path="hello-world.xsl" log="true"/>
   </pipeline:pipeline>
 </xsl:template>

</xsl:stylesheet>
```

[3]This implies that in XSLWeb, other than in for example Cocoon and XProc, pipelines are generated dynamically.

The following example uses the request parameter lang in the request *http://my-domain/my-webapp/hello-world.html?lang=en* to determine the stylesheet. This *lang* parameter is also passed to the stylesheet as a stylesheet parameter:

```
<xsl:template match="/req:request[req:path eq '/hello-world.html']">
    <xsl:variable name="lang"
        select="req:parameters/req:parameter[@name='lang']/
req:value[1]"/>
    <pipeline:pipeline>
        <pipeline:transformer name="hello-world"
            xsl-path="{concat('hello-world-', $lang, '.xsl')}">
            <pipeline:parameter name="lang" ..>
                <pipeline:value>{$lang}</pipeline:value>
            </pipeline:parameter>
        </pipeline:transformer>
    </pipeline:pipeline>
</xsl:template>
```

A slightly more complicated pipeline shows how you could render to different formats (e.g., HTML, PDF, EPUB) by using a request parameter to generate format-specific pipelines:

```
<xsl:variable name="reqparms" as="element(req:parameter)*"
    select="/req:*/req:parameters/req:parameter"/>

<xsl:template match="/req:request[req:path eq '/result-document']">
    <xsl:variable name="format" as="xs:string?"
        select="$reqparms[@name eq 'format']/req:value"/>

    <pipeline:transformer xsl-path="retrieve-xml.xsl"/>

    <xsl:choose>
        <xsl:when test="$format eq 'html'">
            <pipeline:transformer xsl-path="xml2html.xsl"/>
        </xsl:when>
        <xsl:when test="$format eq 'pdf'">
            <pipeline:transformer xsl-path="xml2fo.xsl"/>
            <pipeline:fop-serializer/>
        </xsl:when>
        <xsl:when test="$format eq 'fo'">
            <pipeline:transformer xsl-path="xml2fo.xsl"/>
        </xsl:when>
        <xsl:when test="$format eq 'epub'">
            <!-- xml2epub.xsl generates a response with a body that
                 contains an XML container file in a format the
                 zip-serializer can serialize.
            -->
```

```
            <pipeline:transformer xsl-path="xml2epub.xsl"/>
            <pipeline:zip-serializer/>
        </xsl:when>
        <xsl:otherwise>
            <pipeline:transformer xsl-path="error.xsl"/>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

3.3. Pipelines

A pipeline consists of:

- One or more of the following transformation pipeline steps:
  - *transformer*: transforms the input of the pipeline step using an XSLT version 1.0, 2.0 or 3.0 stylesheet;
  - *query*: processes the input of the pipeline step using an XQuery version 1.0, 3.0 or 3.1 query;
  - *transformer-stx*: transform the input of the pipeline step using a STX (Streaming Transformations for XML) version 1.0 stylesheet.
- Zero or more of the following validation pipeline steps:
  - *schema-validator*: validates the input of the step using an XML Schema version 1.0;
  - *schematron-validator*: validates the input of the step using an ISO Schematron schema.
- Zero or one of the following serialization pipeline steps:
  - *json-serializer*: serializes XML output to a JSON representation;
  - *zip-serializer*: serializes an XML ZIP specification to an actual ZIP file;
  - *resource-serializer*: serializes a text or binary file to the response;
  - *fop-serializer*: serializes XSL-FO generated in a previous pipeline step to PDF using the Apache FOP XSL-FO processor.

The output of the pipeline can be cached by specifying extra attributes on the <*pipeline:pipeline*/> element.

3.3.1. Goodies

XSLWeb extends the standard XSLT/XPath 1.0, 2.0 and 3.0 functionality in a number of ways:

- XSLWeb provides a number of built-in XPath extension functions that you can use to read and write files and directories, execute HTTP requests, access the

Request, Response and Context, Session and WebApp objects, log messages, send e-mails, query databases and so on;

- Other pipelines can be called from within a stylesheet and the result of this nested pipeline can be used or embedded in the calling stylesheet by passing a URI that starts with the scheme "*xslweb://*" to the standard XSLT *document()* or *doc()* function;
- URLs that are passed to XSLT's *document()* or *doc()* function and that must be proxied through a proxy server can be provided with two extra request parameters: *proxyHost* and *proxyPort*;
- Pipeline stylesheets are also provided with any parameters that are defined within the element *pipeline:transformer* in the Request dispatcher stylesheet *request-dispatcher.xsl*. The parameters only have to be declared in the stylesheets (as <*xsl:param*/> elements) when they are actually used;
- Within every transformation a number of standard stylesheet parameters is available, such as:
  - The configuration parameters from the parameters section in the the configuration file of an XSLWeb application (*webapp.xml*);
  - *config:home-dir*: the path to the XSLWeb home directory;
  - *config:webapp-dir* and *config:webapp-path*: the paths to the base directory of the webapp and the path in de url to the web application, respectively;
  - etc.

3.4. Web applications

An XSLWeb installation can contain multiple separate web applications. A web application can be added under the folder «*xslweb-home*»/*webapps* and has the following folder structure:

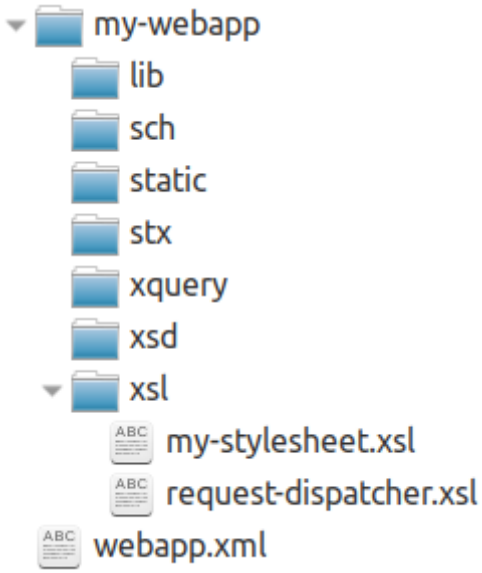


Figure 2. XSLWeb folder structure

Apart from the top-level folder (here: *my-webapp*) and one additional XSLT- or XQuery-file, the only required files are *webapp.xml* and *xsl/request-dispatcher.xsl*.

The folder *my-webapp* can have any name you like (provided it doesn't contain spaces or other strange characters; its name may come back in the URL of the application). The folder *lib* can contain any custom XPath extension functions you have developed in Java and 3rd party libraries they depend on. The folder *static* contains all static files you use in your web application, like images, css stylesheets and javascript files. The folder *xsl* contains the XSLT stylesheet *request-dispatcher.xsl* and at least one pipeline XSLT stylesheet that transforms Request XML to Response XML. The folders *xsd* and *sch* can contain XML Schema or Schematron validation specifications. The file *webapp.xml* contains further configuration of the web application.

3.4.1. The file webapp.xml

The file *webapp.xml* contains the configuration of the web application. It must conform to its own XML Schema and contains the following configuration items:

- *Title*: The title of your web application;
- *Description*: The description of your web application;
- *Development mode*: whether or not developmen mode is active.The development mode mainly defines caching, buffering and logging behaviour of the application;
- *Resources*: The definition of requests to static files that should not be processed by the request dispatcher (but should be served straight away) and the duration these resources should be cached by the browser (default 4 hours);

- *Parameters*: The definition of webapp specific configuration parameters that are passed as stylesheet parameters to every XSLT transformation;
- *Jobs*: The definition of scheduled jobs (a crontab-like facility, used when you want to execute a pipeline (repeatedly) on certain moments without user interaction);
- *Data sources*: the definition of JDBC data sources;
- *FOP configurations*: configurations for the Apache FOP serialization step.

3.5. Running XSLWeb

XSLWeb is a Java application that conforms to the Java Servlet Specification. It can be run from within any Java application server, such as Apache Tomcat. In development and testing situations, it can also be run with its own built-in servlet container.

3.6. Performance and data model

XSLWeb has been subjected to performance and stress tests as part of a product selection process in a department of the Dutch ministry of internal affairs. XSLWeb was required to transform randomly chosen XML texts to HTML within 140 ms. One of the problems of measuring XSLWeb's performance is that measurements are influenced by the size of the XML documents to be transformed and by the efficiency of the stylesheets.

The machines used for the test were a fast laptop (Core i9 processor with 6 cores, and SSD) and a slower virtual server with physical disk and a Xeon CPU E5-2690 processor. The stylesheets for this test had been routinely used for off-line production of the same HTML pages. Some of the XML files were large, and some stylesheets were rather inefficient.

Two XML collections with different document formats and stylesheets were used for the test, but due to lack of time, one of the tests could only be performed on the slower machine.

Given a load of approx. 15 concurrent requests, we obtained the following averages:

Table 1.

	Fast laptop	**Server**
Test 1	99% of requests served within 61 ms	95% of requests served withing 161 ms
Test 2	n/a	99% of requests served within 55 ms

	Fast laptop	**Server**
Average response times (both tests)	14 ms	88 ms
Worst cases	1287 ms	7292 ms

Investigation of the worst cases revealed that performance was severely hampered by the inefficiency of one and the same stylesheet. It should be relatively easy to correct this stylesheet in such a way that it navigates the large XML document with less overhead and by switching to XSLT 3.0 and use (hash)maps.

Internally, XSLWeb uses Saxon's efficient tiny tree model, as discussed by Michael Kay on the XML Prague 2018 conference.

4. XSLWeb in the real world

XSLWeb is used, among others, in webservices and websites of KOOP (Kennis- en Exploitatiecentrum Officiële Overheidspublicaties; Knowledge and Exploitation Centre Official Government Publications, a department of the Dutch ministry of internal affairs), the Dutch ministry of foreign affairs (treaty database), the Dutch *Kadaster* (land registry), Octrooicentrum (patents) and many other places.

But: why not try it yourself? It's free, easy to use, and best of all: it's fun!

A. References

- XSLWeb is available on Github, https://github.com/Armatiek/xslweb.
- For more information about XSLWeb, refer to its documentation: https:// raw.githubusercontent.com/ Armatiek/ xslweb/ master/ docs/ XSLWeb_3_0_Quick_Start.pdf.
- XProc: https://www.w3.org/TR/xproc/ and https://xproc.org/.
- Calabash: https://xmlcalabash.com/
- Piperack: https://xmlcalabash.com/docs/reference/piperack.html
- Cocoon: https://cocoon.apache.org/.
- Servlex: http://servlex.net/.
- About the Tiny Tree model: https:// archive.xmlprague.cz/ 2018/ files/ xmlprague-2018-proceedings.pdf# d6e1190 or http:// www.saxonica.com/ papers/xmlprague-2018mhk.pdf.

Things We Lost in the Fire

Geert Bormans

Ari Nordström

Abstract

This is about all those markup consulting projects where you realise that something isn't quite as it should be. Early on, your internal alarm bells are set off by a technology choice, legacy systems or processes, or maybe internal conflicts, and you realise there are some hard decisions to make. Yes, you have bills to pay but is this one of those projects you should stay away from... or have stayed away from in the first place?

For example, what if you realise that your project was never meant to succeed? What if a legacy system stands in the way of your every deliverable but is regarded as untouchable? And what if you've been brought in to solve pressing and immediate problems but office politics, legacy systems, fundamental misconceptions or all of the above stand in your way? What if the team's skill set would be a trigger to obstruction and sabotage? What if people were losing their jobs if you were successful? Or maybe it's simply a disruptive atmosphere and more than anything it's all about breaking through that.

We take a hard look at past projects and try to analyse what went wrong and why, and what we learned from them. Perhaps we can impart some degree of objectivity on a novice in the field, or at least have him or her think again. If there is success - flipping adversity into success - we'll be more than happy to claim credit.

1. We Call Ourselves Grumpy Old Men

Let us introduce ourselves. We're a pair of somewhat aged markup geeks with a combined 50 years of consulting experience between ourselves[1]. This is not the paper that will reveal all. However, it is the paper that will discuss some of the implications of those 50 years.

Or at least have a few laughs while reminiscing.

Bitching about our projects, we came to realise that many of our projects follow a pattern. This pattern can be illustrated through the diagram below.

If we would reduce the number of stakeholders in a project to three, there seems always to be someone (or a team) in power, a team doing the technical work, and a consultant either providing advice or assisting development.

[1] A lot of which was spent bitching about what had already been.

All three stakeholders have a certain view on the project: where the resources should go, what technology or approach should be used, and what the best route to success would be. Obviously the project's succes would depend on the space where all views meet.

However we came to the conclusion that this zone of succes, for various undefined reasons, often is a “hidden zone” or even a “forbidden zone”.

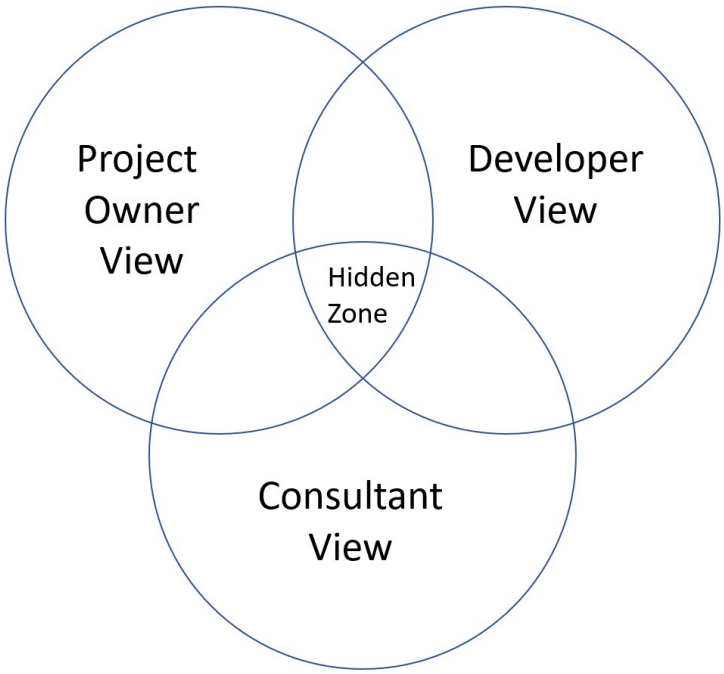


Figure 1. That Basic Venn Diagram

If you're one of us - a grumpy old man, basically[2], with some years of angled brackets under your belt - you will recognise this diagram, nodding and — perhaps — bitterly thinking back in time. You will have had many experiences relating directly to this simple diagram. Memories.

This is a paper about those memories. We'll tell you about some of *our* memories, reliving the key points, bitching about what we went through while thinking — hoping, even — that you'll be nodding right along with us.

[2]No harm in being a grumpy old woman here.

2. The Things That Lit the Matches

2.1. Programming Languages as Religion

Being too religious about a programming language or a vocabulary does not always help a project.

Some years ago I held a workshop after the audit of an XML transformation code base. I was invited to do so because the customer found out that very small functional changes to the existing proprietary transformer really took developers a lot of time to develop and testing always revealed that a small fix at one point raised another issue elsewhere.

It was obvious they were using the wrong technology for the job at hand.

I managed to convince the managers in the workshop that a different technology (XSLT to no surprise) would pay off quickly, as it would be a much better fit to the job. After a coffee break we would discuss migration plans, training...

None of the developers present realised I understood the local language, so near the coffee machine I overheard an agitated discussion about the techology. One of the developers mentioned firmly that XSLT would only be used "“over his dead body”"

I was assured by the manager that they would handle the situation without much problems and we planned migration, training, contracts.

After my flight back home, there was a message on my voicemail thanking me for the audit and workshop. The project however was cancelled early because of developer protest.

2.2. The Strict DTD

I wrote a set of DTDs and a bunch of transformation pipelines for a client that was merging their content with another company's (as in actually merging two sets of documents with the same text but with differing tagging). Among the DTDs was an exchange DTD, an intermediate format when converting from one format to another, and a somewhat more strict DTD for authoring the merged content. The two DTDs were related, of course; the exchange format was the intermediate format used when converting external documents from whatever source they used to the new authoring DTD.

In my mind, the big job was to move from the external source to the exchange format. Moving from the exchange format to the authoring format was mostly about tidying things up. Typically, the first pipeline, from the source to exchange, would be in the range of 80 or so XSLT steps while the second pipeline, from exchange to authoring, was 18 steps. The authoring format would still have various optional structures, though, and #IMPLIED attributes, as the merge resulted in inevitable compromises. My plan was to add a Schematron to do some additional validation and tightening-up, based on the authoring format.

But when some of the other company's devs heard about my plans, they said "we MUST do a stricter DTD to aid the authors!" Nonplussed, I repeated the bits about additional validation using a Schematron. Maybe they missed that part.

"No, we MUST have a strict DTD!"

This was getting weird. I explained that there was no way to do that strict DTD - things would have to remain optional and #IMPLIED, or quite a few documents wouldn't be valid. I asked what they had against the DTD+SCH combo. We're talking about well-known and well-supported standards. There were no actual answers at first, only the insistence of a strict DTD, "because authors have a hard time knowing what to do if the DTD doesn't properly guide them."

I went through what's normally my sales pitch about the usefulness and adaptability of Schematron rules, and how they can help authors in ways DTDs cannot. And thinking that I shouldn't have to be doing this.

Much later, I had a one-on-one with one of the devs and the conversation drifted to Schematrons. And after some discussion, he finally said "Schematron is a Java library, right?"

2.3. Page by Page

I was tasked with collecting requirements from a number of companies owned by the same global monster in order to design a single system and associated schemas and processes for handling the documentation and publishing needs of all those companies.

This one company I talked to needed their manuals published in 27 languages (EU, mostly, as you might guess) but their output was handled in, shall we say, a doubtful manner.

Basically, they had one MS Word file per manual page. A 70-page manual would thus consist of 70 files. They then made sure that the translations of one page - or file - would fit into that same page, including images and everything.

All 27 languages would thus have the same number of pages, which they thought was great and really cost-saving because they'd then be able to use the same ToCs and indexes regardless of language.

And so they thought the natural progression from this state would be to do those files in XML instead, because writing and translating is supposed to save a lot of money. Right??

2.4. We've Always Done This

Recently I developed (yet another) MS Word transformer to XML. Yes, customers do use MS Word for XML authoring.

I was called in because a first attempt had failed over the past few years.

This story is about legal publishing. Some acts have books, chapters and sections, others just have chapters, and yet others have deeper nesting. The information model caters for all of that.

The contractor had told the publisher that it was impossible to differentiate between a chapter in one document or another if chapters would not always be styled reliably on the same level. For example, the project could only be successful if chapters were always styled as `heading3` and sections as `heading4`. So the publisher started restyling all their content to be in line with the contractors requirements... and gave up on that effort after some 1,000 pages out of 200,000.

They realised it would take years just to update the legacy laws, and they would never be able to publish new laws in time in the future.

2.5. We've Always Done This, Part 2

I inherited the stylesheet development work of an old DITA OT project in the automotive industry for print and HTML. Challenging, quite a few car types, over 20 languages including Hebrew and Arab.

For some years by the time I joined, the information architect had defined every suggestion from the car manufacturer about the printed pages into separate tasks for stylesheet development. The reviewers of those printed manuals all had different ideas, sometimes conflicting, and the response of the information architect had always been to allow for exceptions to be fixed in the stylesheets.

It had never occurred to them to fix anything in the actual DITA content, except to add another output class every once in a while.

There was a continuous pressure to publish, one car model after another, one language after another, each one leading to incremental changes to the stylesheets, causing massive delays in delivering the printed manuals. The DITA OT developer had indicated he no longer had the time to handle all the work coming in.

So I was hired for some smaller development tasks and inherited a code base that had more `xsl:if` and `xsl:when` clauses then it had templates. I slowly started changing the mentality from fix-in-stylesheets (one part time developer) to fix-in-content (5 full time editors) but never managed to change course in time.

Eventually my contractor lost his contract because of the delays.

The same contractor then got the brilliant idea to start using their expertise and DITA for a legal content publishing project. I politely passed on that one.

2.6. 90-minute Standups

In the early days of using agile development methods, I worked half time for a somewhat smaller integrator. A single big project consumed most of the company's resources and also charged some of the work out to consultants such as myself.

In order to glue all of teams efforts together we had a daily standup meeting with about 45 people. The daily standup took an hour, sometimes up to 90 minutes. Working only halftime in the project, I was still attending the standup every day. Well, I got dismissed from the standup after I started mentioning the daily meeting explicitly in my timesheets.

In the end, the end-customer cancelled the project for budgetary reasons.

2.7. The Build Is Green

Speaking of agile development, in a project I once worked in there were some trained “scrum masters” doing the Java development part of the work. They were extremely keen on using all the techniques they learned in the project: pair programming but mainly Test Driven Development.

After setting up a test environment and a big screen, their only focus to the project became “making the build green”.

At some point we had a couple of rough days in the project, and "the build" had been red for days. Suddenly the scrum masters started singing “the build is green, the build is green” and prepared to leave for the evening. Well it had never been as bad as on that particular point. Everything broke apart, no results to be found anywhere.

Pointing that out the response was: “You can not spoil the fun, the build is green”.

They left the office with a suggestion... if there is something you don't like, you will have to write a test for that.

2.8. Make It Better

We do often get so focused on the technical aspects of a task that we forget about the legacy or the team.

In a very recent gig the task was very clear: work through the codebase and make it better. It needs to be more robust and should run faster. Reaching the set goals would not be extremely difficult. The existing code was already above average, and improving on some of the techniques used would already make enough of a difference. However, it was hard to get all the teams to accept the things I was doing until a sprint evaluation revealed that people disliked the black-box approach I took.

The development team came up with a plan to have a weekly meeting to discuss the changes made so far and the reasons for making the change. That communicative approach made a huge difference for the atmosphere and cooperation.

It is all so obvious after the facts, but it taught me to have even more attention for the different sensitivities in a team.

2.9. An XSD for Appearances

I was tasked with designing an XSD for a client. The XSD was supposedly for describing messages in a system. The work was to be done on site and my contract was for six weeks. Oh, and this was in around 2005, before the advent of the smartphone, mobile internet, etc.

There was no computer ready for me when I arrived. That is, the hardware was there but the software wasn't. It was "on order" from the IT department. I didn't have much interaction with the other devs, just specs from an initial implementation proposal and some mostly irrelevant background information. I also didn't have much contact with the other devs; they were busy coding their thing based on the implementation proposal, plus their own ideas (which I found out later). The only guy I really talked to was the project manager, mostly because he was a friend.

Luckily, I had my own personal laptop. No internet connection since this was on-site and foreign computers were a no-no, so no email, but I had my own software for the XML stuff, plus all those implementation proposal specs (which I pretty much followed to the letter).

I delivered an XSD that did exactly what it was supposed to on time during week #6, on a floppy disk since I had no means to deliver without an internet connection.

On the Friday of week #6, I had lunch with the project manager. He said "we're not going to use your XSD." And the computer I was appointed never got its software.

I later realised that the waterfall, up-front approach was probably not what the devs or the company wanted and their solution never included an XSD. It was, however, described in the proposal, and so it was preordained which meant that it had to be included in the project. Yes, that's me. But *my XSD was never meant to be included in the solution*.

2.10. 'oy' DaSIQjaj[3]

Remember that Strict DTD? It was meant to do both the work of a DTD and what you'd normally entrust Schematrons and style guides with, but the project was also very much about office politics.

They wouldn't give an inch on the strictness of that DTD. We couldn't tighten the DTD because that would invalidate about 10,000 large legal documents.

And then they decided the DTD must be in their local language rather than English, which was the only language many of our devs understood. I thought they might as well do the DTD in Klingon, in that case, so very quietly I added several attributes in Klingon. It took them a couple of months to spot them.

[3]Klingon for "May you endure the pain".

As I write this, this has been "solved" by introducing a black box that translates XML between the DB storage format ("loose DTD") and the authoring format (the "strict DTD"), every time something needs to be edited. The two are *not* fully compatible so there are a lot of problems and more being discovered every day.

2.11. Open Source as Policy

Around 20 years ago, a publisher invited me to look into the publication process of one of their somewhat more intensive publications.

There was a team of about 10 developers. This means the department had 10 computers. One of the developers had designed a system that would make the publication, occupying all the computers in the department full time for three days.

Incidentally, the solution required Sablotron, Perl and a lot of network communication.

That implied they had to wait for a long weekend to publish. If something failed during the process, they'd have to wait for the next long weekend. The publication was often delayed by several months because of that.

I looked into this and could prove that the entire process could be done in just a few hours on a single machine. However, this required the use of a reasonably priced licensed software.

The manager refused the proposal because it failed against their open source software-only policy... and the existing solution did work, didn't it? They continued to use the existing approach for a couple more years.

2.12. Old Software

Speaking of licensed software, I had to work around missing functionality doing XSLT development in the DITA Open Toolkit a number of times, simply because it came bundled with a 10 year-old XSLT2.0 processor. That seemed to be the only option to use a `node-set()` extension function without paying a license cost.

Effectively, in several projects over the years, I had to spend multiple hours to work around a license cost equalling about one of those hours.

2.13. Subscription Services

I am running a subscription service providing some transformed data in specific formats to subscribers. However I am getting the actual information for this service from crawling databases via a web interface. This is all nicely covered by a ten year old contract with the information provider.

But then, one day the information provider was purchased by an international consortium and the new owners simply blocked my server's IP address. Informa-

tion could now only be obtained through the API they had developed, but they no longer had room for another partner.

It took me weeks and a good lawyer to force myself into a partner agreement. Then it took me weeks to replace the crawler with a service that communicated with the API.

During that period I lost half of the subscribers to a service that was supposed to update at least weekly.

2.14. You Can Choose Any Software You Want

A well-known global automotive manufacturer needed to XML-ify their aftermarket documentation using a system they wanted us to design and build (at the time they mostly used PageMaker for their aftermarket docs). I was first out, to analyse requirements, write DTDs, and recommend ways to do the system so it would support everything in a really, really cool way.

My analysis of the existing documents (glovebox manuals, accessory catalogues, warranty booklets, etc) suggested a lot of savings to be made by modularising the information, aided by some light profiling (different engines, gearboxes, etc), linking, and general standardisation of processing. I wrote a set of DTDs using all the then-modern technologies such as extended XLink[4] and set up a list of cool things we could do with the system.

This is when they came back with a bunch of requirements:

- You can choose any editor you want but it has to be [X].
- You can choose any underlying DB you want but it all needs to be based on [Y].
- You can choose any formatting engine you like but it has to be [Z] (but please use any standards you want).
- Oh, and we need you to move all the old SGML stuff we have for service info, a parts DB, and the like to the new system as well. Which means going from SGML to XML, migrating old DBs, etc.

Let's see. No extended XLink (because of no way to properly handle out-of-line lookups in the editor OR the DB). No inline `CDATA`-based links either (because the editor did not do them). Which meant generating `ID/IDREF` pairs in a really mad process whenever handling the information, plus a LOT of other compromises.

The new system was added a document management layer that required horrible processing attributes on many elements (this is how we found out that the editor's parser at the time included a hard 8,000-character attribute length limit). The first login in test took about 30 minutes. In the words of the software architect: "This could have been a really fast system without [Y]." I said something in a

[4]Xinclude wasn't really a thing at that point.

similar vein about [X], and the guy who did most of the formatting cursed about [Z].

The system ended up costing about ten or fifteen times the budget, required a LOT of bug fixing (the cost of which wasn't part of the inflated budget), but it's supposedly still in use.

2.15. Not Hawt Enough

By the way, that well-known automotive maker's technical writers were an opinionated bunch. They had a look at the early output from formatting engine [Z] and decided the driver's manuals couldn't be done in XML because there would be compromises in the layout and the manuals need to look pretty. Never mind that they went to 42 languages and, by not using XML, added a 70% cost to every single manual produced. PageMaker it is.

And no, they didn't want the accessory catalogues in XML either, for similar reasons.

2.16. Those Were the Days

I was tasked with writing an XSD for a company that wanted to move away from this ancient COBOL monster with heavily typed data ensuring that they'd never go past a couple of Megabytes, created in a time when every byte cost a fortune.

Yet, it quickly became apparent that they chose XSDs because the data described by them can be typed… just like before. In the end, I delivered a schema that rather faithfully reproduced that old COBOL monster but with no additional value.

This is not about a legacy system as much as it is about a legacy mindset. I could have avoided the pain simply by asking "what do you want to achieve with your new XSD?""

2.17. Latin 1 and Entitites

Publisher X stores all of their documents - hundreds of thousands of them - in an old, heavily customised Oracle DB. They've built a document management layer on top of the thing but they don't really have proper versioning on either documents or the DTDs that govern them. They still run OmniMark scripts to do validation, some light processing and the like.

Oh, and it's all in Latin 1, with about 250 or so general entities. This includes the Omnimark scripts.

Long before I first knew the company, they've been wanting to move to UTF-8 but the management and various project decisions have consistently held them back. They've bought companies and merged entire content libraries with their own, and it's all been converted to… Latin 1. On an office whiteboard, there is a

counter for the number of days since the last encoding-related error that never wanders far from "1"; in other words, there are issues almost daily, ranging from web pages that refuse to format to documents that refuse to load to that ancient Oracle DB.

Yet, the company handles this technical debt mostly by ignoring it. They've built new presentation systems and increased their print offering, and now they're about to add a MarkLogic DB on the side. MarkLogic will mirror Oracle and will be used for analysis to begin with; in time, it might replace the Oracle monster.

One assumes that they'll somehow incorporate Latin 1 and DTDs in it, though, since while they're always saying that they MUST move to Unicode and UTF-8, everything else always comes first.

3. Opposing Views

3.1. Sometimes SGML Is What You Want

I was part of a consulting effort to deliver a new system to an aerospace company and among the first in to take a long hard look at their current information set, most of which was early S1000D SGML. This was close to 20 years ago.

After careful consideration, I realised that them using SGML was just fine; no need to author in XML, no need to write an XML DTD or use the XML version of the standard (S1000D back then did not yet have a proper XML DTD or schema, or proper support for it) and migrate the content. All we needed was a modern approach to authoring, managing, and storing the SGML, and a bunch of conversions to XML and other formats *when publishing*. Their suppliers all used SGML, and they delivered to companies and organisations that all used SGML. No need to change any of this; it would be costly and unnecessary, and we'd have to convert back to SGML anyway.

Not to mention that the client was fine with SGML, too. The issue was not SGML, the issue was an aging system that couldn't keep up.

My employer was in the process of merging with another consultancy, however, and the players all needed to score. They both liked Documentum and were partners with them, so there were political advantages to using it. For the business, that is; never mind the client. So they decided they need XML because XML is modern and new and hip, and it will brush your teeth in the evening and wake you up with coffee the next day. And besides, Documentum wouldn't do SGML.

The process dragged on and on, and I was eventually pulled from the project. I left the company not too long after. The thing they ended up building was an absolute disaster that was eventually settled in court while the client bought a competitor's product instead.

3.2. Sometimes Word Is What You Want

Most legal publisher use Word in one way or another. I worked with one that published Precedents document templates for lawyers in Word format, in spite of the Word files actually being produced in XML, added some intricate and very clever tagging, and then converted to Word. As an XML geek, I was dead against this, of course. Word seemed like an unnecessary extra layer.

But when I put aside my pride and supposed XML competence, and really started thinking about the process, I realised how wrong I was: the end users are non-technical lawyers and it would be a lot more complex (technically and politically) to get them to use an XML tool, regardless of all the cool things we could do the make life easier for them.

Sometimes MS Word is what you really want.

4. Things We Found among the Ashes

It goes without saying. First rule of thumb: be pragmatic. Not only do you have to make a living. Sometimes you need to swallow your pride and go for the poor solution. Trying to push the purist solution you know would work, might make you sleep better. But at the end of the day, do whatever it takes to make things work... if if you know deep down it is stupid.

Drink a lot of coffee... or beer. Companies do have organisational charts and official guidelines. But it is the unwritten rules in the workplace and the personal connections that will tell you so much more. You gain valuable knowledge and better grasp any political sensitivities by listening to coworkers in an informal context.

Don't ever engage in religious discussions. When “forces” at your customer's are convinced that their technology is the one to use, *use it*. You can bring all the arguments you want to the table. Objective criteria won't be sufficient to convert the religious inspired.

Choose your battles, even if you risk frustration over your work.

But do engage in discussions about licenses. Too many companies have an open source software policy because they think it is the same as “free software”.

Valuable time is lost in projects working around restrictions in basic freeware. For a small fee one can often buy a lot of robustness, functionality, performance, etc. But also, someone has to develop and maintain the tools you use. One should always consider the long term validity of what you build and bring.

For the long-term benefit of your customer, this is a battle worth fighting.

Don't bargain on your price. Failure is inherent if you bridge a large gap in price discussions.

You sell value, not hours. If the customer thinks you're valued lower than you do yourself and you agree on a compromise higher than theirs and lower than yours, you are about to be hired for the wrong job. They will think they are pay-

ing you too much for what you do, and you will be frustrated because you know they make you do things for less than they should be paying.

Get your responsibilities straight from the start.

Customers often don't realize what exactly they are hiring you for. A quick development task at a fair price, will most likely lead to taking over an architect role for a longer time for the wrong price. Try to discover early what the roles are, and guide your boundaries

Organize yourself to be able to get out quickly by offering a solid long term self sustainability.

Develop and document whatever you do as if you won't come back tomorrow. You don't want these projects to continue to haunt you for the rest of your life.

And if they do haunt you... you've made sure you can get back in with a smile

Accept failure gracefully. You can only do the best you can do. Some projects simply fail. Because of you or despite of you.

We are all very proud people. Yet, there is no shame in admitting that one project or another has failed... and that you might have played a role in that failure. Maybe you did not push enough for a change, maybe you did not pay attention.

Maybe you (shudder) were wrong.

Sequence alignment in XSLT 3.0[1]

David J. Birnbaum
Department of Slavic Languages and Literatures, University of Pittsburgh (US)
<djbpitt@gmail.com>

Abstract

The Needleman Wunsch algorithm, which this year celebrates its quinquagenary anniversary, has been proven to produce an optimal global pairwise sequence alignment. Because this dynamic programming algorithm requires the progressive updating of mutually dependent variables, it poses challenges for functional programming paradigms like the one underlying XSLT. The present report explores these challenges and provides an implementation of the Needleman Wunsch algorithm in XSLT 3.0.

Keywords: sequence alignment, xslt

1. Introduction

1.1. Why sequence aligment matters

Sequence alignment is a way of identifying and modeling similarities and differences in sequences of items, and has proven insightful and productive for research in both the natural sciences (especially in biology and medicine, where it is applied to genetic sequences) and the humanities (especially in text-critical scholarship, where it is applied to sequences of words in variant versions of a text). In textual scholarship, which is the domain in which the present report was developed, sequence alignment assists the philologist in identifying locations where manuscript witnesses agree and where they disagree.[2] These agreements and disagreements, in turn, provide evidence about probable (or, at least, candidate) moments of shared transmission among textual witnesses, and thus serve as evidence to construct and support a philological argument about the history of a text.[3]

[1]I am grateful to Ronald Haentjens Dekker for comments and suggestions.
[2]*Witness*, sometimes expanded as *manuscript witness*, is a technical term in text-critical scholarship for a manuscript that provides evidence of the history of a text.
[3]For an introduction to the evaluation of shared and divergent readings as a component of textual criticism see Trovato 2014.

1.2. Biological and textual alignment

Insofar as biomedical research enjoys a larger scientific community and richer funding resources than textual humanities scholarship, it is not surprising that the literature about sequence alignment, and the science reported in that literature, is quantitatively greater in the natural sciences than in the humanities. Furthermore, insofar as all sequence alignment is similar in certain mathematical ways, it is both necessary and appropriate for textual scholars to seek opportunities to adapt biomedical methods for their own purposes. For those reasons, the present report, although motivated by text-critical research, focuses on a method first proposed in a biological context and later also applied in philology.

This report does not take account of differences in the size and scale of biological and philological data, but it is nonetheless the case that alignment tasks in biomedical contexts, on the one hand, and in textual contexts, on the other, typically differ at least in the following ways:

- Genetic alignment may operate at sequence lengths involving entire chromosomes or entire genomes, which are orders of magnitude larger than the largest real-world textual alignment tasks.
- Genetic alignment operates with a vocabulary of four words (nucleotide bases, although alignment may also be performed on codons), while textual alignment often involves a vocabulary of hundreds or thousands of different words.

The preceding systematic differences in size and scale invite questions about whether the different shape of the source data in the two domains might invite different methods. Especially in the case of heuristic approaches that are not guaranteed to produce an optimal solution, is it possible that compromises required to make data at large scale computationally tractable might profitably be avoided in domains involving data at a substantially smaller scale? Although the present report does not engage with this question, it remains part of the context within which solutions to alignment tasks in different disciplines ultimately should be assessed.

1.3. Global pairwise alignment

The following two distinctions—not between biological and textual alignment, but within both domains—are also relevant to the present report:

- Both genetic and textual alignment tasks can be divided into *global* and *local* alignment. The goal of global alignment is to find the best alignment of *all items in the entire sequences*. In textual scholarship this is often called *collation* (cf. e.g., Frankenstein variorum reader). The goal of local alignment is to find moments where *subsequences* correspond, without attempting to optimize the alignment of the entire sequences. A common textological application of local

alignment is *text reuse*, e.g., finding moments where Dante quotes or paraphrases Ovid (cf. Van Peteghem 2015, Intertextual Dante).

- Both genetic and textual alignment tasks may involve *pairwise alignment* or *multiple alignment*. Pairwise alignment refers to the alignment of two sequences; multiple alignment refers to the alignment of more than two sequences. In textual scholarship multiple alignment is often called *multiple-witness alignment*.

The Needleman Wunsch algorithm described and implemented below has been proven to identify all optimal global pairwise alignments of two sequences, and it is especially well suited to alignment tasks where the two texts are of comparable size and are substantially similar to each other. The present report does not address either local alignment or multiple (witness) alignment.

1.4. Overview of this report

This report begins by introducing the use of dynamic programming methods in the Needleman Wunsch algorithm to ascertain all optimal global alignments of two sequences. It then identifies challenges to implementing this algorithm in XSLT and discusses those challenges in the context of developing such an implementation. Original code discussed in this report is available at https://github.com/djbpitt/xstuff/tree/master/nw.

It should be noted that the goal of this report, and the code underlying it, is to explore global pairwise sequence alignment in an XSLT environment. For that reason, it is not intended that this code function as a stand-alone end-user textual collation tool. There are two reasons for specifying the goals and non-goals of the present report in this way:

- Textual collation as a philological method involves more than just alignment. For example, the Gothenburg model of textual collation, which has been implemented in the CollateX [CollateX] and Juxta [Juxta] tools, expresses the collation process as a five-step pipeline, within which alignment serves as the third step. [Gothenburg model]
- Real-world textual alignment tasks often involve more than two witnesses, that is, they involve multiple-witness, rather than pairwise, alignment. While some approaches to multiple-witness alignment are implemented as a progressive or iterative application of pairwise alignment, these methods are subject to order effects. Ultimately, order-independent multiple-witness alignment is an NP hard problem with which the present report does not seek to engage.[4]

[4]Multiple sequence alignment (Wikipedia) provides an overview of multiple sequence alignment, the term in bioinformatics for what philologists refer to as multiple-witness alignment.

2. About sequence alignment

2.1. Alignment and scoring

An optimal alignment can be defined as an alignment that yields the best *score*, where the researcher is responsible for identifying an appropriate *scoring method*. Relationships involving individual aligned items from a pair of sequences can be categorized as belonging to three possible types for scoring purposes:

- Items from both sequences are aligned and are the same. This is called a *match*. If the two entire sequences are identical, all item-level alignments are matches.
- Items from both sequences are aligned but are different. This is called a *mismatch*. Mismatches may arise in situations where they are sandwiched between matches. For example, given the input sequences "The brown koala" and "The gray koala", after aligning the words "The" and "koala" in the two sequences (both alignments are matches), the color words sandwiched between them form an aligned mismatch.
- An item in one sequence has no corresponding item in the other sequence. This is called a *gap* or an *indel* (because it can be interpreted as either an *in*sertion in one sequence or a *del*etion from the other). Gaps are inevitable where the sequences are of different lengths, so that, for example. given "The gray koala" and "The koala", the item "gray" in the first sequence corresponds to a gap in the second. Gaps may also occur with sequences of the same length; for example, if we align "The brown koala lives in Australia" with "The koala lives in South Australia", both sequences contain six words, but the most natural alignment, with a length of seven items and one gap in each sequence, is:

Table 1. Alignment example with gaps

The	brown	koala	lives	in		Australia
The		koala	lives	in	South	Australia

A common scoring method is to assign a value of `1` to matches, `-1` to mismatches, and `-1` to gaps. These values prefer alignments with as many matches as possible, and with as few mismatches and gaps as possible. But alternative scoring methods might, for example, assign a greater penalty to gaps than to mismatches, or might assign different penalties to new gaps than to continuations of existing gaps (this is called an *affine* gap penalty).

The scoring method determines what will be identified as an optimal alignment for a circular reason: optimal in this context is defined as the alignment with the best score. This means that the selection of an appropriate scoring method during philological alignment should reflect the researcher's theory of the types

of correspondences and non-correspondences that are meaningful for identifying textual moments to be compared. In the examples below we have assigned a score of `1` for matches, `-1` for mistmatches, and `-2` for gaps. This scoring system minimizes gaps.

2.2. Sequence alignment algorithms

A naïve, brute-force approach to sequence alignment would construct all possible alignments, score them, and select the ones with the best scores. This method has exponential complexity, which makes it unrealistic even for relatively small real-world alignment tasks. [Bellman 1954 2] Alternatives must therefore reduce the computational complexity, ideally by reducing the search space to exclude from consideration in advance all alignments that *cannot* be optimal. Where this is not possible, a *heuristic* method excludes from consideration in advance alignments that are *unlikely* to be optimal. Heuristic methods entail a risk of inadvertently excluding an optimal alignment, but in the case of some computationally complex problems, a heuristic approach may be the only realistic way of reducing the complexity sufficiently to make the problem tractable.

In the case of global pairwise alignment, the Needleman Wunsch algorithm, described below, has been proven always to produce an optimal alignment, according to whatever definition of optimal the chosen scoring method instantiates. Needleman Wunsch is an implementation of *dynamic programming*, and in the following two sections we first describe dynamic programming as a paradigm and then explain how it is employed in the Needleman Wunsch algorithm. These explanations are preparatory to exploring the complications that dynamic programming, both in general and in the context of Needleman Wunsch, pose for XSLT and how they can be resolved.

3. Dynamic programming and the Needleman Wunsch algorithm

3.1. Dynamic programming

Dynamic programming, a paradigm developed by Richard Bellman at the Rand Corporation in the early 1950s, makes it possible to express complex computational tasks as a combination of smaller, more tractable, overlapping ones.[5] A commonly cited example of a task that is amenable to dynamic programming is the computation of a Fibonacci number. Insofar as every Fibonacci number beyond the first two can be expressed as a function of the two immediately preceding Fibonacci numbers, a naïve top-down approach to computing the value of the *nth* Fibonacci number would start with *n* and compute the two preceding val-

[5]For more information about dynamic programming see Bellman 1952 and Bellman 1954.

ues. This requires computing all of their preceding values, which requires computing their preceding values, etc., which ultimately leads to computing the same values repeatedly. A dynamic bottom-up computation, on the other hand, would calculate each smaller number only once and then use those values to move up to larger numbers.[6]

Sequence alignment meets the two requirements for a problem to be amenable to dynamic programming.[Grimson and Guttag] First, it satisfies *optimal substructure*, which means that an optimal solution to a problem can be reached by determining optimal solutions to its subproblems. Second, it satisfies *overlapping subproblems*, where *overlapping* means "common" or "shared", that is, that the same subproblems recur repeatedly. In the Fibonacci example above, the computation of a higher Fibonacci number depends on the computation of the two preceding numbers (optimal substructure), and the same preceding numbers are used repeatedly in a top-down solution (overlapping subproblems). In the case of pairwise sequence alignment, the Needleman Wunsch algorithm, discussed below, observes both optimal substructure (an optimal alignment is found by finding optimal alignments of subsequences) and overlapping subproblems (the same properties of these subsequences are reused to solve multiple subproblems).

3.2. The Needleman Wunsch algorithm

The history of the Needleman Wunsch algorithm is described by Boes as follows:

> *We will begin with the scoring system most commonly used when introducing the Needleman-Wunsch algorithm: substitution scores for matched residues and linear gap penalties. Although Needleman and Wunsch already discussed this scoring system in their 1970 article [NW70], the form in which it is now most commonly presented is due to Gotoh [Got82] (who is also responsible for the affine gap penalties version of the algorithm). An alignment algorithm very similar to Needleman-Wunsch, but developed for speech recognition, was also independently described by Vintsyuk in 1968 [Vin68]. Another early author interested in the subject is Sellers [Sel74], who described in 1974 an alignment algorithm minimizing sequence distance rather than maximizing sequence similarity; however Smith and Waterman (two authors famous for the algorithm bearing their name) proved in 1981 that both procedures are equivalent [SWF81]. Therefore it is clear that there are many classic papers, often a bit old, describing Needleman-Wunsch and its variants using different mathematical notations. (Boes 2014 14; pointers are to Needleman and Wunsch 1970, Gotoh 1982, Vintsyuk 1968, Sellers 1974, and Smith et al. 1981)*

[6]The implementation of dynamic programming according to a bottom-up organization is called *tabulation*. A top-down dynamic approach would perform all of the recursive computation at the beginning, but *memoize* (that is, store and index) the sub-calculations, so that they could be looked up and reused, without having to be recomputed, when needed at lower levels.

Boes further explains that Needleman Wunsch "is an *optimal* algorithm, which means that it produces the best possible solution with respect to the chosen scoring system. There [exist] also non-optimal alignment algorithms, most notably the heuristic methods …" [Boes 2014 13] "Non-optimal" here means not that the method is *incapable* of arriving at an optimal solution, but that it is *not guaranteed* to do so.

Performing alignment according to the Needleman Wunsch dynamic programming algorithm entails the following steps:[7]

1. Construct a grid with one of the sequences to be aligned along the top, labeling the columns, and the other along the left, labeling the rows.
2. Determine a scoring system. Here we score matches as `1`, mismatches as `-1`, and gaps as `-2`.
3. Insert a row at the top, below the labels, with sequential numbers reflecting consecutive multiples of the gap score. For example, if the gap score value is `-2`, the cell values would be `0`, `-2`, `-4`, etc. Starting from the `0`, assign similar values to a column inserted on the left, after the row labels. By this point the grid should look like:

Table 2. Initial grid for Needleman Wunsch

		k	o	a	l	a
	0	-2	-4	-6	-8	-10
c	-2					
o	-4					
l	-6					
a	-8					

4. Starting in the upper left of the table body, where the first items of the two sequences intersect, and proceeding across each row in turn, from top to bottom, write a value into each cell. That value should be the highest of the following three candidate values:
   - The value in the cell immediately above plus the gap score.
   - The value in the cell immediately to the left plus the gap score.
   - The value in the cell immediately diagonally above to the left plus the match or mismatch score, depending on whether the intersecting sequence items constitute a match or a mismatch.

   For example, the first cell is the intersection of the "k" at the top with "c" at the left, which is a mismatch, since they are different. The cell immediately

[7] For a more detailed tutorial introduction see Global alignment.

above has a value of `-2`, which, when augmented by the gap score, yields a value of `-4`. The same is true of the cell immediately to the left. The cell diagonally above and to the left has a value of `0`, which, when combined with the mismatch score, yields a value of `-1`. Since that is the highest value, write it into the cell. Proceed similarly across the first row, then traverse the second row from left to right, etc., ending in the lower right. The completed grid should look like:

Table 3. Completed grid for Needleman Wunsch

		k	o	a	l	a
	0	-2	-4	-6	-8	-10
c	-2	-1	-3	-5	-7	-9
o	-4	-3	0	-2	-4	-6
l	-6	-5	-2	-1	-1	-3
a	-8	-7	-4	-1	-2	0

We fill the cells in the specified order because each cell depends on two values from the row above (the cell immediately above and the one diagonally above and to the left) and the preceding cell of the same row. Filling in the cells from left to right and top to bottom ensures that these values will be available when needed. For reasons discussed below, these ordering dependencies pose a challenge for an XSLT implementation.

5. Starting in the lower right corner, trace back through the sources that determined the score of each cell. For example, the `0` value in the lower right inherited from the upper diagonal left because the `-1` that was there plus the match score of `1` yielded a `0`, and that value was higher than the scores coming from the cell immediately above (`-3` plus the gap score of `-2` yields `-5`) or to immediately to the left (`-2` plus the gap score of `-2` yields `-4`). In the following image we have 1) added arrows indicating the source of each value entered into the grid and 2) shaded match cells green and mismatch cells pink:

		k	o	a	l	a
	0	→ **-2**	→ **-4**	→ **-6**	→ **-8**	→ **-10**
c	↓ **-2**	↘ -1	↘ -3	↘ -5	↘ -7	↘ -9
o	↓ **-4**	↘ -3	↘ 0	→ -2	→ -4	→ -6
l	↓ **-6**	↘ -5	↓ -2	↘ -1	↘ -1	→ -3
a	↓ **-8**	↘ -7	↓ -4	↘ -1	↘ -2	↘ 0

Figure 1. Completed alignment grid

6. At each step along this traceback path, starting from the lower right, if the step is diagonal and up, align one item from the end of each sequence. If the step is to the left, align an item from the sequence at the top with a gap (that is, do not select an item from the sequence at the left). If the step is up, align an item from the sequence at the left with a gap. In case of ties, the choices with the highest value are all optimal and can be pursued as alternatives. In the present case, this process produces the following single optimal alignment:

koala	k	o	a	l	a
cola	c	o		l	a

Figure 2. Alignment table

4. The challenges of dynamic programming and XSLT

XSLT, at least before version 3.0, plays poorly with dynamic programming because each step in a dynamic programming algorithm depends on values calculated at preceding steps. Functional programming of the sort supported by XSLT `<xsl:for-each>` does not have access to these incremental values; if we try to run `<xsl:for-each>` over all of the cells and populate them according to the values before and above them, those neighboring values will be the values in place initially, that is, null. The reason is that `<xsl:for-each>` is not an iterative instruction: it *orders the output* according to the input sequence, but it does not necessarily *perform the computation* in that order. This is a feature because it means that such instructions can be parallelized, since no step is dependent on the output of any other step. But it also means that populating the Needleman Wunsch grid in XSLT requires an alternative to `<xsl:for-each>`.

Tennison draws our attention to this issue in her XSLT 2.0 implementations of a dynamic programming algorithm to calculate Levenshtein distance (Tennison 2007a, Tennison 2007b), and with respect to constructing the grid, the algorithms for Levenshtein and Needleman Wunsch are analogous. The principal difference is that Levenshtein cares only about the value of the lower right cell, and there-

fore does not require the traceback steps that Needleman Wunsch uses to perform the alignment of actual sequence items.

4.1. Why recursion breaks

The traditional way to mimic updating a variable incrementally in XSLT is with recursion, cycling the newly updated value into each recursive call. The challenge to this approach is that deep recursion can consume enough memory to overflow the available stack space and crash the operation. XSLT processors can work around this limitation with *tail call optimization*, which enables the processor to reduce the consumption of stack space by recognizing when stack values do not have to be preserved. Tail call optimization is finicky, however, first because not all XSLT implementations support it, second because functions have to be written in a particular way to make it possible, and third because some operations that can be understood as tail recursive may not look that way to the processor, and may therefore fail to be optimized.

The important insight with respect to recursion in Tennison's second engagement with the Levenshtein problem (Tennison 2007b) is that it is possible to construct the grid values for Levenshtein (and therefore also for Needleman Wunsch) without recurring on every cell. By writing values into the grid on the anti-diagonal (diagonals that run from bottom left to top right), instead of across each row in turn, as is traditional, Tennison is able to calculate all values on an individual anti-diagonal at the same time, since the cells on any single anti-diagonal have no mutual dependencies.[8] The absence of dependencies within an anti-diagonal means that Tennison can use `<xsl:for-each>`, instead of recursion, to compute all of the values within each anti-diagonal, and recur only as she moves to a new anti-diagonal. The computational complexity of populating the grid remains *O(mn)* (that is, essentially quadratic), since it is still necessary to calculate values for each cell individually, and the total number of cells is the product of the lengths of the two sequences, but Tennison's implementation reduces the recursion from the number of cells to the number of anti-diagonals, which is *n + m - 1*, that is, linear with respect to the total number of items in the two sequences. This implementation also reduces the storage complexity; because each anti-diagonal depends only on the two immediately preceding ones, the recursive steps do not have to pass forward the entire state of the grid.

The potential improvement in computational efficiency that may result from parallelization in an implementation "on the diagonal" was identified initially by Muraoka 1971 (160), who used the term "wave front" to describe the sequential

[8]Not only are there no mutual dependencies within an anti-diagonal, but all of the information needed to process an entire anti-diagonal is available simultaneously from only the two preceding anti-diagonals, without any dependency on earlier ones. This property contributes to the scalability of our implementation in ways that will be discussed below.

processing of anti-diagonals, and then explored further by Wang 2002 (8) and Naveed et al. 2005 (3–4).[9] Although these earlier researchers had previously reported that items on the anti-diagonal could be processed in parallel, it was Tennison who recognized that this observation could also be used to reduce the depth of recursion in XSLT.

4.2. Iteration to the rescue

Tennison's anti-diagonal implementation reduces the depth of recursion, but does not eliminate recursion entirely: because the values in each anti-diagonal continue to depend on the values in the two immediately preceding anti-diagonals, it nonetheless requires recursion on each new anti-diagonal. The reduction in the depth of recursion from quadratic to linear scales impressively; for example, with two 20-item sequences and 400 cells, the traditional method would have recurred 400 times, while the anti-diagonal method makes only 39 recursive function calls. In XSLT 3.0, however, it is possible to use `<xsl:iterate>` to avoid recursive coding entirely:

> *[xsl:iterate] is similar to xsl:for-each, except that the items in the input sequence are processed sequentially, and after processing each item in the input sequence it is possible to set parameters for use in the next iteration. It can therefore be used to solve problems that in XSLT 2.0 require recursive functions or templates. (Saxon xsl:iterate)*

The use of `<xsl:iterate>`, which was not part of the XSLT 2.0 that was available to Tennison in 2007, in place of the recursion that she was forced to retain, thus observes her wise recommendation to "try to iterate rather than recurse whenever you can" (Tennison 2007b).

4.3. Processing the anti-diagonal

The classic description of Needleman Wunsch differs from Levenshtein by requiring that the entire grid be available at the end of its construction so that it can be traversed backwards to perform the actual item alignment (Levenshtein cares only about the final value), but the two algorithms agree in the fact that cells on each consecutive anti-diagonal can be constructed using information from only the two immediately preceding anti-diagonals. Within our `<xsl:iterate>` instruction we return these two preceding anti-diagonals as parameters called `$ult` (immediately preceding) and `$penult` (preceding `$ult`), promoting the previous `$ult` to the new `$penult` on each iteration and adding the current anti-

[9] Wang 2002 also uses the term "wave front" (two words), as introduced by Muraoka; Maleki et al. 2014 modify this as "wavefront" and introduce the term "stage" to refer to the individual anti-diagonals.

diagonal as the new `$ult`. We attempt to improve the retrieval of these preceding cells while computing new values by using `<xsl:key>` with a composite `@use` attribute that indexes the two anti-diagonals that constitute the search space according to the `@row` and `@col` attribute values of each cell. At a minimum, each new cell holds information, in attributes, about its row, column, and score (all used to compute the values of subsequent cells) and the prior cell that was used to determine that score (diagonal, up, or left; used for the backward tracing of the path once construction has been completed); we also store some additional values, which we discuss below.

It is possible for more than one neighboring cell to tie for highest value, and because the task that motivated this development required only *an* optimal alignment, and not *all* such alignments, we record only one optimal path to each cell, resolving ties by arbitrarily favoring diagonal, then left, and only then upper sources. There is, however, nothing about the method that would prohibit recording and later processing multiple paths, and thus identifying all optimal alignments.

In the Needleman Wunsch (and also Levenshtein) context, then, all values on the same anti-diagonal can be calculated in parallel, and Tennison's use of `<xsl:for-each>` in her improved code in Tennison 2007b to process the anti-diagonal is compatible with this observation because `<xsl:for-each>` can be parallelized. Whether it *is* executed in parallel, however, is often unpredictable, since standard XSLT 3.0 does not give the programmer explicit control over processes or threads in the same way as other languages (cf. Python's `multiprocessing` module). However, Saxon EE (although not PE or HE) provides a custom `@saxon:threads` attribute that allows the developer to specify that an `<xsl:for-each>` element should be processed in parallel. The documentation explains that:

> *This attribute may be set on the xsl:for-each instruction. The value must be an integer. When this attribute is used with Saxon-EE, the items selected by the select expression of the instruction are processed in parallel, using the specified number of threads. (Saxon saxon:threads)*

The Saxon documentation adds, however, that:

> *Processing using multiple threads can take advantage of multi-core CPUs. However, there is an overhead, in that the results of processing each item in the input need to be buffered. The overhead of coordinating multiple threads is proportionally higher if the per-item processing cost is low, while the overhead of buffering is proportionally higher if the amount of data produced when each item is processed is high. Multi-threading therefore works best when the body of the xsl:for-each instruction performs a large amount of computation but produces a small amount of output. (Saxon saxon:threads)*

The computation of a cell score produces a small amount of output, but it also involves only a small amount of computation (compared to read/write memory

operations). As we discuss below, in this case parallelization did not lead to reliably improved performance.

4.4. Save yourself a trip … and some space

The process of constructing the scoring grid for Needleman Wunsch on the anti-diagonal is identical to that of constructing the grid for Levenshtein, but, as was noted above, the key difference is what happens next: Levenshtein cares only about the value of the lower right cell, and therefore does not need to walk back through the grid the way Needleman Wunsch does to align the actual sequences. This means that an anti-diagonal implementation for a Levenshtein distance calculation can throw away each anti-diagonal once it is no longer needed, and the single-cell anti-diagonal at the lower right will contain the one piece of information the function is expected to return: the distance between the two sequences. An implementation of Needleman Wunsch according to the classic description of the method, however, cannot economize on space in this way, which means that although Needleman Wunsch and Levenshtein have comparable *computational* complexity, classic Needleman Wunsch has quadratic *storage* complexity because it preserves and passes along the entire grid, while Tennison's anti-diagonal Levenshtein implementation has linear storage complexity because it throws away anti-diagonals as soon as it no longer needs them, and the length of the diagonal is linear with respect to the lengths of the input sequences.

The storage requirements of Needleman Wunsch are quadratic, however, only as long as the entire grid must be preserved for backward traversal at the end of the construction process, and the only information needed for that traversal is the direction (diagonal, left, up) of the optimal path steps. At each step along that traversal we do not need to know the score and we do not need the row and column labels. This means that we can avoid the backward traversal of the grid entirely if we write the cumulative full path to each cell into the cell alongside its score, instead just the source of the most recent path step, so that the lower right cell will already contain information about the full path that led to it. We can then use those directional path steps to construct an alignment table on the basis of the original sequences, without further reference to the grid. Avoiding the backwards trip through the grid after its completion comes at the expense of writing full path information into every cell during the construction of the grid, which entails extra computation and storage, even though we will ultimately use this information only from the one lower right cell for the final alignment. In compensation for storing that additional information in the cells, though, we no longer need to pass the entire cumulative grid through the iterations, so the additional paths must be stored only for the three-anti-diagonal life cycle of each cell. The section below documents the improvement this produces with respect to both execution time and memory requirements.

4.5. Performance

We implemented the method described above using vanilla XSLT 3.0 of the sort that can be executed in Saxon HE without any proprietary extensions. As a small optimization, because each cell is used an average of three times to compute new cell values (once each as diagonal, left, and upper), and the left and upper behaviors are the same (sum the score of the cell and the gap penalty), we perform that sum operation just once and store it when the cell is created, instead of computing it twice on the two occasions when it is used.[10]

We then revised the code for Saxon EE with two further types of modification:

- We use the `@saxon:threads` attribute with an arbitrary value of `10` on our `<xsl:for-each>` elements. This ensures that the body of the `<xsl:for-each>` element will be parallelized, although 1) regardless of the value of the `@saxon:threads` attribute, the number of computations that can actually be performed simultaneously depends on the number of cores provided by the CPU and on other demands on CPU resources, and 2) parallelization improves performance only when the benefit of parallel executation is greater than the overhead of managing it. In practice, in this case the use of `@saxon:threads` produced no reliable improvement in performance; see the discussion below.
- We use schema-aware processing with type annotations (using the `@type` attribute) on the temporary `<cell>` attributes that are used in computation, which means principally the `@row` and `@col` (column) attributes, which we type as `xs:integer`. By default attributes on elements that do not undergo validation are typed as `xs:untypedAtomic`, and without our explicit typing we had to convert them explicitly to numerical values on some occasions when we needed to operate with them. Typing them as they are created and preserving the typing removes the need to cast them explicitly as numbers later.[11] The reduction in processing that results from not having to perform explicit casting must be balanced against the overhead of performing schema validation (or, perhaps more accurately, type validation).

To explore the performance and scalability of the implementations we conducted word-level alignment on portions of Chapter 1 of the 1859 and 1860 (first and second) editions of Charles Darwin's *On the origin of species*, which we copied from *Darwin online* (http://darwin-online.org.uk/). We chose these editions to simplify the simulation of natural testing circumstances across different quantities of

[10]See Space–time tradeoff.

[11]For example, we use keys to retrieve cells by row and column number, the values of which we compute, and the type of the value used to retrieve an item with a key must match the type of the value used to index it originally (Kay 2008 813). Typing the row and column number as integers when they are created removes the need to cast them as numerical types for query and retrieval.

text. Specifically, these chapters have the same number of paragraphs, and the paragraphs observe the same order with respect to their overall content, although there are small differences in wording within the paragraphs. (This is not the case consistently with later editions, which deviate more substantially from one another.) This means that we can scale the quantity of text while always working with a natural comparison by specifying the number of paragraphs (rather than the number of words) to align. We ran the Saxon EE (v. 9.9.1.5J) and HE (v. 9.9.1.4J) transformations from the command line with the following commands, respectively:

- `java -Xms24G -Xmx24G -jar /path/saxon9ee.jar -sa -it -o:/dev/null -repeat:10 nw_ee.xsl`
- `java -Xms24G -Xmx24G -jar /path/saxon9he.jar -it -o:/dev/null -repeat:10 nw_he.xsl`

These instructions make 24G of RAM available to Java and cause Saxon to perform the specified transformation 10 times and report the average execution time of the last 6 runs. The testing platform was a mid-2018 MacBook Pro with a 2.9 GHz Intel Core i9 processor (6 physical and 12 logical cores) and 32 GB 2400 MHz DDR4 RAM. Times are in milliseconds. The "N/A" values in the table below reflect processing that crashed with Java memory errors; see below for discussion. The table below shows the EE and HE processing time (total and ms per token); it reports on the time EE requires to output not just the alignment table, but also the full alignment grid; and it compares the EE and HE times directly.

Table 4. Comparison of EE and HE performance

Tokens				EE					HE		
Paras	**1859 tokens**	**1860 tokens**	**total tokens**	**time**	**ms per token**	**time with grid**	**ms per token with grid**	**grid cost**	**time**	**ms per token**	**EE vs HE**
1	193	194	387	567	1.5	880	2.3	155%	669	1.7	84.8%
2	232	233	465	751	1.6	1221	2.6	163%	641	1.4	117.1%
3	679	683	1362	4740	3.5	11898	8.7	251%	5498	4.0	86.2%
4	772	777	1549	6464	4.2	14903	9.6	231%	6627	4.3	97.5%
5	810	815	1625	7082	4.4	15031	9.2	212%	7389	4.5	95.8%
6	942	947	1889	9573	5.1	20599	10.9	215%	9963	5.3	96.1%
7	1187	1193	2380	15437	6.5	N/A	N/A	N/A	17501	7.4	88.2%
8	1363	1369	2732	22007	8.1	N/A	N/A	N/A	26263	9.6	83.8%
9	1583	1589	3172	29636	9.3	N/A	N/A	N/A	36266	11.4	81.7%
10	1676	1682	3358	32570	9.7	N/A	N/A	N/A	41517	12.4	78.5%
11	1908	1912	3820	44568	11.7	N/A	N/A	N/A	54075	14.2	82.4%
12	2233	2239	4472	63820	14.3	N/A	N/A	N/A	67932	15.2	93.9%
13	2659	2663	5322	96120	18.1	N/A	N/A	N/A	98069	18.4	98.0%

Tokens				EE					HE		
Paras	**1859 tokens**	**1860 tokens**	**total tokens**	**time**	**ms per token**	**time with grid**	**ms per token with grid**	**grid cost**	**time**	**ms per token**	**EE vs HE**
14	2966	2974	5940	120520	20.3	N/A	N/A	N/A	124798	21.0	96.6%
15	3147	3126	6273	134375	21.4	N/A	N/A	N/A	138405	22.1	97.1%

The chart below compares EE and HE performance.

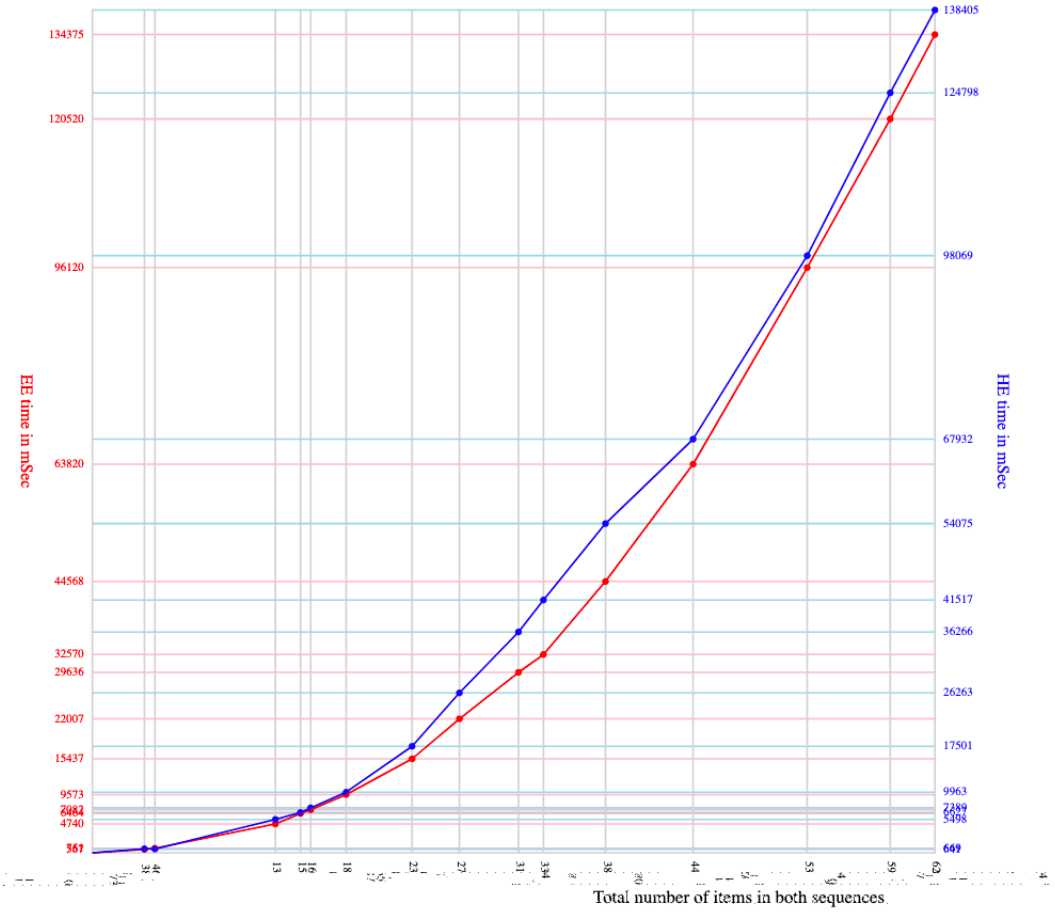


Figure 3. Performance with Saxon EE and HE

Except with a very small number of tokens, EE runs the same operation as HE more quickly, but the effect of the relative difference in execution time diminishes as the volume of input grows. We had anticipated that there would be at least a small improvement in performance because EE let us parallelize `<xsl:for-each>` operations, but when we tested the parallelization with thread counts ranging from 1 to 10, the results were small, inconsistent, and contradictory, which led us to suspect that the better performance by EE was because it incorporates more sophisticated optimization in general, and not specifically because of our use of

`@saxon:threads`. In the chart below, the difference (across 1 to 10 threads) between best and worst performance is never greater than 11%, and it is neither uniformly monotonic nor consistent across different text quantities. The number to the left is the number of paragraphs, the percentage to the right is the difference between the best and worst performance, and the sparkline, from left to right, records the direction and relative degree of change in the timing with 1 to 10 threads:[12]

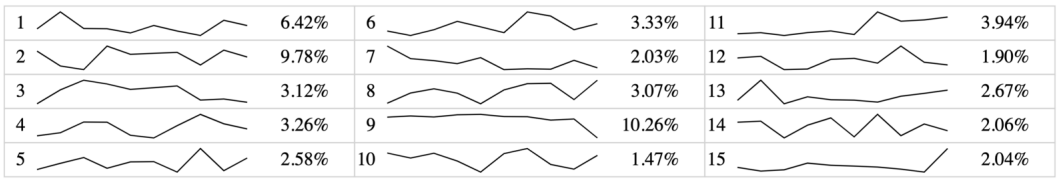


Figure 4. Effect of threading `<xsl:for-each>` on total execution time

If we recall that parallelization of the `<xsl:for-each>` instances in this project satisfies the "small amount of output" condition for optimal use of the `@saxon:threads` attribute, but not the "large amount of computation" one, it may be that this particular computation might be considered *embarrassingly unparallel*.[13]

The fact that the storage requirement scales linearly (as long as we do not attempt to maintain the entire grid) means that it is possible to align long sequences without overflowing the available memory, but the quadratic execution time means that the alignment of long sequences is nonetheless not well suited for real-time interactive processing.[14] If we do attempt to maintain the entire grid, which grows quadratically, the poor scaling, which is primarily an inconvenience with respect to processing time, quickly turns fatal with respect to storage. When asked to compose and maintain the entire grid (instead of just three anti-diagonals needed to compute the alignment), Saxon EE eventually crashed with a Java

[12]Tests were performed with the same settings as above: we processed each combination of threads (1 to 10) and paragraphs (1 to 15) 10 times, and Saxon EE reported the average of the last 6 iterations.

[13]See Embarrassingly parallel.

[14]As a test of larger capacity, we aligned the entire first chapter of the 1859 and 1860 editions of *On the origin of species*. The 49 paragraphs of the 1859 and 1860 editions contain 11590 and 11632 word tokens, respectively. The total number of word tokens in the two editions is 23222, and there are 134814880 (1.3481488e8) cells in the complete grid. The alignment, using EE and the default Java memory allocation, reported real time of 64m43.159s, user time of 538m5.128s, and sys time of 13m25.910s. Real time is lower than user time plus sys time because of parallel execution.

With respect to storage, processing maintains only a constant three anti-diagonals at a time, and the length of an anti-diagonal is linear with respect to the sum of the lengths of the sequences being compared. The lengths of the full paths that are accumulated on the cells grow linearly with respect to the number of anti-diagonals, which also enjoys a linear relationship with the lengths of the two sequences being aligned. The number of cells on an anti-diagonal grows, levels off, and then shrinks linearly with respect to the number of tokens in the two sequences being compared; the first and last anti-diagonal each contain a single cell.

memory error, which a larger Java `-Xmx` parameter could forestall, but not prevent. If the entire grid is an output requirement with a large amount of data, then, it will have to be output in a way that does not require it to be stored in memory in its entirety. Fortunately, as this implementation demonstrates, aligning the sequences does not require simultaneous access to the entire grid.

5. Conclusions

The code underlying this report is available at https://github.com/djbpitt/xstuff/tree/master/nw, and has not been reproduced here. It is densely commented, and thus offers tutorial information about the method. Small exploratory stylesheets that were used to develop individual components of the code have been retained in a *scratch* subdirectory. Performance testing code and results are in the *performance* and *threads* subdirectories.

Tennison concludes her second, improved computation of Levenshtein distance by writing that:

> *I guess the take-home messages are: (a) try to iterate rather than recurse whenever you can and (b) don't blindly adapt algorithms designed for procedural programming languages to XSLT. [Tennison 2007b]*

The XSLT 3.0 `<xsl:iterate>` element provides a robust method to iterate reliably that was not available to Tennison in 2007. Beyond that, as we extend Tennison's XSLT-idiomatic implementation of a Levenshtein distance algorithm to the closely related domain of Needleman Wunsch sequence alignment, we avoid the need to maintain and traverse the entire grid that is part of the standard description of the algorithm, thus reducing the storage requirement from quadratic to linear.

Works cited

[1] Bellman, Richard E. 1952. "On the theory of dynamic programming." *Proceedings of the National Academy of Sciences* 38(8):716–19. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1063639/

[2] Bellman, Richard E. "The theory of dynamic programming." Technical report P-550. Santa Monica: Rand Corporation. http://smo.sogang.ac.kr/doc/bellman.pdf

[3] Boes, Olivier. 2014. "Improving the Needleman-Wunsch algorithm with the DynaMine predictor." Master in Bioinformatics thesis, Université libre de Bruxelles. http://t.ly/rzxZZ

[4] CollateX—software for collating textual sources. https://collatex.net/

[5] Embarrassingly parallel. https://en.wikipedia.org/wiki/Embarrassingly_parallel.

[6] "Mary Shelley's Frankenstein. A digital variorum edition." http://frankensteinvariorum.library.cmu.edu/viewer/. See also the project GitHub repo at https://github.com/FrankensteinVariorum/.

[7] "Global alignment. Needleman-Wunsch." Chapter 9 of Pairwise alignment, Bioinformatics Lessons at your convenience, Snipacademy. https://binf.snipcademy.com/lessons/pairwise-alignment/global-needleman-wunsch

[8] "The Gothenburg model." Section 1 of the documentation for CollateX. https://collatex.net/doc/#gothenburg-model

[9] Gotoh, Osamu. 1982. "An improved algorithm for matching biological sequences." *Journal of molecular biology* 162(3):705–08. http://www.genome.ist.i.kyoto-u.ac.jp/~aln_user/archive/JMB82.pdf

[10] Grimson, Eric and John Guttag. "Dynamic programming: overlapping subproblems, optimal substructure." Part 13 of *Introduction to computer science and programming*, Massachusetts Institute of Technology, MIT Open Courseware. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00-introduction-to-computer-science-and-programming-fall-2008/video-lectures/lecture-13/

[11] "Intertextual Dante." https://digitaldante.columbia.edu/intertexual-dante-vanpeteghem/

[12] Juxta. https://www.juxtasoftware.org/

[13] Kay, Michael. 2008. *XSLT 2.0 and XPath 2.0 programmer's reference.* 4th edition. Indianapolis: Wiley (Wrox).

[14] Maleki, Saeed, Madanlal Musuvathi, and Todd Mytkowicz. 2014. *Parallelizing dynamic programming through rank convergence.* Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '14), February 15–19, 2014. Pp. 219–32. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/ppopp163-maleki.pdf

[15] Multiple sequence alignment (Wikipedia). Accessed 2019-11-03. https://en.wikipedia.org/wiki/Multiple_sequence_alignment

[16] Muraoka, Yoichi. 1971. "Parallelism exposure and exploitation in programs." PhD dissertation, University of Illinois Urbana-Champaign. https://catalog.hathitrust.org/Record/100700411

[17] Naveed, Tahir, Imitaz Saeed Siddiqui, and Shaftab Ahmed. 2005. "Parallel Needleman-Wunsch algorithm for grid." Proceedings of the PAK-US International Symposium on High Capacity Optical Networks and Enabling

Technologies (HONET 2005), Islamabad, Pakistan, Dec 19–21, 2005. https://upload.wikimedia.org/wikipedia/en/c/c4/ParallelNeedlemanAlgorithm.pdf

[18] Needleman, Saul B. and Christian D. Wunsch. 1970. "A general method applicable to the search for similarities in the amino acid sequence of two proteins." *Journal of molecular biology* 48 (3): 443–53. doi:10.1016/0022-2836(70)90057-4.

[19] Saxon documentation of *saxon:threads*. https://www.saxonica.com/html/documentation/extensions/attributes/threads.html

[20] Saxon documentation of *xsl:iterate*. http://www.saxonica.com/documentation/index.html#!xsl-elements/iterate

[21] Sellers, Peter H. 1974. "On the theory and computation of evolutionary distances." *SIAM journal on applied mathematics* 26(4):787–93.

[22] Smith, Temple F., Michael S. Waterman, and Walter M. Fitch. 1981. "Comparative biosequence metrics." *Journal of molecular evolution*, 18(1):38–46. https://www.researchgate.net/publication/15863628_Comparative_biosequence_metrics

[23] Space–time tradeoff. https://en.wikipedia.org/wiki/Space%E2%80%93time_tradeoff

[24] Tennison, Jeni. 2007. "Levenshtein distance in XSLT 2.0." Posted to *Jeni's musings*, 2007-05-03. https://www.jenitennison.com/2007/05/06/levenshtein-distance-on-the-diagonal.html

[25] Tennison, Jeni. 2007. "Levenshtein distance on the diagonal." Posted to *Jeni's musings*, 2007-05-06. https://www.jenitennison.com/2007/05/06/levenshtein-distance-on-the-diagonal.html

[26] Trovato, Paolo. *Everything you always wanted to know about Lachmann's method. A non-standard handbook of genealogical textual criticism in the age of post-structuralism, cladistics, and copy-text.* Padova: libreriauniversitaria.it , 2014

[27] Van Peteghem, Julie. 2015. "Digital readers of allusive texts: Ovidian intertextuality in the Commedia and the Digital concordance on intertextual Dante." *Humanist studies & the digital age*, 4.1, 39–59. DOI: 10.5399/uo/hsda.4.1.3584. http://journals.oregondigital.org/index.php/hsda/article/view/3584

[28] Vintsyuk, T[aras] K[lymovych]. 1968. "Speech discrimination by dynamic programming." *Cybernetics* 4(1):52–57.

[29] Wang, Bin. 2002. "Implementation of a dynamic programming algorithm for DNA sequence alignment on the cell matrix architecture. MA thesis, Utah

State University." https://www.cellmatrix.com/entryway/products/pub/wang2002.pdf

Powerful patterns with XSLT 3.0 hidden improvements

Patterns have changed significantly in XSLT 3.0, opening subtle ways to improve your code, that may have been hidden in plain sight

Abel Braaksma
Exselt
<abel@exselt.net>

Abstract

With XSLT 3.0 slowly becoming more mainstream since its status as a Recommendation in 2017, it is now a good moment to review one of the smaller changes to the XSLT language, namely: patterns. Though the changes are subtle, they add some powerful new ways to the pattern syntax and template matching.

Patterns are ubiquitous in XSLT, in fact, they are the cornerstone to successful programming in this language. This paper is not meant as an introduction to patterns and pattern matching through `xsl:apply-templates`, for that other resources exist. Instead, it will focus on some of the changes in the syntax and the new additions to pattern matching rules.

After reading this paper, you should have a firmer grasp of the new capabilities of patterns in XSLT 3.0 and of ways to apply them in your own day-to-day coding practices.

Keywords: XML, XSLT, XPath, patterns, XSLT-30

1. Resources

This paper discusses the capabilities of patterns in XSLT 3.0 which has reached W3C Recommendation status in 2017[1]. The latest version can be found at [18], which is the Recommendation. When this paper refers to XPath functions, operators or syntax, it is either the XPath 3.0 Recommendation [12], together with the Functions and Operators Recommendation [14], or the XPath 3.1 Recommendation [13], together with the Functions and Operators Recommendation [15].

An XSLT 3.0 processor can support either XPath 3.0 and F&O 3.0[2] or XPath 3.1 and F&O 3.1 The 3.1 editions of these specifications define the `map` and `array` types, and the functions and operators that can operate on them, plus a number

[1]See announcement: https://www.w3.org/blog/news/archives/6377.

of smaller changes that are irrelevant for the discussion of patterns. The XSLT 3.0 specification itself defines the map types and its functions as well, leaving the main difference between XPath 3.0 and 3.1 to be the `array` type[3].

The W3C Recommendation status means that these documents can be considered final, and will not be changed in the future.

2. A quick tour on patterns

As a quick recap on what patterns are and how they are applied in XSLT, this section will provide the basics of the understanding the interaction between `xsl:apply-templates` and `xsl:template match="..."`. A good summary is given by Jeni Tennison in her book XSLT and XPath On The Edge [8]:

> *Any XSLT stylesheet is comprised of a number of templates that define a particular part of the process. Templates [are top-level constructs] defined with `xsl:template` elements, each of which holds a sequence of XSLT instructions that are carried out when the template is used.*
>
> *The two ways of using template by calling them and by applying them. If an `xsl:template` telement has a `name` attribute, it defines a named template, and you can use it by calling it with `xsl:call-template`. If an `xsl:template` element has a `match` attribute, it defines a matching template, and you can apply it by applying templates to a node [or any other sequence][4] that it matches using `xsl:apply-templates`[5].*

A stylesheet can be called in a variety of ways, typically by either implicitly starting to apply templates, or by explicitly calling a named template. Common practice has it that if you want a clear starting point when starting a stylesheet in apply-templates mode, that you define the match pattern as `match="/"`, which will match the document node at the root of a typical tree[6].

By default, a processor that is invoked with apply-templates mode, will process the *initial match selection*. In XSLT 2.0 there was only one way to invoke a processor with apply-templates mode, and that was by using a processor-dependent way of setting the *initial context node*, which could only ever be a single node. Typically, this was referred to as source or input document.

---

[2]The acronym *F&O* is short for *XPath and XQuery Functions and Operators*. When people refer to XPath they typically mean both the XPath language and the F&O. The former contains the syntax of the XPath language and its basic operations, the latter contains the definitions of all functions and operators that are available from XPath (and XQuery). Both specifications rely heavily on one another.

[3]For a full list of changes, see section I in [13] and section F in [15].

[4]Since XSLT 3.0, you can apply templates to any item, not just nodes.

[5]It is possible for a template to have both a `match` and a `name` attribute, in which case it can be both called and applied.

[6]It is no requirement that an XML tree has a document node at its root, but it is the most common scenario. We will see later how to deal with trees that are not rooted at a document node.

2.1. New invocation methods since XSLT 3.0

The following methods of invoking a stylesheet have been introduced in XSLT 3.0. Different processors will have different ways of how to configure these invocation scenarios, but each XSLT 3.0 supports them. Consult your processor's documentation for how you can utilize these ways:

- Apply-templates invocation, arguably the most common method of invoking a stylesheet. This method has the following options:
  - The *initial match selection*. This can be the source document in the form of a document node, a sequence of documents like the result of the `fn:collection` function, a single item like a number, a string or a date, a map, an array of a function item, or a sequence of multiple items, possibly of different types.
  - Optionally, the *global context item*. This item is used as the context in top-level declarations such as variables and parameters. Typically this will be set to the first item from the initial match selection, but this is no requirement and it is allowed to be absent.
  - Optionally, the *initial mode*. Each template can belong to a mode and you can apply templates with the `apply-templates` instruction to only those templates that belong to a mode by using the `mode` attribute. The default mode is the *unnamed mode*, or whatever mode is defined in the `default-mode` attribute of the `xsl:package` element. Modes can be defined explicitly with `xsl:mode` or implicitly with the `mode` attribute. Specifying an initial mode will start the transformation scenario in that mode.
  - Optionally, a list of parameters. Available parameters are defined with the `xsl:param` declarations inside the `xsl:template` declarations. Parameters can be optional or required.
- Call-template invocation. This method has remained largely the same since XSLT 2.0, but a few additions have been made:
  - Invoking a call-template transformation scenario without a name will now default to a pre-defined name which is the same for all processors: `xsl:initial-template`. If a template is defined with that name, it will be the default entry point for this method of invocation.
  - Optionally, a context item to be used with the called template. Since XSLT 3.0 it is possible to define named templates with a required, absent or optional context item through the `xsl:context` declaration. If such declaration is absent and a specific context item is not given, it default to the *global context item*, which in turn can be controlled by the top-level `xsl:global-context-item` declaration.

- Function-call invocation. This method is entirely new in XSLT 3.0 and allows you to execute an individual stylesheet function. This function has to be available and must have visibility *public* or *final*. Options are:
  - Name and arity of the stylesheet function. Stylesheet functions are defined with `xsl:function` and can be overloaded, that is, the same function can exist with a different number of parameters. Processors will allow you to specify precisly what function with what arity to call.
  - A list of items to act as parameters. Other than for templates, parameters for functions are positional and can be defined without giving their name. The number of items must be the same as the arity of the function.
- For all transformation scenarios:

  Optionally, controlling how the result of the invocation should be returned: as a *raw result*, as a *tree* by using the `build-tree` attribute on `xsl:output`, or *serialized*. The latter was the default in XSLT 2.0. Typical serializations include XML and HTML. New in XSLT 3.0 are HTML5, XHTML5, JSON and Adaptive[7].

For the remainder of this paper, we will assume *apply-templates invocation*, as that is the main method for starting a transformation and for applying templates against the matching patterns we will discuss in the up-coming chapters.

2.2. The role of patterns in an XSLT stylesheet

A pattern can be seen as a boolean expression: either an item or node matches the pattern, or it doesn't. If it matches, the node will be selected, which in the case of a template means the template will be executed with that node as the context item.

As mentioned above, a stylesheet typically contains a bunch of top-level elements that are templates. From an imperative view, they can be considered a large `switch` statement, optionally tagged or grouped by their `mode`, where the switch is initiated each time the processor encounters an `xsl:apply-templates` instruction. The `select` attribute of that instruction can be used to limit or broaden the actual nodes the templates can act on. If that attribute is absent, the children of the context node are selected[8].

Templates are typically applied recursively. That is, inside another template, applying templates again through `xsl:template` will apply all templates again.

[7]The serialization methods JSON and Adaptive are only available when XPath 3.1 is supported by the processor. All processors support HTML 5.

[8]The exact expression for the default `select` attribute is `child::node()`. This has the effect that all nodes that are children, but not deeper descendants, are selected. This excludes attributes and namespace nodes, which technically are not children, and won't select a document node, which cannot appear as child or a parent node. From the seven node kinds, this leaves element, comment, text and processing instruction nodes to be selected by default.

As long as children (the default) or descendants are selected, this will not end in an endless loop, however, it is possible to select the current context node, or a parent thereof, again. If re-processing the same node is necessary for your scenario, it is best to do that by specifying a different mode.

An alternative to re-process the currently selected node is by using `xsl:next-match`, which will select the next match in priority order (details below), or `xsl:apply-imports`, which will select the next match in imported order.

A simple example stylesheet (the root element `xsl:stylesheet`, `xsl:transform` or `xsl:package` is omitted in this and other examples for clarity) with the imperative explanation is as follows:

```
<xsl:template match="/">
    <result>
        <xsl:apply-templates />
    </result>
</xsl:template>

<xsl:template match="book">
    <xsl:apply-templates select="*" />
</xsl:template>

<xsl:template match="author/name">
    <author><xsl:value-of select="." /></author>
</xsl:template>

<xsl:template match="text() | comment()" />
```

If the example above is the whole stylesheet, and it is invoked with a document containing `book`, `author` and `name` elements, possibly among others, then an imperative way of reading it is as follows:

- If the current node is the root node, then output `<result>` and inside it, process the children of the document node.
- If the current node is a `book` element, then output nothing, but select all elements that are children of that element.
- If the current node is a `name` element with a parent `author`, then output an `<author>` element and the value of the current node (the name of the author). Do not further apply templates.
- If the current node is a text or comment node, then output nothing, and do not further apply templates.
- If the current node is anything else, apply the default templates. This will in turn apply templates to the chilren and the children of children in depth-first traversal of the tree[9].

2.3. Priority of templates

It is possible, and in fact quite likely, that multiple templates can match the same node. This is called a *conflict* and you have several ways of dealing with such conflicts. By default, the template with the highest priority is chosen. If there are multiple templates with the same priority that match, then it depends on the setting of the attribute `on-multiple-match` of the corresponding `xsl:mode` declaration[10]. The priority resolution goes as follows:

- First, the *import precedence* is considered, and only those templates with the highest import precedence. This effectively means that if you used `xsl:import`, and you have a matching template for the current node in the imported and the main stylesheet, that the matching template rule[11] in the main stylesheet will be considered, and not the one in the imported stylesheet. Using `xsl:apply-imports` will instruct the processor to apply the imported matching templates, or the default template rules if none is found.

- Secondly, the priority is considered. The processor assigns a default priority between -1 and +1 inclusive, but programmers can assign their own priority. The match having the highest priority will be chosen. The instruction `xsl:next-match` can be used to instruct the processor to consider matching templates of a lower priority in the same import precedence[12], then the next matching template in declaration order[13], or the default template rules if none is found.

- Thirdly, the declaration order is considered. By default, the last matching template will be taken. Such a conflict is often a sign of a programming error, and such error can be raised by setting `on-multiple-match="fail"`[14] on the corresponding `xsl:mode` declaration. It is good practice to indeed set this value to `"fail"`, which will allow better analysis of such errors. Again, as with the previous bullet point, `xsl:next-match` can be used to select the next in declaration order[15]. If you don't want an error, but wish to be informed of such

[9]The default templates are briefly discussed below in the section on default templates.
[10]For details, see section 6.4 and section 6.6.1 of [18].
[11]The specification talks of *template rules*, whereas in this paper I will typically use the term *matching template* or *template match*. The terms are interchangeable and refer to an `xsl:template` declaration with a given `match` attribute, its optional parameters and its contents, called the *sequence constructor*.
[12]If a matching template with a lower import precedence exists, `xsl:next-match` will process that instead. There is no mechanism to solely invoke the next matching template in the current import level alone. To overcome this limitation, modes can be used.
[13]later in the stylesheet means higher in the matching order, in other words, `xsl:next-match` looks *up* the tree, not *down*.
[14]The only other valid value is `use-last`, which is the default and does not need to be set explicitly.
[15]If you use both `on-multiple-match="fail"` and `xsl:next-match`, then for cases where there are two matching templates on the same import precedence level, an error will be raised. Therefore,

matching conflicts, you can opt to set `warn-on-multiple-match="true"` on the corresponding `xsl:mode` declaration.

XSLT 3.0 has a big new feature: packages with `xsl:package`, `xsl:use-package`, `xsl:expose` and others [4]. Using packages allows you to override components in a more consistent manner than through import precedence by using `xsl:override`[16]. This applies to functions, variables, named templates and named attribute sets, as well as for template rules. Only named modes can be overridden, which in practice means, they can be expanded upon by writing `xsl:template` declaration under the `xsl:override` declaration using the given mode name.

If conflicts occur in matching templates in package hierarchies, any overriden template rule takes precedence over any used template rule (through a used package with the `xsl:use-package` instruction). Within this set of overriding templates, the second and third conflict resolution points above apply. If still no overridden rule matches, the matching templates in the used package (within the same mode) are considered. Here, all three conflict resolution rules apply[17]. Like with the previous precedence rules, it is possible to use `xsl:next-match` to invoke the used template rules from the used package, if any. However, it not possible to use `xsl:apply-imports` within an overriding matching template, it will raise error `XTSE3460`[18].

2.3.1. Default priority

In a matching template declaration you can give an explicit priority: `<xsl:template match="*" priority="5" />`. If the `priority` attribute is absent, the processor will assign a default priority. Roughly said, this default priority assigns a higher priority to more specific patterns, but this is not always the case. For instance, a match pattern with one predicate and one with 10 predicates both receive the same priority, even though the latter is much more specific. A summary of the rules is as follows, from low to high[19]:

`xsl:next-match` will never select another matching template declaration at the same level and with the same lowest priority and will instead have the effect of calling the default templates.

[16]Packages are a large subject on their own and may be the subject of a future talk. Several websites and the slides in reference [4] provide good starting points.

[17]The reason that `xsl:import` does not apply to overriding templates is that `xsl:override` is part of an implicit or explicit package, and `xsl:import` cannot be used with either, it can only be used with importing a stylesheet module, which cannot contain overridable components. See for discussion the following W3C bug report: https://www.w3.org/Bugs/Public/show_bug.cgi?id=24310.

[18]According to bug report #29210, comment #3, this error should have been dropped and the list of imported template rules be empty by definition, without causing an error. The report also mentions that the error cannot always be raised statically. See: https://www.w3.org/Bugs/Public/show_bug.cgi?id=29210.

[19]The precise rules can be found in section 6.5 of the specification [18].

- **-1.0**, if the pattern is a *predicate pattern* of the form `"."` (which matches any item or node).
- **-0.5**, if the pattern is any of the following:
  - exactly `"/"` or `"*"`;
  - exactly `node()`;
  - any of `element()`, `attribute()`;
  - any of `element(*)`, `attribute(*)`[20];
  - exactly `"document-node()"`;
  - any of `text()`, `processing-instruction()`, `comment()`, `namespace-node()`;
  - a document-node test with an element test like above, for instance `document-node(element(*))`;
  - any of the above, preceded by an axis, for instance `child::element()`, `namespace::*`.
- **-0.25**, if the pattern is a single path expression like any of the following:
  - `ns:*` (if only the namespace is specified);
  - `*:foo` (if only the local-name is specified);
  - `Q{http://somenamespace}*` (if only the namespace is specified)[21];
  - any of the above, preceded by an axis, like `child::ns:*`.
- **0.0**, if the pattern takes any of the following forms:
  - a single path expression, like `book`;
  - an element or attribute test like `element(foo)`, `attribute(bar)`;
  - an element or attribute test with only the type specified, like `element(*, xs:string)`;
  - a node-test for a specific processing instruction, like `processing-instruction('bar')`;
  - a document-node test with an element test like the above, for instance `document-node(element(book))`;
  - any of the above, preceded by an axis, for instance `child::author`, `descedant::para`, `self::processing-instruction('bar')`, `attribute::foo`[22].
- **+0.25**, if the pattern takes any of the following forms:

[20]The specification [18] does not mention `@*`, though it is likely that this was omitted by accident. In the XSLT 2.0 specification [16], section 6.4, it is correctly specified and given a priority of -0.5, which is what all tested processors do in XSLT 3.0 as well.
[21]This rule is not in the specification, but in the errata [17].

  - an element or attribute test with both name and type specified, like `element(author, xs:string)` or `attribute(id, xs:ID)`;
  - a schema-element test like `schema-element(X)`;
  - a schema-attribute test like `schema-attribute(X)`;
  - a document-node test with an element test with both name and type specified, like `document-node(element(author, xs:string))`;
  - a document-node test with a schema-element test, like `document-node(schema-element(X))`;
  - any of the above, preceded by an axis, for instance `descendant::element(author, xs:string)`.
- **+0.5**, if the pattern does not fit in any of the above categories, and is not a *predicate pattern*. This is true, for instance, as soon as a pattern has more than one path element, as in `book/author`, or has one or more predicates, as in `book[3]` or `book[author="Tolkien"][title]`.
- **+1.0**, if the pattern is a *predicate pattern* of the form `".[…][…]`, that is, has one or more predicates.

It should be noted that it is possible to write patterns that match a more generic set of nodes, that have nonetheless a higher precence. For instance, if you were to write `node()[self::element()]`, it has priority **+0.5**, and matches any element, but the more selective pattern `book` will only match elements that have the name `"book"`, but this has now a lower priority of **0.0**.

As mentioned before, to prevent confusion and unless priorities are really trivial or irrelevant (for instance, if your pattern matches one and only one node from your input document), then it is best to specify priorities explicitly, or switch to a different mode to prevent multiple matches or match conflicts that are otherwise hard to diagnose.

2.3.2. Priorities as inheritance

Another way of looking at the priority conflict resolution mechanism is as a way virtual methods work in object-oriented languages. In OO, the most specific virtual method usually wins when there are multiple overriding definition in the OO hierarchy. This is the same with XSLT's matching templates: the most specific, i.e. closest to the definition in the principal stylesheet, usually wins.

Writing stylesheets to match templates that depend highly on those relatively complex rules of *template rule inheritance* (not a real term), is often considered poor form and in courses and books the typical advice is to either use explicit pri-

[22]The specification [18] does not mention `@foo` as abbreviated forward step, though it is likely that this was omitted by accident. In the XSLT 2.0 specification [16], section 6.4, it is correctly specified and given a priority of 0.0, which is what all tested processors do in XSLT 3.0 as well..

orities using the `priority` attribute, or by using modes. In XSLT 3.0 you can now require modes to be declared by setting `declared-modes="true"` in the `xsl:package` element, which makes them more resilient for typos, and a proper structure with modes is often the most readable one in complex scenarios.

Using XSLT packages with `xsl:mode` and setting its `visibility` attribute to `private`, `final` or `public`, when used through an `xsl:use-package` declaration they can be overriden in `xsl:override` if public, used if they are `final` and hidden and never used if they are `private`. This provides a better protection to the *template rule inheritance* chain than is available with `xsl:import`, which is flimsy at best.

2.3.3. Mode declarations

What happens if you apply templates to a set of nodes or other items and there is no matching template? In XSLT 1.0 and 2.0, this would mean that the default template is called and generally speaking, this would output the *value of* the element nodes only. In other words: if your output contains only the text nodes from the input tree, you know that your templates are not being matched correctly.

This behavior has been one of the most controversial, and also one of the most asked about on sites like StackOverflow and the XSL Mailing List. XSLT 3.0 attempts to alleviate the pain a little bit by allowing you to have more control over the behavior of the processor when it comes to non-matching templates through declared modes using the `xsl:mode` declaration.

The full syntax allowed by `xsl:mode` is as follows:

```
<xsl:mode
  name? = eqname
  streamable? = boolean
  use-accumulators? = tokens
  on-no-match? = "deep-copy" | "shallow-copy" | "deep-skip" | "shallow-
skip" | "text-only-copy" | "fail"
  on-multiple-match? = "use-last" | "fail"
  warning-on-no-match? = boolean
  warning-on-multiple-match? = boolean
  typed? = boolean | "strict" | "lax" | "unspecified"
  visibility? = "public" | "private" | "final" />
```

Some of those options have already been discussed or are clear from the their name, nevertheless, let's briefly go over each of them before diving into the default templates.

- `name`, if present is the name of the mode, if not present, defines the behavior of the default mode.
- `streamable`, if present and set to `true`, declares that all patterns and templates in this mode must meet the streamability requirements.

- `use-accumulators`, only applicable if `streamable="true"`, and defines which accumulators need to be calculated while the mode is active, and these accumulators must themselves be streamable. This distinction allows mixing non-streamable accumulators and streamable accumulators in a mixed-mode transformation where both streamable and non-streamable modes are used.
- `on-no-match` will be explained in the next section. The default is `text-only-copy`.
- `on-multiple-match` was discussed above, and defines whether an error should be raised when equal-priority and equal-import-precedence matches are encountered. The default is `use-last`.
- `warning-on-no-match` defines whether a non-match in this mode will lead to a warning. This can be helpful in analyzing pattern issues. Though the warning is processor-defined, it will likely give position and description of the node that is not matched explicitly. Setting this to `true` will not prevent the default templates to be used, but will issue a warning in such cases. The default value is implementation-defined, but most processors have this set to `false`.
- `warning-on-multiple-match`, if present and set to `true`, will issue a warning when multiple template rules are matched that have the same priority and import-precedence. It is therefore similar to `on-multiple-match`, but will not halt the processor. The default is implementation defined, but most processors appear to have this set to `false`.
- `typed` determines whether the document or node(s) processed by this mode should be typed or not. This is mainly relevant for *schema-aware* processors. It has the following allowed values:
  - `unspecified` (the default), whether or not the nodes processed by this mode are typed is irrelevant.
  - `true`, means all nodes must be typed. Using `xsl:apply-templates` with this mode and the selection contains one or more nodes that are untyped (i.e., have `xs:untyped` or `xs:untypedAtomic`) will lead to an error.
  - `false`, means none of the nodes must be typed. If any node has a different type than `xs:untyped` or `xs:untypedAtomic`, an error will occur.
  - `strict` is almost analogous to `true`, except that for each pattern that matches elements by its EQName, the element-name in the first step in such expressions must be available in the in-scope schemas, and it is interpreted as if it was written as `schema-element(E)`, where `E` is the name of the element. For non-elements and wild-card matches, this rule does not apply.

  - `lax` is the same as `strict`, except that no error is raised if the element declaration is *not* available in the in-scope schemas.

- `visibility` applies to packages. If it is `public`, it is possible for a *using package* to add matching templates to this mode. If it is `final`, the mode is available and can be used in `xsl:apply-templates`, but cannot be expanded. If it is `private`, it is not available in *using package*, but only in the *containing package*. The unnamed mode is always private and it is not possible to give it a different visibility.

As described before, using an `xsl:package` element as the top element of your stylesheet forces you to use `xsl:mode` for each mode that you want to use. If it is `xsl:stylesheet` or `xsl:transform`, you can still create modes the old way[23], just by using a new name in the `mode` attribute of `xsl:template`. This mode will then have all the default settings only. You can change this behavior by setting the `declared-modes` attribute on `xsl:package`. This attribute is *not* available on `xsl:stylesheet` or `xsl:transform`, though there's nothing stopping you from declaring modes regardless.

For more control over your modes and less chance of typing errors leading to modes magically coming into existence, it is commonly considered best-practice to always declare modes to enforce that by using `xsl:package`. Using that as top-level element does *not* change the behavior of the stylesheet. In fact, if you use `xsl:stylesheet` or `xsl:transform` instead, these are internally transformed into an `xsl:package` anyway, with all other things remaining equal.

2.3.4. The six build-in templates

There are in total six default, or build-in templates that are called when there's no matching template. Which one is effectively called depends on whether there are nodes or items that are not matched by any of the matching templates, and which one is requested by the `xsl:mode` declaration.

If build-in templates skip over, or shallow-copy nodes and process nested children, they will always stay in the current mode when the implicit `xsl:apply-templates` is called. Likewise, any parameters remain untouched (that is, they are passed on).

It is not possible to call a build-in template rule directly. However, a simple trick is to declare a mode through `xsl:mode` with a unique name and no matching templates in that mode. Applying templates to such an empty mode, will call the build-in template as defined on the `on-no-match` attribute of that `xsl:mode` declaration.

[23]The "old way" for mode declarations are officially called *implicit mode declarations*, if there's an explicit matching `xsl:mode` declaration for a mode, this is called an *explicit mode declaration*.

- `text-only-copy`, this is the same behavior as in XSLT 1.0 and XSLT 2.0. The rules are a little different because of the possibility to match any item:
  - Document nodes and elements are not copied, but their contents are applied as if there's one `<xsl:apply-templates />` statement.
  - Text nodes and attribute nodes, their string value is copied.
  - Comments, namespace nodes and processing instructions are skipped.
  - Atomic types, their string value is copied.
  - Functions and maps are skipped.
  - Arrays, all items in the array are aplied as if there's one `<xsl:apply-templates select="*?" />` statement[24].

  The equivalent templates for above behavior could look as follows[25]:

```
<!-- skip document nodes and elements, but process children -->
<xsl:template match="document-node()|element()" mode="M">
  <xsl:apply-templates mode="#current"/>
</xsl:template>

<!-- output text and attribute nodes value -->
<xsl:template match="text()|@*" mode="M">
  <xsl:value-of select="string(.)"/>
</xsl:template>

<!-- output any atomic type's value -->
<xsl:template match=".[. instance of xs:anyAtomicType]" mode="M">
  <xsl:value-of select="string(.)"/>
</xsl:template>

<!-- skip any other node -->
<xsl:template
   match="processing-instruction()|comment()|namespace-node()"
   mode="M"/>

<!-- skip functions and maps -->
<xsl:template
   match=".[. instance of function(*)]"
   mode="M"/>

<!-- process items of an array -->
<xsl:template match=".[. instance of array(*)]" mode="M">
```

[24]Arrays are only supported by processors that support the XPath 3.1 feature. If only XPath 3.0 is supported, arrays are not an available type, nor is the related syntax.
[25]Those examples come from section 6.7.1 of the XSLT 3.0 specification [18] and illustrate the behavior, but don't include the maintaining of the parameters, which cannot be expressed this way.

```
  <xsl:apply-templates mode="#current" select="?*"/>
</xsl:template>
```

The big two surprises in this behavior are that any item, like a string, number or QName will be output, and that the content of arrays are further processed.

- `deep-copy`, essentially means: if a node is matched, the whole node is copied, as if copied by the `xsl:copy-of` instruction. This includes all it descendants, and no further processing takes place. This is not the same as an *identity template*, as for that the descendants should be processed. The code could look as follows:

```
<!-- copy any item, do not process children further -->
<xsl:template match="." mode="M">
  <xsl:copy-of select="." validation="preserve"/>
</xsl:template>
```

This behavior means that functions, maps and arrays are copied to the output as-is, but they are not atomizable. If your input contains such items, it will lead to an error upon serialization. It is allowed to have non-atomizable items in your output, but then you should not serialize it, instead, you should process the raw result, or catch the result in a variable and re-apply it for further processing.

- `shallow-copy`, essentiall means: if a node is matched, that node is shallow-copied as if copied by the `xsl:copy` instruction. The descendants are then further processed. This is closely similar to the popular *identity template* programming model. The code could look as follows:

```
<!-- process contents of nodes, copy any other item -->
<xsl:template match="." mode="M">
  <xsl:copy validation="preserve">
    <xsl:apply-templates select="@*" mode="M"/>
    <xsl:apply-templates select="node()" mode="M"/>
  </xsl:copy>
</xsl:template>
```

The same notes for maps, arrays and functions, as mentioned with `deep-copy`, apply here.

The two `xsl:apply-templates` lines have the effect that the size and position of the nodes during further processing can be different from the more traditional `<xsl:apply-templates select="@* | node()" />`. With the build-in template there are two sets, both starting at position 1, one with all attributes, the other with all other nodes.

Namespace nodes are not selected and applied from inside the `xsl:copy`, but they are copied to the output as a result of how `xsl:copy` works. The only

way to process namespace nodes is to select them specifically inside the `xsl:apply-templates` call in user code.

- `deep-skip` is the opposite of `deep-copy`: any node, except document nodes, are skipped and their descendants are not processed further. This can be useful if you are only interested in a small subset of nodes from the input tree. Using this as the default template setting means that every node must be carefully matched, or the output will not contain it. The equivalent template rules for `deep-skip` are:

```
<!-- process contents of document nodes -->
<xsl:template match="document-node()" mode="M">
  <xsl:apply-templates mode="#current"/>
</xsl:template>

<!-- stop processing anything else -->
<xsl:template match="." mode="M"/>
```

- `shallow-skip` is the opposite of `shallow-copy`: any node is skipped, but the descendants of the node are processed further. Any other item is skipped without further processing, except for arrays, in which case each element in the array is processed further.

```
<!-- process contents of document and element nodes -->
<xsl:template match="document-node()|element()" mode="M">
  <xsl:apply-templates select="@*" mode="#current"/>
  <xsl:apply-templates mode="#current"/>
</xsl:template>

<!-- process each item in the array -->
<xsl:template match=".[. instance of array(*)]" mode="M">
  <xsl:apply-templates mode="#current" select="?*"/>
</xsl:template>

<!-- skip the rest -->
<xsl:template match="." mode="M"/>
```

- `fail` simply raises an error when no match is found in the user supplied matching templates. This is equivalent to `warning-on-no-match="yes"`, except that instead of a warning, an error is raised and further processing stops. It is equivalent to the following matching template:

```
<!-- throw error on any item not matched -->
<xsl:template match="." mode="M">
  <xsl:message terminate="yes" error-code="err:XTDE0555"/>
</xsl:template>
```

3. What's new in XSLT 3.0 patterns

At first glance it might seem that patterns in XSLT 3.0 didn't get that much of an overhaul, especially compared with large new features of the language like packages, maps, higher order functions, accumulators and streaming. However, the syntax has been brought more in line with XPath syntax, parenthesized expressions are now possible, tfunctions have been added for *rooted patterns*, as well as `except` and `intersect` expressions at the top level of a pattern. Furthermore, much requested axes have been added. Where in XSLT 2.0 only the child and attribute axes were available[26], this has now been expanded to include all forward axes: `self`, `namespace`, `descendant` and `descendant-or-self`.

3.1. Main new features

The following is a list of the new features of the pattern language, along with several additions in other areas of XSLT that influence how matching patterns behave:

- *Predicate patterns*. These are patterns of the form `.[predicate]`, where predicate is any XPath predicate expression. Such patterns can be used to match any node, atomic type, map, array or function. This is arguably one of the biggest changes to the pattern syntax, as previously patterns were only allowed to operate on nodes.

  There's one small, yet important difference between matching with *predicate patterns* and normal patterns: the former are matched with singleton focus, which means that size and position are always one. While normal node patterns *can* match on position. Therefor, `match=".[2]"` will never select anything, not even a second child node, while conversely, `match="*[2]"` will match each second child element, or `match="node()[2]"` will match each second node.

- *Applying templates to any kind of item*. Previously, using `xsl:apply-templates` only applied to nodes, and trying otherwise resulted in an error. In line with the mentioned predicate patterns, it is now possible to select any kind of item. For instance, `<xsl:apply-templates select="('one', 'two', 'three')" />` will apply templates on the three strings in the sequence. Normal node patters won't match these, but a predicate pattern would. For instance:

```
<xsl:template match=".[. = 'one']">
    <xsl:text>Caught the first!</xsl:text>
</xsl:template>
```

[26] One might argue that an XSLT pattern always could have a pattern like `"foo//bar"`, but technically, this expands to a child axis on the last step.

```
<xsl:template match=".[. instance of xs:string]">
    <string value="." />
</xsl:template>
```

One sublety remains: if you use `xsl:apply-templates` without a `select` attribute, the default of it selecting the children of the context node remains. If the context item is not a node, this will raise an exception.

- *New axes: `self`, `namespace`, `descendant` and `descendant-or-self`*. These axes were not previously directly available, though a close proximity with the descendant axis could be achieved with the double-slash path operator `//`. Now, these axes are directly available in any pattern expression. Using these patterns influences counting position and size, which is explained in the next section.
- *`except` and `intersect` patterns*. In XSLT 2.0 it was comparatively hard to match over a set of nodes *except* an other set of nodes. Suppose you want to match all elements, except `div`, you can now write a pattern like `match="element() except div`.
- *Parenthesized patterns*. On the face, this is a trivial change, allowing parentheses around pattern expressions. However, the details of the syntax rules provide a loophole to match against disjunctive trees (as opposed to matching only against the current node and its ancestors), for instance, `chapter/(/section/list)/para` is a valid expression. How this can play out, and how processors support this kind of expression is explained in the section on parenthesized patterns.
- *Additional functions in rooted patterns*. In XSLT 1.0 and 2.0, a pattern was allowed to start with `id` and `key`. Especially the latter has proven to be very useful in XSLT 1.0 to provide Muenchian Grouping[27] and other optimizations. XSLT 3.0 expands on this set by adding: `doc`, `element-with-id`, `root`. These functions, esp. the `doc` function, can add a simple, but powerful way to check for the origin of nodes. The `root` function can be helpful with, contrary to its name, matching against parentless nodes. This will be explored in the section on new functions.
- *Rooted patterns with variable reference*. A rooted path can start with a variable reference, as with `$doc/chapter`. This allows to match against the same tree the variable reference refers to. This effectively allows certain patterns with

[27]Muenchian, or Münchian Grouping was a technique developed by Steve Münch that allowed efficient grouping in XSLT 1.0. Since the advent of `xsl:for-each-group` in XSLT 2.0, this has become a less needed technique, but keys can still be used to speed up matches in particularly complex cases that would otherwise involve expensive `O(n^2)` predicates with the `following(-sibling)` or `preceding(-sibling)` axes. For more info and a discussion, see [11].

axes that would've otherwise been illegal in the pattern syntax. The section on rooted patterns explores this deeper.

- *Comments in patterns*. Strictly speaking, XSLT 2.0 did not allow XPath-style comments of the form `(: a comment here :)` to be used within patterns. In XSLT 3.0 this is allowed in all places where XPath allows it, to align it better with XPath. This can be useful to document large and complex, multi-line patterns.

- *Errors in patterns match false*. In XSLT 2.0 errors in patterns were considered *recoverable errors*. The notion of recoverable errors has disappeared entirely in XSLT 3.0, and the default action on such errors is now mandatory. This means that an error in a pattern, other than a static error or static type error, will lead to a pattern never matching. Such errors can happen dynamically, for instance when converting a node to number and there is now no way to catch such errors anymore. See section on errors for a way around this limitation.

3.2. Other related new features

Apart from the above list, there are several smaller changes in relation to patterns that have improved or changed, and some other new features that are also useful in patterns.

- *Streamable patterns*. If you need to process large documents, XSLT 3.0 introduces the *streaming feature*[28], which requires the patterns to be streamable. This paper will not go into this subject as it is vast and beyond its scope, however, I and others have previously given talks on streaming, see [1], [2], [5], [6].

- *Explicit mode declarations*. Previously, modes existed just by naming them in the mode attribute of `xsl:template`. It is now possible to give a mode more properties and to explicitly declare them with `xsl:mode`, such as that typed input is required, what to do when there is no match, what accumulators are applicable and whether streaming is allowed. Furthermore, mistakes in naming modes can be caught by using `declared-modes="true"`.

- *Initial match selection*. Previously, the input to an XSLT stylesheet was a single document or node. If you were to process multiple documents, you would need to use stylesheet parameters, or the `doc`, `document` or `collection` functions. Now, the input can be any sequence of any type. It can be seen as if the stylesheet was called with an initial call to `xsl:apply-templates` with the initial match selection as the result of its select expression.

[28]Whether a processor supports streaming can be checked with the expression `system-property('xsl:supports-streaming')`.

Each item in the initial match selection will be matched initially against the available `xsl:template` declarations in the given mode, with the item, its position and size of the selection as the focus.

- *Matching parentless namespace nodes*. This fixes a bug in XSLT 2.0. You were allowed to have a pattern like `match="namespace-node()"`, but it would only ever match namespace nodes that have a parent. The rules have been updated to allow it to match parentless namespace nodes[29], and the `namespace` axis is now also made available.

- *Qualified names for root pattern functions*. Previously it was illegal to use qualified names like `fn:id` or `fn:key` within a pattern. This restriction is now lifted. You can also a URI Qualified Name like `Q{http://www.w3.org/2005/xpath-functions}doc`, which is sometimes helpful in auto-generated patterns, or patterns that have been created using the new XPath 3.0 `path` function[30].

- *Expanded QNames for name-tests*. Technically a feature of XPath 3.0 [12] and 3.1 [13], an *expanded qualified name*, or *EQName* can now be used as a name-test. An EQName has the form `Q{nsURI}localpart`. Within the accolades, which is whitespace-sensitive, you put the namespace URI, after the closing accolade, you put the local name. The namespace does not have to be declared, which can be handy if you need a namespace only once, or when the paths are created from output from the `fn:path` function.

  Suppose you have a namespace declared and in scope as `xmlns:ns="urn:my-namespace`, then `ns:person` and `Q{urn:my-namespace}person` are equivalent for all intents and purposes. Likewise, `ns:*` is the same as `Q{urn:my-namespace}*`.

- *Third argument in key is allowed*. You can now write `key(X, Y, Z)`, where `Z` points to a document node that the key should appear in. This allows you to use a global variable as the third argument, set to an external document and to match over that explicitly using keys. In certain cases this simplifies stylesheet development that involves multiple documents.

- *Second argument in fn:id and fn:element-with-id*. Similar to previous point, where you can set the second argument to a document rooted in a specific tree, by using a global variable that points to such tree. That way, these functions will only match on id's in that specific document.

[29] A parentless namespace node is very rare, and arguable, matching over them even rarer. You can create one throug the new `copy-of` function, the `xsl:copy(-of)` instruction, or by using the `xsl:namespace` declaration.

[30] The `fn:path` function will output `Q{http://www.w3.org/2005/xpath-functions}root` as the start for paths that include a root when the root is not a document node.

- *`union` instead of `|` can now be used*. This change merely aligns the pattern syntax better with the XPath syntax. Writing `foo union bar` was disallowed in XSLT 2.0 and could only be written as `foo | bar`. This restriction is now lifted.
- *Patterns as shadow attributes*. Since XSLT 3.0 you can turn any attribute into a statically expanded attribute, aka *shadow attribute*, that takes an XPath expression that is evaluated at compile-time. To do so, simply prepend it with an underscore `_`. For instance, writing `<xsl:template _match="$var"` means that you can set the *static parameter* `$var` to whatever pattern you want. This feature allows for a certain level of meta-programming.

All the changes to the pattern syntax are available in all locations where patterns are allowed. These places are:

- `xsl:template`, the `match` attribute,
- `xsl:number`, the `count` and `from` attributes,
- `xsl:accumulator-rule`, the `match` attribute,
- `xsl:for-each-group`, the `group-starting-with` and `group-ending-with` attributes.
- `xsl:key`, the `match` attribute.

4. Position and size in XSLT 3.0 patterns

In XSLT 2.0, it was simple: position was always relative to the child axis, even when you used the `//` operator, since the latter expands to `/descendant-or-self::node()/`, and a name-test without a specific axis is essentially a child-axis name-test. Which means, given an expression like `foo//bar`, this expands to `child::foo/descendant-or-self::node()/child::bar` and if you were to use a positional predicate, as with `foo//bar[4]`, this would therefore select the fourth `bar` child element.

This all changes in XSLT 3.0, where all forward axis are available in a pattern expression, plus parenthesized expressions also influence counting. The counting rules are the same as with XPath, and have not changed since XPath 1.0. For every axis, the counting is done based on the node test. The node test is the part *after* the `::`. For instance, `child::author` counts only the *elements* that match `author`; `child::*` counts all elements, `child::node()` counts all nodes etc.

If predicates are chained, the size and position are dependent on the predicates that come before. For instance, `child::author[@name][3]` will select the third element that has name `author` *and* attribute `name`. Once a step expression returns a singleton, size and position remain 1. Predicates can never increase the size of the set.

As a summary, counting is as follows:

- *child axis*: counts towards the immediate children, order is document order.

- *attribute axis*: counts the attributes, order is implementation dependent, but stable.
- *namespace axis*: counts the namespace nodes, order is implementation dependent, but stable.
- *descendant axis*: counts all children and children of children etc, depth-first, order is document order.
- *descendant-or-self axis*: same as *descendant* axis, but includes *self*, provided it matches the nametest, of course.
- *self axis*: counts only self, that is, size is 1 or 0.
- *no axis*, depends:
  - *implicit child axis*, this is true if axis is absent for a name test, like with `person`, or it is a kind test for element, comment, text or processing instruction. Then same as *child axis* above.
  - *implicit attribute axis*, this is true if it is a kind test for an attribute node, like `attribute(age)`. Then same as *attribute axis* above. The `@` prefix is a short way of explicitly using the attribute axis.
  - *implicit namespace axis*, this is true if it is a kind test for a namespace node, like `namespace-node()`. Then same as *namespace axis* above.
  - *implicit self axis*, this is only true if it is the *first step* of a pattern, and the step is a document node kind test, like `document-node()` or `document-node(element(root))`. Then same as *self axis* above.

Special attention should go to the position of parentless nodes. Suppose you have a variable like the following:

```
<xsl:variable name="people" as="element()*">
    <person>John Doe</person>
    <person>Angela Dickens</person>
</xsl:variable>
```

If you apply over this variable with `xsl:apply-templates`, to match the parentless node you can simply do `match="person"`, since special rules require this to match *child or top* elements. But suppose you want to match the second person in `$people`? You may be tempted to do `match="person[2]"`. But this will never match anything, because the elements inside the variable are without parent, they do not have a root document node.

However, after entering the template, the position in the sequence and the size of the sequence are available. As a workaround you can use something like `<xsl:if test="position() = 2">...` Another workaround is to change the variable to have an implicit document node, which you can achieve by, for instance, omitting the `as="element()*"`[31].

5. Reading a pattern

What is a pattern really? What does it mean to write `match="book/title"`? Patterns are designed to allow processors to quickly determine if a node belongs to a given pattern. A pattern itself is a subset of an XPath expression, simplified precisely for this purpose. The reason that only a subset of axes is available is to allow all steps on a pattern to be exclusively on the ancestor axis alone.

Officially, a pattern answers the question: given an pattern P, and a node N, then, with focus on N and with its position and size set to 1[32], does N occur in the result of the expression `root(.)//(P)`[33]? As an example, consider the following two templates:

```
<xsl:template match="book[1]/title">
    <first-book><xsl:value-of select="." /></first-book>
</xsl:template>

<xsl:template match="title">
    <other-book><xsl:value-of select="." /></other-book>
</xsl:template>
```

the pattern `book[1]/title` and the following input document:

```
<list>
    <book>
        <title>Lord of the Rings</title>
    </book>
    <book>
        <title>Lord of the Rings</title>
    </book>
    <book>
        ... etc
</list>
```

Then answering the question could go something as follows:

- Set N to be the current node, let's say we're processing the first `title` element through `<xsl:apply-templates select="/list/book/title" />`.
- Set its position and size to 1. Now `.` refers to the node, `position()` is `1` and `last()` is `1`.
- Evaluate the XPath expression `root(.)//(book[1]/title)`. The result is the first `title` element, let's call it R.

[31]A variable without an as-clause, that has a sequence constructor (as opposed to a `select`-statement), defaults to creating a document node with as its contents the contents of the sequence constructor.
[32]This is called a *singleton focus*.
[33]There's a little bit more to getting to a proper equivalent expression, the specification gives details to work around some corner cases of top-level nodes and parentless nodes, see section 5.5.3 of [18].

- Check if the result R contains N.
- Result is *true*, which means the contained template will be processed.

Next, the processor will do the same for the next element from the expression `"/list/book/title"`, which is the second book's `title` element:

- `N` is set to the second `title` element in the same way
- Again, we evalute the equivalent expression `root(.)//(book[1]/title)`. The result `R` is again the first `title` element.
- Check if the result `R` contains `N` (which is now the *second* `title`).
- Result is *false*, which means the contained template will not be processed and the next template based on priority will be checked.
- The equivalent expression for the next template is `root(.)//(title)`. The result `R` is now every `title` element from the source.
- Check if the result `R` contains `N` (still the *second* `title`).
- Result is *true*, the processor will evaluate the second template from our example (the one with `match="title"`).

The above approach is a definitive approach to determining whether a template's pattern matches a given node. Certain corner-cases for parentless nodes are given special treatment though. For instance, if a node does not have a parent, a pattern like `title` will still match this node, even though it would officially be expanded into `child::title` by the XPath rules. In the specification, these first axes of the path are called `child-or-top`, `attribute-or-top` and `namespace-or-top` and they work as one might expect from the names: they either match a child node, or the top node (that is, the node that is at the root of the tree).

5.1. Reading from the left and a note on performance

Processors won't use the equivalent expression approach internally, since that would mean going over the whole tree for each pattern time and time again. Instead, processors likely use the information of the current node that is readily available without moving away from the node, or having to browse the children or descendants, where possible.

As briefly mentioned in the previous section, the allowed axes and steps to be taken in a pattern are chosen such that it is only ever needed to traverse the *ancestor-or-self* axis of the current node. This allows for virtually an `O(1)` performance with respect to the size of the whole tree (technically, it would be `O(1)` best case and `O(n)` worst case, where `n` is the depth of the tree, not the size of the tree, but since most trees have a limited depth, this is irrelevant in almost all cases).

To do this, patterns that match element nodes, which are the most common type of pattern, are considered *right-to-left*. That is, if you remove the predicates, the right-most path expression is first evaluated. This is typically fast, because

most patterns will have name-tests or type-tests at the right-most position and all a processor needs to to is to check if the name of the current node, and its type, match the current pattern.

The same process is applied recursively to the next step in the pattern. For a pattern such as `book/author/surname`, the algorithm is typically as follows [34] (see also [9]):

- Test if the current node is an element;
- If *yes*, test if the QName of the current node is `surname`;
- If *yes*, test if the parent has the QName `author`;
- If *yes*, test if the next parent has the QName `book`;
- Apply the predicates, either left-to-right or right-to-left, depending on existing keys, optimizations and performance characteristic per predicate[35].

The method holds well for XSLT 1.0 and 2.0, but for XSLT 3.0, it becomes a little more complex. The reason for this is that in XSLT 1.0 and 2.0 a processor only needs to check the parent axis, and with the exception of `//` expressions, does not need to do any backtracking. Furthermore, overlap is not possible (every step will move up at least one level on the ancestor axis).

In XSLT 3.0, the new axes `descendant(-or-self)` and `self` allow overlap and require a different approach to counting with respect to predicates. Add to that further complexity introduced by parenthesized patterns such as `(a//b)/(c//d)`, patterns with `except` and `intersect`, and combinations like `(* except foo/bar)/zed` which leads to multi-level decision tree, with each having a certain set of backtracking.

Still, patterns are processed *right-to-left*, similar to the original. And if performance is important, or you fear your patterns are slowing the processor down, following the expression from right-to-left through matching and non-matching nodes is a good exercise in finding out bottlenecks. For instance, say your pattern is `(descendant::node() intersect descendant::book//author)//name`, the processor needs to do a lot to calculate the intersection of all the descendant nodes. In this case, rewriting it like `descendant::book//author//name`, may already yield a better performance. And since we arent counting on the descendant axis, this is equivalent to `book//author//name`.

Another sure sign where the processor may require too much backtracking[36] is with overlapping axes. If you have multiple `//` and/or multiple `descendant(-`

[34]Each processor will likely have its own optimization, but this approach is as good as any to understand the general principle behind most pattern matching algorithms.

[35]An optimizing processor will likely process a predicate like `[1]` or `[@foo]` before it will process expensive predicates like `[//x[preceding-sibling::y]]`.

[36]The principal of backtracking in patterns is similar to backtracking as used in regular expressions, and can be similarly detrimental to the performance of the pattern, and other than with regexes, greediness cannot be controlled.

`or-self)` axes in your path, consider analyzing whether you can rewrite it without these paths. For instance, say you have `descendant::a/descendant::b`, but you know that these are either one or two ancestors away from each other, you can simplify this, and likely speed up, by writing `a/(* | */*)/b`. This kind of optimization did not exist in XSLT 2.0 and can prove quite powerful in practice.

6. Writing patterns

This section explores some patterns that are now possible in XSLT 3.0 that weren't that easy or typical in XSLT 2.0.

6.1. Matching every node

For backward compatibility reasons, the expression `node()` without an explicit axis does *not* match every node. It only matches nodes that can be a child, that is, that would, in any other position, match the XPath expression `child::node()`: element, text, comment and processing instruction nodes. It does *not* match attributes, namespace or document nodes.

Since, in most scenarios, programmers have a special template for the top document node, and are not interested in processing namespace nodes specifically, matching *every* node in root-position or any other position is not often a requirement. However, processing attribute nodes is quite common. A typical pattern for processing attribute nodes and any other node (except namespace and document nodes) is `match="node() | @*"`, or more explicitly, `match="node() | attribute::node()"`.

To truly match any node, several expressions can be used. In XSLT 2.0, such expression would look something like `match=" / | node() | @* | namespace-node()"`[37]

. In XSLT 3.0, you have more freedom over this because of the new pattern language features; all the following expressions match *any node*:

- `.[self::node()]`
- `.[. instance of node()]`
- `document-node() | node() | attribute() | namespace-node()`
- `self::node()`
- `descendant-or-self::node()`, but this is less self-explanatory than the previous choice.

[37]This wouldn't, however, match parentless namespace nodes, this was an omission in XSLT 2.0 and has been rectified in XSLT 3.0.

6.2. Matching the new axes

In XSLT 2.0, only the child and attribute axes were available, and indirectly the descendant axis through `x//y`, but as explained above, technically the `y` nametest is still on the child axis. This changes in XSLT 3.0 with the addition of all forward axes to be available at the top-level of a pattern. Their meaning is the same as in XPath, but as a refresher, here are all axes and their invluence on the matching behavior of the pattern:

- `descendant::x`, matches `x` at any depth in the tree, except if it is a root node. Position and size are those of the descendant axis that match `x` in the current selection.
- `descendant-or-self::x`, matches `x` at any depth and when it is itself `x`. Position and size are those of the descendant-or-self axis that match `x`, meaning the self-node has position 1, and the rest is the same as the descendant axis position & size +1.
- `self::x`, matches `x` on the self axis. Position and size are always 1, if `x` is matched.
- `namespace::x`, matches namespace nodes `x`, position is implementation-dependent, but stable during a transformation, and size is the number of namespaces that match `x` in the current selection.

There's little use in using these axes on the first step in a path expression, *unless* position is important in the predicate. Using these new axis, it become much easier to count towards the descendant(-or-self) axis for position and size as it was in XSLT 2.0, where this wasn't directly possible. Example, consider the following input document:

```
<head>
    <div>
        <p>The quick brown fox</p>
    </div>
    <div>
        <p>jumps over</p>
    </div>
    <div>
        <p>the lazy dog</p>
    </div>
</head>
```

A pattern like `match="//p[last()]"` would match every `p` in this input document, because it counts towards the child axis just like in XSLT 2.0, but this is probably not what the user intended. If you want to get the last paragraph only, you can match that now directly by using `match="/descendant::p[last()]"`, which will only match the last paragraph, regardless whether it is preceded by

other elements. The leading / is required to force counting descendants from the root.

One possible XSLT 2.0 equivalent would be to solve this in the `xsl:apply-templates` select expression, or by using a more complex expression like `match="div[last()]/p"`, which is also much harder for a processor to optimize because it requires evaluation of `div` children each time it encounters a `p`[38].

6.3. Matching nodes with or without a parent

It is quite common to have intermediate trees that are elements, or other node kinds, without a document node at their root. For instance, suppose you have `<xsl:variable select="copy-of(para)" />`, all `para` elements in this sequence are without parent. Similarly, if you have something like the following:

```
<xsl:variable name="config" as="element()*">
    <source ip="123.43.22.3" />
    <protocol type="odbc" />
    <port>1433</port>
</xsl:variable>
```

then the three elements here, `source`, `protocol` and `port` will not have a document node as their parent. Therefor, writing `match="/source"` will not match the `source` element.

In many cases this is not a problem, as simply omitting the `"/"` in this case will match `source`. The following patterns can be used if you need to match nodes with, or without a parent:

- Child of a document node: `/nodename`.
- Rooted at a document node, at any level: `//nodename`.
- Rooted at an element or document node, at any level: `nodename`.
- Top-level node not-rooted at a document node: `nodename[not(parent::document-node())]`.
- Top-level node of any kind: `nodename[not(parent::node())]`.
- Node at any level, not-rooted at a document node: `nodename[root()[not(self::document-node())]]`.
- Node at any level, except root node, regardless whether the root is a document node or not: `descendant::nodename`.
- Node at any level, including root node, regardless whether the root is a document node or not: `descendant-or-self::nodename`. Unless position is impor-

[38]It can be assumed that a processor keeps track of certain sizes and positions, but it cannot keep track of all, and specifically predicates tend to require more processor time than straight paths that have a maximum evaluation time of `O(1)` for all intends and purposes, assuming the hierarchy is not too deep.

tant in your predicate, this behavior can also be reached with just `self::nodename`.

See also the section on `root()` below, which expands on this list a bit.

You may be tempted to write that last and 2nd from last as `nodename[not(/)]` or `nodename[not(/ )]`, however, the XPath expression `/` or `//` must raise an error[39] when the root is not a document node. As a result, such patterns would never match anything, as errors are considered non-matches. To overcome this error, *and* to match nodes that do not have a document node at their root, we need to use expressions that do not start with `/` or `//`. This error is, however, not raised when `//` appears in the middle of a pattern or XPath expression.

In the list above, you can replace `nodename` with any node test or node type test.

6.4. Matching complex patterns through variables

Since XSLT 3.0, you can start a pattern with a (usually global) variable reference. This means that the rest of the pattern will only be a positive match if it is rooted at the same node as the variable. Suppose you need to make several matches that repeat the same first part of the pattern over and over, then you could do something like this:

```
<xsl:variable name="section"
    select="book/contents/(chapter | foreword)//section" />

<xsl:template match="$section/para[1]"> ...
<xsl:template match="$section/footnote">...
<xsl:template match="$section/biblioref">...
```

Using a coding pattern like this allows for better self-documenting code. The one downside of this approach is that template patterns can only reference global variables and parameters. If you need to match multiple documents, or your *initial match selection* is not the same as the *global context item*, you can extend this coding pattern by adding `doc(...)` in front of the expression, assuming you know the document URIs.

This approach is more flexible when used inside `xsl:number` or `xsl:for-each-group`, since you'll have access to all in-scope variables.

6.5. The use-case for root()

Since XSLT 3.0 you can start your pattern with the function `root()`. Inside a pattern that function can only be used without a parameter. It can be useful to match

[39]The error raised is XPDY0050, which is a `treat as` error, because `/` is short for `(fn:root(self::node()) treat as document-node())`, and `//` is short for `(fn:root(self::node()) treat as document-node())/descendant-or-self::node()/`.

from the root of a tree, regardless of whether the root is a document node or something else. Furthermore, the `root()` function always succeeds and doesn't throw an error like `//` or `/` (errors in patterns are hidden and lead to a non-match). Some examples:

- Match any node that is top-most: `root()`.
- Match a specific node that is top-most: `root()[self::nodename]`, or `root()[self::attribute()]`.
- Match any non-top element: `root()/descendant::nodename`.
- Count the descendant axis from the top: `root()/descendant::para[3]` will select one, and only one para element that is the third such element from the root of the tree. Conversely, note that `descendant::para[3]` will select each `para` element that is the third such descendant from some ancestor[40], and that the XSLT 2.0 style `//para[3]` will only select the third *child* `para` element.
- Match the top-most element, whether it is parentless, or has a document-node as root: `root()/ descendant-or-self::*[1]`, or alternatively, `root()/(self::* | *)[1]`.

In general, it is good practice to use `root()` instead of `/` or `//` so that your code is resilient for sources that are rooted at a document node and the ones that are rooted at something else, like an element node.

The main difference to remember is that if the tree has an element at its root, instead of a document-node, that `root()/x` will select the *child* `x` of that parentless root element, or the root element if there's a document node. If you know this beforehand, you can use the *self* axis if you need to access the parentless root element. If you deal with either parentless root elements, or root elements under a document-node, then use the trick of the last bullet point above to select the highest element in the tree.

6.6. Patterns with doc()

The `doc` function, not to be confused with the `document` function, matches zero or one document nodes, if and only if the given URI matches the document URI of the node that is currently being tested. Just like the other so-called *rooted patterns*, the `doc` function can only appear at the start of an expression[41]. Some examples:

- Match nodes inside a specific document only: `doc('source.xml')/paper/section`

[40]Note that if the root element is parentless and can be `para`, you could also write `root()/descendant-or-self::para[3]` to include that element in the counting.
[41]See for an exception to this rule, parenthesized expressions.

- Equivalent XSLT 2.0 pattern would be: `/paper/section[doc('source.xml')]`, but this expression is harder to optimize for a processor.
- Match nodes that have the same URI as the current XSLT stylesheet: `doc('')/xsl:stylesheet/xsl:param`[42].

6.7. Patterns with except and intersect

Not possible in XSLT 2.0, but now allowed in XSLT 3.0: patterns with `intersect` and `except`. These patterns work essentially the same as their XPath equivalent. That is, `A intersect B`, where A and B are themselves patterns, will only match if the current node is in both `A` and `B`. And `A except B` only matches if the current node is in `A`, but not in `B`.

Some examples:

- `self::node() except title` matches every node, but not `title`.
- `para/descendant::*[4] intersect child::*[1]` matches elements that are the fourth descendant under `para` *and* are the first child element of any other element.
- `*:para except ns:para` matches all `para` elements in any namespace, except the ones in the `ns` namespace.
- `* except (foo | bar)` matches all elements, except `foo` or `bar`.
- `* except foo except bar` matches all elements, except `foo` or `bar`.
- `(node() | @*) except * except foo`, matches all nodes, except elements. The last part, `except foo` is irrelevant, because `except` expressions are grouped left-to-right. Meaning, this can be read as `((node() | @*) except *) except foo`, and the first part already eliminates all elements. See next bullet for a workaround.
- `(node() | @*) except (* except foo)`, matches all nodes, except all elements, except for the element `foo`.
- `*[@age] intersect *[@name]`, matches all elements that have both a `name` and an `age` attribute. Another way of writing this is by using two adjacent predicates: `*[@age][@name]`.

6.8. Root level parenthesized patterns

It seems such a small change, allowing parens, but is opens up a lot of creative and useful patterns that were much harder to express in XSLT 2.0. Originally, the

[42]This works, because the empty string is a relative URI that will be expanded using the rules for `resolve-uri`, which means that it has the same URI as the containing document, in this case the XSLT document where the pattern appears.

intention of adding this feature was to allow such parenthesized expressions at the root level of the pattern. That is, `"(foo | bar)[2]"`, or `"* except (p | para)"`. The official syntax, however, makes it legal to also write sub-expressions as part of a path expression, that is, expressions such as `"chapter/(para | p)/text()`. These will be explored in the next section.

Parenthesized patterns open up the following use-cases:

- Position and size of grouped patterns, including the axis. The pattern `chapter/descendant::p[3]` will select every `p` that is a third descendant of `chapter`. But `(chapter/descendant::p)[3]` will first group all of `chapter/descendant::p`, and of that set, it takes the third `p`, this will likely select only one node, unless `chapter` is nested in itself in the source document.
- Position and size of union patterns. Suppose your document paragraphs defined as `span`, `p` and `div` elements, then you can apply predicates on the combination of these elements, for instance, `(span | p | div)[@class='x'][position() > 1]`. This will select any `span`, `p` or `div` that is not the first `span`, `p` or `div` counted from its parent.
- Matching over multiple documents: you can use parens with root level functions, this allows you to write something like `(doc('a.xml') | doc('b.xml'))//section`, which will only apply to documents `"a.xml"` and `"b.xml"`, but not others.
- Top-level subexpressions. This adds to the expressiveness of mixing operators `except`, `intersect` and `union` in patterns, where the operand can be parenthesized. For instance, `* except (para | p)` will match all elements except `para` or `p`.
- Treat union expressions as a single expression. By default, a template with top-level expression that includes `union` or `|` is split up in multiple matching templates. Each of these templates will have its own priority based on that pattern. If you have `match="div | p/span"`, it will be split in one template with `match="div"`, with priority `0.0` and one template with `match="p/span"`, with priority `+0.5`. In most cases this is not problematic, but you can overcome this by writing `match="(div | p/span)[true()]`, which will have priority `+0.5` for both `div` and `p/span`. The added predicate is necessary, because the specification requires redundant outer parameters to be removed before assigning the priority; the predicate prevents that from happening.
- Explicitly counting the descendant axis from an anchor. In cases where you may have overlapping nodes (like a `section` within a `section`, or a `div` within a `div`), and you want to find the Nth node counted from the top-level of such overlapping nodes, you cannot simply do `section/descendant::para[2]`, because that will restart counting from each `section`.

Instead, you can use `root()` to *anchor* the counting: `//(* except section)/descendant::para[2]`

Note that it is not possible to use parentheses with *predicate patterns*. It is therefore illegal to write `"(.[. = 't'] | .[@foo])"`. The reason for this is simple: this prevents patterns that match only nodes and patterns that match anything (predicate patterns) to be mixed.

6.9. Parenthesized steps in patterns

As briefly explained in the previous section, the specification allows you to parenthesize steps. That means, given `a/b/c`, any of the steps `a` or `b` or `c` can be parenthesized. Each of these parenthesized steps can contain a full pattern (but not a *predicate pattern*).

Suppose you want to match a path on a child of `para` and `p` at the same time. In XSLT 2.0, this could be written with predicates like `*[self::para | self::p]/span`. Predicates are, however, comparatively hard to optimize efficiently by processors. A more performant pattern expression in XSLT 3.0 would be `(para | p)/span`.

At the moment of this writing, not all processors support this type of pattern natively, even though it is part of the XSLT 3.0 specification. An exception is Exselt [7], which does allow parenthesized step expressions, and Saxon [10], but the latter currently only if the step in parentheses is a simple, single expression or step. Some more examples:

- `chapter/(* except section)/para` will match all `para` elements that have any parent, except `section`, and a grand-parent `chapter`.
- `html//(div | p)/descendant::span`, will match all `span` that are under a `div` or `p` element.
- `list/(* | */* | */*/*)/listitem` will only match `listitem` elements that are two, three or four levels deep under `list`. An equivalent variant is relatively hard to achieve here, but typically it is solved in XSLT 2.0 with a predicate like this: `list//listitem[count(ancestor::*) le 3]`, but here the ancestor axis would include `list` and ancestors before that, and more complex code is needed. The parenthesized pattern is a much easier solution.
- `a/(b/c | e/f/g)/j` will match a path `a` followed by one of the paths in the parens, followed by `j`. This form is quite powerful in writing deterministic patterns where you want to match over several sub-paths.

6.10. Disjunctive patterns with parenthesized rooted steps

Since a parenthesized step in a pattern can contain any pattern, it can itself start with a *rooted step*, that is, a step that starts with `//`, `/`, `id()`, `doc()`, `root()`,

`element-with-id()`, `key()` or a variable reference. Such a step has the effect of *breaking out* of the tree, because the rooted step will go to the root of the tree. Anything before that must be in the tree, but not necessarily on the same ancestor path. Anything after it behaves like a normal path expression in a pattern.

This type of patterns is not supported by any processor that I know of, though it is part of the specification. This may be because of the complexity of matching it efficiently, as for the match to be evaluated, often the whole tree will need to be evaluated. A subset of these expressions is supported by Exselt [7] at the moment: those where all nodes exist on the ancestor axis of the current node.

A pattern becomes a *disjunctive pattern* if at least one step is a *rooted step* and the rooted step is not the first step (parameterized or not). A pattern like `(/root | /head)/footer` (a `footer` with parent `head` or `root`) is not disjunctive, as all path segments are still on the same ancestor axis, the left-most step being the root step. But once the root step is not the left-most step, it becomes disjunctive. If we were the previous expression as `footer/(/root | /head)` it would match `head` or `root`, but only if anywhere in the tree there's also a `footer` element, hence the term *disjunctive*: it breaks the common rule of patterns that all steps must be on the ancestor axis.

To read a pattern like this, anything to the left of a rooted step should be considered as an equivalent predicate that searches the whole tree. Such a rewrite doesn't always hold, but in the general case it suffices. For instance, `para/(/root/div)` can be rewitten as `/root/div[root()//para]` for most cases. The right-most part after the rooted step still behaves like a normal pattern, that is, the right-most step still has to match the current node, as can also be seen in the equivalent pattern.

Some other examples of this type of pattern, including the rough equivalent:

- `section/chapter/(/)` will select the document node, provided that that document has, at any level, a `chapter` with a parent `section`. This is broadly equivalent to `(/)[//section/chapter]`
- `section/(/root)/(/)//chapter` will match if the current node is `chapter`, has a top-level element `root` and has an element `section` at any level. The broadly equivalent pattern is `//chapter[/root][//section]`.
- `(/)/(root | start)/(/)/comment()` matches a comment node that belongs to a document node that has a top-level element of either `root` or `start`. The broadly equivalent pattern is `comment()[/root | /start]`, which is arbuably easier to read.
- `div/($someVar)//*`. Matches any node in `$someVar`, provided that it also contains `div` at any level. The broadly equivalent pattern is `$someVar[//div]//*`.
- `doc('a.xml')/(id('b12'))` will match an element with id `'b12'`, provided the document being applied over has relative URI `"a.xml"`.

- `id('b12')/(id('b13'))` will match any element that has two ID attributes, `'b12'` and `b13'`.
- `(//doc)/(//chapter)/(//section)//endnote` will match any `endnote`, provided the document contains at least one element `section`, `chapter` and `doc` at any position in the document as well.

As can be seen in this short list, such expressions can quickly become hard to read, and in most cases they will have proper equivalent pattern expressions using predicates. Since support in processors is unreliable, at the moment it is better to stay away from such expressions.

The syntax in this section was discussed by the XSLT Working Group and was considered valid, yet sufficiently peculiar to warrant an editorial erratum entry E18, see [17]. In the related bug entry [3], the validity and variants of this behavior were discussed. The source of the peculiarity is that the syntax allows `id(..)/(id(..))` or `div/($var)`, but not `id(..)/id(..)` or `div/$var`. The parentheses appear to be redundant, and in XPath they are, but are required in a pattern to make the steps valid if you want to use a function in something else than the first step.

7. Surprising patterns

As a bonus, let's list a few surprising patterns. Most of them are surprising because they expose subtleties in the pattern or XPath language.

7.1. Single step axes, subtle differences

The differences between several one-step, or almost one-step element tests. Most commonly, one would write simply `match="para"` to match an element named `para`, but what are the differences when you add an axis?

In the following overview, *counts overlapping nodes* means whether or not a positional predicate may match more-than-one node. Consider the following input:

```
<root>
    <para>Some text</para>
    <para>a <para>nested</para> paragraph</para>
</root>
```

Here, the third `para` is nested inside the second `para`. If overlapping nodes are counted, it means that counting can start from different ancestors. For instance, `descendant-or-self::para[2]` will match both the second and the third `para`, because it can start counting descendants from `root` or from any other node, and here, counting from the second `para` will give the second position to the third `para`. To remedy this, you can anchor the counting, for instance by starting the

pattern with a non-ambiguous, non-overlapping node test. In this case, `root/descendant-or-self::para[2]` would match only the second overlapping `para` from `root`.

- `para` and `child::para` are synonymous:
  - with or without parent: *both*,
  - position and size: as the child axis,
  - counts overlapping nodes: *no*.
- `self::para`:
  - with or without parent: *both*,
  - position and size: always `1`,
  - counts overlapping nodes: *no*.
- `descendant::para`:
  - with or without parent: *only with*,
  - position and size: as the descendant axis,
  - counts overlapping nodes: *yes*.
- `descendant-or-self::para`:
  - with or without parent: *both*,
  - position and size: as the descendant-or-self axis,
  - counts overlapping nodes: *yes*.
- `attribute::para`, only matches attributes named `para`:
  - with or without parent: *both*,
  - position and size: as the attribute axis,
  - counts overlapping nodes: *no*.
- `namespace::para`, only matches namespace nodes named `para`:
  - with or without parent: *both*,
  - position and size: as the namespace axis (position is processor-dependent, but stable),
  - counts overlapping nodes: *no*.
- `/para`:
  - with or without parent: *only with*,
  - position and size: typically `1`[43],
  - counts overlapping nodes: *no*.
- `//para`:

[43]It is possible to have a document node with multiple elements in a temporary tree like a variable, but this is comparitively rare. Most documents have only one child element, the root element.

  - with or without parent: *only with*,
  - position and size: as the child axis,
  - counts overlapping nodes: *no*.
- `root()/para`:
  - with or without parent: *both*,
  - position and size: as the child axis, typically `1`,
  - counts overlapping nodes: *no*.
- `root()/self::para` matches a `para` element that has no document node as parent.

7.2. Potentially erroneous patterns

The following patterns should either be avoided, or written in a different way, or are patterns that will never match.

- `/@name` or `/attribute()` never matches an attribute node. If you want to match a top-level (parentless) attribute node, use `root()/self::attribute(name)`.
- `*[local-name() = 'person']`. This pattern is seen a lot *in the wild*, but already since XSLT 2.0, this can be written much better using a *partial wildcard match*[44], in this case as `*:person`. This allows the processor to better optimize matching, and it is easier to read and understand.
- `//elem` is a legal, yet commonly misunderstood pattern, seen in the wild *a lot*. In all but a very few cases, this is exactly the same as just `elem` (without the double slash) and both syntax variants count toward the child axis anyway. The only exception is when you want to distinguish between an `elem` that has a document node or not. A more performant pattern is than `descendant::elem`, or `elem[/]`. Both only succeed if there is a document node, just like `//elem`, but the latter requires the processor to test the whole descendant axis to the root[45].
- `foo/descendant::attribute()` never matches anything. You'd probably want `foo//attribute()`, since `//` allows the next step to use the default axis, which for `attribute()` tests is, well, the attribute axis.

  However, this is not the whole story. If you want to get all attributes of all descendants, and you want to count them, or use positional predicates over the whole set, you can use `foo/(descendant-or-self::*/attribute())`.

[44]A partial wildcard match is a wildcard match where either the namespace with `"*:div"`, or the local name with `"ns:*"`, is the wildcard.

[45]Processors may have this optimized, but even for clarity, if you do need to distinguish between parentless and elements with a document node as parent, then it is better to be explicit.

- `/comment()` matches comments, but only when they appear before or after the root element of a document. This may be deliberate, but if you want to match any comment that is at a root level, you can use `root()/self::comment() | /comment()`.
- `foo[empty(/)]` or any variant with `empty(/)` or `not(/)` will never match anything. The reason is that `/` throws an error when there is no document node at the root of a tree, causing the match to fail silently.

  When you use such expression, the intention was probably to match a node that is not rooted at a document-node. One way of doing that is `foo[empty(root()/self::document-node())]`, which properly only matches when there is no document at the root. Instead of a predicate, you can also use the more readable `root()/self::*/descendant-or-self::foo`.
- `foo[empty(root())]` or any variant with `empty(root())` or `not(root())` will never match anything. The `root()` function always succeeds and returns the root of the tree. It does not determine whether the tree is rooted at a document node.
- `foo[empty(parent::node())]` is a creative way of matching a top-level element `foo` without a parent.
- `(root() except /)//foo` fails always for the same reason as `foo[empty(/)]` fails: `/` throws an error when there is no document node root, and when the root is a document node root, it also returns *false*.
- `node() except /` is unnecessary, `node()` by itself does not match document nodes. See also next point.
- `node()` does not match document nodes, attributes or namespace nodes. It only matches elements, comment nodes, text nodes and processing instruction nodes. It is better written as `self::node()` in XSLT 3.0 or `/ | node() | @*` in XSLT 2.0 if you wish to match any node kind.

7.3. Descendant axis variants as middle step

The descendant axis is often abused, or misunderstood, and that's perhaps partially because in XSLT 2.0 you didn't really have a descendant axis in patterns to begin with. Let's have a look at what variants are now available and how they compare to one another:

- `section//para` or `section//child::para`:
  - matches `para` at any depth,
  - position and size: as the child axis,
  - counts overlapping nodes: *no*.
- `section/descendant::para`:

  - matches `para` at any depth,
  - position and size: as the descendant axis,
  - counts overlapping nodes: *yes*.
- `section/descendant-or-self::para`:
  - matches `para` at any depth (but the `-or-self` is redundant in this example),
  - position and size: as the descendant-or-self axis,
  - counts overlapping nodes: *yes*.
- `section//self::para`:
  - matches `para` at any depth,
  - position and size: always `1`,
  - counts overlapping nodes: *no*.
- `section//descendant::para`:
  - matches `para` at any depth, but this pattern should be avoided, the `//` does not add extra meaning and can lead to significant extra backtracking byt the processor,
  - position and size: as the descendant axis,
  - counts overlapping nodes: *yes*.
- `head//middle//tail//end` is another pattern that is seen a lot *in the wild*. Sometimes such patterns are a necessary evil if the depth of nesting is not known beforehand. But more often than not, there's some knowledge of the source tree structure and the depths are well-known and fixed (and I've seen cases where the user only needed children and not descendants). If that is the case, rewrite it with wildcard steps, something like: `head/*/middle/*/*/tail/end`. Rewriting such patterns to be more deterministic can significantly speed up matching.

7.4. Predicate patterns and other suprising patterns

Last but not least, a few surprising and/or new patterns that are now avaialable in XSLT 3.0. Remember, *predicate patterns* are the ones that start with a `.` and are followed by zero or more predicates.

- `.[. instance of node()]` matches any node, which is notable, considering that `node()` by itself only matches a subset (see item on `node()`).
- `self::node()` matches any node which is notable, considering that `node()` by itself only matches a subset (see also item on `node()`).
- `document-node()` matches the document node, but only because the pattern syntax rules have an explicit exception for this pattern. Normally, it would be

expanded as `child::document-node()`, which would never match anything, but when used in a pattern, it is expanded as `self::document-node()`.

- `namespace-node()` does not match a parentless namespace node in XSLT 2.0, but does so in XSLT 3.0. This is so uncommon, that it doesn't even show up in the list of changes of XSLT 3.0.
- `$var/(/doc/column)`. If a variable is a reference, and not a copy of a node, its ancestors are still accessible through patterns. Suppose you have `<xsl:variable name="var" select="/doc/*/row[1]" />`, then this matching pattern will match element `column`, even though it is a parent of `row` pointed to by the variable.
- `.[. instance of function(*)]` matches any function. Inside the template body, the context item expression `"."` will point to the function. For instance, if the function takes an integer, you can do `<xsl:value-of select=". (42)" />` to get the value of the result of calling the function dynamically.
- `.[. instance of map(*)]` matches any kind of map.
- `.[.('birthday') = 1984]` matches when (a) the context item is a function or map and (b) that function returns `1984` when the argument is `'birthday'`, or (c) if it's a map, and the map returns `1984` for the key pointed to by `'birthday'`. There is no need to explicitly test for whether the item is a function, because if it isn't, an error would be raised, and errors are ignored and considered a non-match.
- `.[. lt 42]` matches any item or node whose atomized value is numeric and less than `42`, or that is untyped (like for nodes) and it can be converted to a numeric value and is less than `42`. It is different from `node()[. lt 42]` in that it matches any kind of item.
- `.[. instance of text() or . instance of xs:string]` matches either text nodes or strings. Explictly does *not* match element nodes that have text content, or atomized values from nodes, because they do not derive from string (they are typically `xs:untypedAtomic`).
- `.[. castable as xs:string]` will match any item that can be cast to a string. This will include nodes, which can be atomized and their value can be cast to string. Almost any item can be cast to strings, except function items.
- `some/path/here/(/)` matches a document node that contains the path that precedes it, at any level. Can be used to differentiate between documents based on their contents, and to switch to a different mode depending on that. The XSLT 2.0 equivalent is to use a predicate instead. See also the section on disjunctive patterns.
- `person[current()/@personref = @id]//biblo` matches a `biblio` element that has an attribute `personref` that is equal to the attribute `id` of ancestor ele-

ment `person`. This is an example of the use of the `current()` function, which points to the node currently being matches against. Since XPath changes focus from path to path, this way you can still reference the current element (here: `biblio`) while at another part of the path.

8. Conclusion

Since XSLT 1.0, through XSLT 2.0 and now in XSLT 3.0 a lot has changed when it comes to patterns. We've seen added root functions like `doc()` and `root()`, parenthesized expressions, adding a lot of extra power to patterns and better alignment with XPath, like allowing patterns to use `except` and `intersect`. Among the biggest changes is perhaps the feature to match any kind of item through *predicate patterns* that start with `..`

Related to patterns, the addition of statically declaring several properties of modes through `xsl:mode` adds structural clarity to modes and less chance of typos when using `declared-modes="yes"`. The vast extension of six types of build-in templates makes several scenarios easier to write and requires less code than with, say the modified identity template.

We've also seen that not all features are currently fully supported by all processors, though most of it is, except for some fringe cases. Hopefully in the near future we will see full support in all XSLT 3.0 processors, such that these powerful features can become available to everyone.

With more awareness of subtleties of pattern matching, especially the ones shown in the last section, you will be able to write more robust stylesheets, with hopefully less surprises and a better understanding of the underpinnings of pattern matching.

Bibliography

[1] *Streaming Design Patterns or: How I Learned to Stop Worrying and Love the Stream*. 10.14337/XMLLondon14.Braaksma01. XML London 2014 proceedings, pp24–52, https://doi.org/10.14337/XMLLondon14.Braaksma01. Abel Braaksma. 2014.

[2] *XSLT 3.0 Streaming for the masses*. XML Prague 2014 proceedings, pp29–80, http://archive.xmlprague.cz/2014/files/xmlprague-2014-proceedings.pdf. Abel Braaksma. 2014.

[3] *Patterns like a/(id('x')) are allowed by the syntax* . https://www.w3.org/Bugs/Public/show_bug.cgi?id=30229 (archive link). Abel Braaksma and Michael Kay. 2013.

[4] *Stylesheet Modularity in XSLT 3.0*. Presented at XML Amsterdam 2013 http://www.xmlamsterdam.com/pdf/2013/2013-michaelhkay-ansterdam.odp/[46] (only direct link is still available, it cannot be found from the homepage http://

www.xmlamsterdam.com/). I saved a copy in the Wayback Machine so it is available for the foreseeable future.. Michael Kay. 2013.

[5] *Streaming in the Saxon XSLT Processor*. XML Prague 2014 proceedings, pp81–102, http://archive.xmlprague.cz/2014/files/xmlprague-2014-proceedings.pdf. Michael Kay. 2014.

[6] *Analysing XSLT Streamability.. Presented at Balisage: The Markup Conference 2014, Washington, DC, August 5 - 8, 2014.* 10.4242/BalisageVol13.Lumley01. Proceedings of Balisage: The Markup Conference 2014. Balisage Series on Markup Technologies, vol. 13 (2014)https://doi.org/10.4242/BalisageVol13.Lumley01. John Lumley. 2014.

[7] *Exselt, a concurrent streaming processor*. http://exselt.net. Abel Braaksma.

[8] *XSLT and XPath On The Edge*. 0-7654-4776-3. Jeni Tennison. 2001.

[9] *Anatomy of an XSLT processor*. https://www.ibm.com/developerworks/library/x-xslt2/index.html. Michael Kay. 2001.

[10] *Saxon XSLT processor*. http://saxonica.com. Michael Kay.

[11] *XSLT Grouping techniques*. http://gandhimukul.tripod.com/xslt/grouping.html. Mukul Gandhi, Jeni Tennison, Michael Kay, and and others. 2009.

[12] *XML Path Language (XPath) 3.0, W3C Recommendation 08 April 2014*. http://www.w3.org/TR/xpath-30. Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson.

[13] *XML Path Language (XPath) 3.1, W3C Recommendation 21 March 2017*. http://www.w3.org/TR/xpath-31. Jonathan Robie, Michael Dyck, and Josh Spiegel.

[14] *XPath and XQuery Functions and Operators 3.0, W3C Recommendation 08 April 2014*. https://www.w3.org/TR/xpath-functions-30. Michael Kay.

[15] *XPath and XQuery Functions and Operators 3.1, W3C Recommendation 21 March 2017*. https://www.w3.org/TR/xpath-functions-31. Michael Kay.

[16] *XSL Transformations (XSLT) Version 2.0, W3C Recommendation 23 January 2007*. http://www.w3.org/TR/xslt20. Michael Kay.

[17] *Draft Errata for XSL Transformations (XSLT) Version 3.0, 20 February 2019*. https://htmlpreview.github.io/?https://github.com/w3c/qtspecs/blob/master/errata/xslt-30/html/xslt-30-errata.html, the source and reports being now at https://github.com/w3c/qtspecs/tree/master/errata/xslt-30 . Michael Kay.

[46] http://www.xmlamsterdam.com/pdf/2013/2013-michaelhkay-ansterdam.odp

[18] *XSL Transformations (XSLT) Version 3.0, W3C Recommendation 8 June 2017*. http://www.w3.org/TR/xslt-30. Michael Kay.

A Proposal for XSLT 4.0

Michael Kay
Saxonica
<mike@saxonica.com>

Abstract

This paper defines a set of proposed extensions to the XSLT 3.0 language [18], suitable for inclusion in version 4.0 of the language were that ever to be defined.

The proposed features are described in sufficient detail to enable the functionality to be understood and assessed, but not in the microscopic detail needed for the eventual language specification.

Brief motivation is given for each feature. The ideas have been collected by the author both from his own experience in using XSLT 3.0 to develop some sizable applications (such as an XSLT compiler: see [4], [3]), and also from feedback from users, reported either directly to Saxonica in support requests, or registered on internet forums such as StackOverflow.

1. Introduction

The W3C is no longer actively developing the XSLT and XPath languages, but this does not mean that development has to stop. There is always the option of some other organisation taking the language forward; the W3C document license under which the specification is published [1] explicitly permits this, though use of the XSLT name might need to be negotiated.

This paper is a sketch of new features that could be usefully added to the language, based on experience and feedback from users of XSLT 3.0.

XSLT 3.0 (by which I include associated specifications such as XPath 3.1) introduced some major innovations [18]. A major theme was support for streaming, and by and large that aspect of the specification proved successful and complete; I have not felt any need to propose changes in that area. Another major innovation was packages (the ability to modularize a stylesheet into separate units of compilation). I suspect that there is room for polishing the spec in this area, but to date there has been relatively little feedback from users, so it is too early to know where the improvement opportunities might lie. The third major innovation concerns the data model, with the introduction of maps, arrays, JSON support, and higher-order functions, and it is in these areas that most of the proposals in this

[1] See https://www.w3.org/Consortium/Legal/2015/doc-license

paper fall, reflecting that there has been signficant user experience gained in these areas.

Some of this user experience comes from projects in which the author has been directly involved, notably:

- Development of an XSLT compiler written in XSLT, reported in [4] and [3]. The resulting compiler, at the time of publication of this paper, is almost ready for release.
- Development of an XSD validator written in XSLT, reported in [2] (The project as described was 90% completed, but the code has never been released).
- An evaluation of the suitability of XSLT 3.0 for transforming JSON files, reported at XML Prague [1].

These projects stretched the capabilities of the XSLT language and in particular involved heavy use of maps for representing data structures.

Other feedback has come from users attempting less ambitious projects, and typically reporting difficulties either directly to Saxonica or on internet forums such as StackOverflow.

The paper is concerned only with the technical content of the languages, and not with the process by which any new version of the standards might be agreed. In practice XSLT development is now being undertaken by only a small handful of implementors, and therefore a more lightweight process for agreeing language changes might be appropriate.

The proposal involves changes to the XPath language and the function library as well as to XSLT itself. In this paper, rather than organise material according to which specification is affected, I have arranged it thematically, so that the impact of related changes can be more easily assessed. I have also tried to organise it so that it can be read sequentially; I try never to use a new feature until it has been introduced.

2. Types

Types are fundamental to everything else, so I will start with proposed modifications to the type system.

XSLT 3.0 (by which I include XPath 3.1) enriches the type system with maps and arrays, which greatly enhances the power of the language. But experience has shown some limitations.

2.1. Tuple types

Maps in XSLT 3.0 are often used in practice for structures in which the keys are statically known. For example, a complex number might be represented as `map{"r": 1.0e0, "i": -1.0e0}`. Declaring the type of this construct as

`map(xs:string, xs:double)` doesn't do it justice: such a type definition allows many values that don't actually represent complex numbers.

I propose instead to allow the type of these values to be expressed as `tuple(r as xs:double, i as xs:double)`.

Note that I'm not introducing tuples as a new kind of object here. The values are still maps, and the set of operations that apply to tuples are exactly the same as the operations that apply to maps. I'm only introducing a new way of describing and constraining the type.

A few details on the specification:

- The field names (here `r` and `i`) are always `xs:string` instances (for a map to be valid against the tuple type definition, the keys must match these strings under the same-key comparison rules). Normally the names must conform to the rules for an `xs:NCName`; but to allow processing of any JSON object, including objects with keys that contains special characters such as spaces, I allow the field names to be arbitrary strings; if they are not `NCName`s, they must be written in quotes.
- If the type allows the value of an entry to be empty (for example `middle` in `tuple(first as xs:string, middle as xs:string?, last as xs:string)` then the relevant entry can also be absent. Values where the entry is absent can be distinguished from those where the entry is present but empty using `map:contains()`, but both satisfy the type.
- The as clause may be omitted (for example `tuple(r, i)`). This is especially useful when tuple types are used as match patterns, where it is only necessary to give enough information to give an unambiguous match. Contrary to convention, the default type for a field is not `item()*` but rather `item()+`: this ensures that a type such as `tuple(ssn)` will only match a map if the entry with key `ssn` is actually present.
- A tuple type may be defined as extensible by adding `,*` to the list of fields, for example `tuple(first as xs:string, middle as xs:string?, last as xs:string, *)`. An extensible tuple type allows the map to contain entries additional to those listed, with no constraints on the keys or values; an inextensible tuple type does not allow extra entries to appear.
- The subtype-supertype relation is defined across tuple types in the obvious way: a tuple type `T` is a subtype of `U` if we can establish statically that all instances of `T` are valid instances of `U`. This will take into account whether `U` is extensible. Similarly a tuple type may be a subtype of a map type: for example `tuple(r as xs:double, i as xs:double)` is a subtype of `map(xs:string, xs:anyAtomicType+)`. By transitivity, a tuple is therefore also a function.
- A processor is allowed to report a static error for a lookup expression `X?N` if it can establish statically that `X` conforms to a tuple type which does not allow an

entry named `N`. For example if variable `$c` is declared with the type `tuple(r as xs:double, i as xs:double)`, then the expression `$c?j` would be a static error. (Note also that `1 to $c?i` might give a static type error, because the processor is able to infer a static type for `$c?i`)

However, a dynamic lookup in the tuple for a key that is not a known field succeeds, and returns an empty sequence. This is to ensure that tuples are substitutable for maps.

- If a variable or function argument declares its required type as a tuple type, and a map is provided as the supplied value, then the map must strictly conform with the tuple type; no coercion is performed. For example if the required type has a field declared with `i as xs:double` then the value of the relevant entry in the map must actually be an `xs:double`; an `xs:integer` will not be promoted.

2.2. Union Types

XSLT 3.0 and XPath 3.1 provide new opportunities for using union types. In particular, it is now possible to define a function that accepts an argument which is, for example, either an `xs:date` or `xs:dateTime`. But this can only be achieved by defining a new union type in a schema and importing the schema, which is a rather cumbersome mechanism.

I therefore propose to allow anonymous union types to be defined inline: for example `<xsl:param name="arg" as="union(xs:date, xs:dateTime, xs:time)"/>`. The semantics are exactly the same as if the same union type were defined in a schema.

The member types must be *generalized atomic types* (that is, atomic types or simple unions of atomic types), which means that the union is itself a generalized atomic type.

2.3. Node types

The `element()` and `attribute()` node types are extended to allow the full range of wildcards permitted in path expressions: for example `element(*:local)`, `attribute(xml:*)`. This is partly just for orthogonality (there is no reason why node types and node tests should not be 100% aligned, and this is one of the few differences), and partly because it is actually useful, for example, to declare that a template rule returns elements in a particular namespace.

This means that patterns such as `match="element(xyz:*, xs:date)` become possible, matching all elements of type `xs:date` in a particular namespace. The default priorities for such patterns are established intuitively: the priority when `foo:*` or `*:bar` is used is midway between the priorities for a full name like `foo:bar`, and the generic wildcard `*`. Since `element(*, T)` has priority 0, while

`element(N, T)` is 0.25, this means the priority for `element(p:*, T)` is set at 0.125.

2.4. Default namespace for types

The XPath static context defines a default namespace for elements and types. I propose to change this to allow the default namespace for types to be different from the default namespace for elements. Since relatively few users write schema-aware code, 99% of all type names in a typical stylesheet are in the XML schema namespace (for example `xs:integer`) and it makes sense to allow these to be written without a namespace prefix. For XSLT I propose to extend the `xpath-default-namespace` attribute so it can define both namespaces, space-separated. (Note however that when constructor functions are used, as in `xs:integer(@status)`, it is the default namespace for functions that applies.)

2.5. Named item types

In a stylesheet that uses maps to represent complex data structures, and especially when these are defined using the new `tuple()` syntax, you quickly find yourself using quite complex type definitions repeatedly on many different variable and function declarations. This has several disadvantages: it means that when the definition changes, code has to be changed in many different places; it fails to capture the semantic intent of the type; and it exposes details of the implementation that might be of no interest to the user.

I therefore propose to introduce the concept of named item types. These can be declared in a stylesheet using top-level declarations:

```
<xsl:item-type name="complex" as="tuple(r as xs:double, i as
xs:double)"/>
```

and can be referenced wherever an item type may appear using the syntax `type(type-name)`: for example `<xsl:param name="arg" as="type(complex)"/>`. Type names, like other names, are QNames, and if unprefixed are assumed to be in no namespace. The usual rules for import precedence apply. Types may be defined with visibility private or final; the definition cannot be overridden in another package.

Named item types also allow recursive type definitions to be created, for example:

```
<xsl:item-type name="binary-tree"
               as="tuple(left as type(binary-tree)?, value as item()*,
right as type(binary-tree)?)"/>
```

This means that item type names (like function names) are in scope within their own definitions. This creates the possibility of defining types that cannot be

instantiated; I suggest that we leave implementors to issue warnings in such cases.

2.6. Type testing in patterns

With types becoming more expressive, and with increasing use of values other than nodes in `<xsl:apply-templates>`, the syntax `match=".[. instance of ItemType]"` to match items by their type becomes increasingly cumbersome. This syntax also has the disadvantage that there is no "smart" calculation of default priorities based on the type hierarchy. I therefore propose to introduce new syntax for patterns designed for matching items other than nodes.

- `type(T)` matches an item of type `T`, where `T` is a named item type. The default priority for such a pattern depends on the definition of `T`, and is the same as that of the pattern equivalent to `T`.
- A pattern in the form `atomic(EQName)`, followed optionally by predicates, matches atomic values of a specified atomic type. For example, `atomic(xs:string)[matches(., '[A-Z]*')]` matches all `xs:string` values comprising Latin upper-case letters.

  Note, this syntax is needed because a bare EQName used as a pattern matches an element node with a given name. Semantically, `atomic(Q)` is equivalent to `union(Q)` (a singleton union).
- Item types in the form `tuple(...)`, `map(...)`, `array(...)`, `function(...)`, or `union(...)` match any item that is an instance of the specified item type.

  In fact, for template rules that need to match JSON objects, a tuple type that names a selection of the fields in the object without giving their types will often be perfectly adequate: for example `match="tuple(ssn, first, middle, last, *)"` is probably enough to ensure that the right rule fires.

  The default priority for these patterns is defined later in the paper.

Any of these patterns may be followed by one or more predicates.

The effect of these changes is that for any `ItemType`, there is a corresponding pattern with the same or similar syntax:

- For the item type `item()`, the corresponding pattern is `.`
- For an item type expressed as an EQName `Q`, the corresponding pattern is `atomic(Q)`
- For an item type written as `type(...)`, `map(...)`, `array(...)`, `function(...)`, `tuple(...)`, or `union(...)`, the item type can be used as a pattern *as is*
- For an item type written as a `KindTest` (for example `element(P)` or `comment()`), the item type can be used as a pattern *as is* (this is because every `KindTest` is a `NodeTest`). There is one glitch here: as an item type, `node()`

matches all nodes, but as a pattern, it does not match attributes, namespace nodes, or document nodes. I therefore propose to introduce the syntax `node(*)`, which is defined to match any node (of any node kind) whether it is used as a step in a path expression or as the first step in a pattern.

These extensions to pattern syntax are designed primarily to make it easier to process the maps that result from parsing JSON using the recursive-descent template matching paradigm. For example, if the JSON input contains:

```
{ "ssn": "ABC12357", "firstName": "Michael", "dateOfBirth": "1951-10-11"}
```

then this can be matched by a template rule with the match pattern

```
match="tuple(ssn as xs:string, dateOfBirth, *)[?dateOfBirth castable as
xs:date]"
```

A possible extension, which I have not fully explored, is to allow nested patterns within a tuple pattern, rather than only allowing item types. For example, this would allow the previous example to be written:

```
match="tuple(ssn as xs:string, dateOfBirth[. castable as xs:date], *)"
```

Indeed, a further extension might be to allow a predicate wherever an item type is used, for example in the declaration of a variable or a function argument. While this is powerful, it creates considerable complications because of the fact that predicates can be context-sensitive

2.7. Function Conversion Rules

The so-called function conversion rules define how the supplied arguments to a function call are converted (where necessary) to the required type defined in the function signature. In XSLT (though not XQuery) the same rules are also used to convert the supplied value of a variable to its required type.

The name "function conversion rules" is rather confusing because the thing being converted is not necessarily a function, nor is the operation exclusively triggered by a function call, so my first proposal is to rename them "coercion rules". This is consistent with the way the term "function coercion" is already used in the spec.

The coercion rules are pragmatic and somewhat arbitrary: they are a compromise between the convenience to the programmer of not having to do manual conversion of values to the required type, and the danger of the system doing the wrong conversion if left to its own devices.

I propose to change the coercion rules so that where the required type is a derived atomic type (for example `xs:positiveInteger`), and the supplied value after atomization is an instance of the same primitive type (for example the `xs:integer` value 17) then the value is automatically converted -- giving a dynamic error, of course, if the conversion fails. Currently no-one uses the

derived atomic types such as `xs:positiveInteger` in a function signature because of the inconvenience that you then can't supply the literal integer 17 in a function call. This change brings atomic values into line with the way that other values such as maps work: if a function declares the required type of a function argument as `map(xs:string, xs:integer)` then the caller can supply any map as an argument, and the function calling mechanism will simply check that the supplied map conforms with the constraints defined by the function for what kind of map it will accept; there is no need for the caller to do anything special to invoke a conversion.

(I would have preferred a more radical change, whereby atomic values are labelled only with their primitive type, and not with a restricted type. So the expression 17 instance of `xs:positiveInteger` would return true, which is probably what most users would expect. However, I think this change would probably be too disruptive to existing applications.)

I also propose to make a change to the way function coercion works. Function coercion applies when you supply a function `F` in a context where the required type is another function type `G`. The current rule is that this works provided that `F` accepts the arguments supplied in an actual call, and returns a value allowed by the signature of `G`; it doesn't matter whether `F` is capable of accepting everything that `G` accepts, so long as it accepts what is actually passed to it.

Currently function coercion fails if `F` and `G` have different arity. I propose to allow `F` to have lower arity than `G`; additional arguments supplied to `G` are simply dropped.

Consider how this might work for the higher-order function `fn:filter`, by analogy with the way it works in Javascript. Currently `fn:filter` expects as its second argument a function of type `$f as function(item()) as xs:boolean`. With this change to function coercion, we can extend this so the declared type is `$f as function(item(), xs:integer) as xs:boolean`. The extended version allows the predicate to accept a second argument, which is the position of the item in the sequence being filtered. But you can still supply a single-argument function; it just won't be told about the position.

The purpose of this change is to allow backwards-compatible extensions to higher-order functions; the information made available to the callback function can be increased without invalidating existing code.

2.8. Static type-checking rules

Some early XQuery developers favoured the use of "pessimistic static type checking", whereby a static type error is reported if any expression is not type-safe. (This is perhaps most commonly seen today in the implementation of XQuery offered with Microsoft's SQL Server database product.) More specifically, pessimistic static type checking signals an error unless the required type subsumes the

supplied type. Experience has shown that pessimistic static type is rather inconvenient for most applications (especially as most applications are not schema-aware). XSLT fortunately steered clear of this area.[2]

The limited ability to perform "optimistic static type checking", whereby a static type error can be reported if the required type and the supplied type are disjoint, has been found to give considerable usability benefits; it is sufficient to detect a great many programming mistakes at compile time, provided that users are diligent in declaring the required types of variables and parameters, but it doesn't force the user to use verbose constructs (such as `treat as`) to enforce compile-time type safety.

I propose some modest changes to allow more obvious errors to be reported at compile time.

- First, I propose to allow a static type error to be reported in the case where the supplied type of an expression can satisfy the required type only in the event that its value is an empty sequence. For example, if the required type is `xs:integer*`, and the expression is a call on `xs:date()`, then it is not currently permitted to report a static error, because a call on `xs:date()` can yield an empty sequence, which would be a valid instance of the required type. In practice this situation is invariably a programmer mistake, and processors should be allowed to report it as such.

- Second, I propose introducing rules that allow certain path expressions (of the form `A/B`) to report an error if it is statically known that the result can only be an empty sequence. If the processor knows the node-kind of A, by means of static type inferencing, then it can report an error if B uses an axis that is always empty for that node kind: so `@A/@B` becomes a static error. (This error is suprisingly common, though it's not usually quite so blatant. It tends to happen when a template rule that only matches attributes does `<xsl:copy-of select="@*"/>`. Of course, this particular example is harmless, so we should reject it only if the stylesheet version is upped to 4.0).

  This ability is particularly useful in conjunction with schema-awareness. Users expect spelling mistakes in element names to be picked up by the compiler if the name used in the stylesheet is inconsistent with its spelling in the schema. Currently the language rules allow only a warning in this case.

[2] I have used the terms *optimistic* and *pessimistic* type checking for many years, but I cannot find any definitions in the literature. By *pessimistic static type checking* I mean what is often simply called *static* or *strict* type checking: a static error occurs if the inferred type of an expression is not a subtype of the type required for the context in which the expression is used. By contrast, I use *optimistic static type checking* to mean that a static error occurs only if the inferred type and the required type are disjoint (they have no values in common); in cases where the inferred type overlaps the required type, code is generated to perform run-time type checking.

- Third, an expression like `function($x){. + 3}` currently throws a dynamic error (`XPDY0002`) because the context item is absent. A strict reading of the XSLT specification suggests that the processor cannot report this as a compile time error (it only becomes an error if the function is actually evaluated). XQuery, it turns out, has fixed this (for named functions, though not for inline functions): it says that the static error XPST0008 can be raised in this situation. I propose changing XPDY0002 to be a type error, which means it can now be statically reported if detected during compilation, not just within function bodies, but in other contexts (such as `<xsl:on-completion>`) where there is no context item defined.

3. Functions

XSLT is a functional language, and version 3.0 greatly increases the role of functions by making them first-class objects and thus allowing higher-order functions. When you start to make extensive use of this capability, however, you start to encounter a few usability problems.

Firstly, the syntax for writing functions starts to become restrictive. You can either write global named functions in XSLT syntax, or local anonymous functions in XPath; neither syntax is particularly conducive to the very simple functions that you sometimes want to use in calls on `fn:filter()` or `fn:sort()`. It is also cumbersome to define a family of functions of different arity allowing some arguments to be omitted. I therefore propose to introduce some new syntax for writing functions.

3.1. Dot Functions

The syntax `.{EXPR}` is introduced as a shorthand for `function($x as item()) as item()* {$x ! EXPR}`. For example, this allows you to sort employees by last name then first name using the function call `sort(//employee, .{lastName, firstName})` where you would currently have to write `sort(//employee, function($emp) { $emp/lastName, $emp/firstName })`.

Exprience with other programming languages suggests that a more concise syntax for inline functions greatly encourages their use; indeed, we can imagine non-programmer users of XSLT mastering this syntax without actually understanding the concepts of higher-order functions.

3.2. Underscore Functions

In dot functions, we are limited to a single argument whose value is a single item (because that's the way the context item works). For the more general case, we introduce another notation: the underscore function. By way of an example, `_{$1 + $2}` is a function that takes two arguments (without declaring their type, so

there are no constraints), and returns the sum of their values. This means that a function call such as `for-each-pair($seq1, $seq2, function($a1, $a2) {$a1 + $a2})` can now be written more concisely as `for-each-pair($seq1, $seq2, _{$1 + $2})`.

The arity of such a function is inferred from the highest-numbered parameter reference. Parameter references act like local variable references, but identify parameters by position rather than by name. There can be multiple references to the same parameter, and the function body doesn't need to refer to any parameters except the last (so the arity can be inferred). Parameters go "out of scope" in nested underscore functions.

The change to the function coercion rules means that if your function doesn't need to use the last argument, it doesn't matter that your function now has the wrong arity. For example, in a later section I propose an extension to the `<xsl:map>` instruction that provides an `on-duplicates` callback, which takes two values. To select the first duplicate, you can write <xsl:map on-duplicates="_{$1}"/>; to select the second, you can write <xsl:map on-duplicates="_{$2}"/>. Although the required type is a function with arity 2, you are allowed to supply a function that ignores the second argument.

Nested anonymous functions are perhaps best avoided in the interests of readability; but of course they are permitted. A numeric parameter reference such as $1 is not directly available in the closure of a nested function, but it can be bound to a conventional variable:

```
_{ let $x := $1, $g := _{$1 + $x} return $g(10) }(5)
```

3.3. Default Arguments

I propose to allow a single `<xsl:function>` declaration to define a family of functions, having the same name but different arity, by allowing parameters to have a default value. For example consider the declaration:

```
<xsl:function name="f:mangle" as="xs:integer">
  <xsl:param name="a" as="xs:string"/>
  <xsl:param name="options" as="map(*)" required="no" select="map{}"/>
  <xsl:sequence select="if ($options?upper) then upper-case($a) else
$a"/>
</xsl:function>
```

This declares two functions, `f:mangle#1` and `f:mangle#2`, with arity 1 and 2 respectively, based on whether the second argument is supplied or defaulted.

A parameter is declared optional with the attribute `required="no"`; if the attribute is optional, then its default value can be given with a `select` attribute. In the absence of a `select` attribute, the default value of an optional parameter is the

empty sequence. A parameter can only be optional if all subsequent arguments are also optional.

The single `<xsl:function` declaration defines a set of functions having the same name, with arities in the range `M` to `N`, where `M` is the number of `<xsl:param>` elements with no default value, and `N` is the total number of `<xsl:param>` elements. The construct is treated as equivalent to a set of separate `xsl:function` declarations without optional parameters; for example, an overriding `xsl:function` declaration (one with higher import precedence, or one within an `xsl:override` element) might override one of these functions but not the others.

4. Conditionals

Conditional (if/then/else) processing can be done both in XPath and in XSLT. In both cases, for such a commonly used construct, the syntax is a little cumbersome. I believe that a few minor improvements can be made without difficulty and will be welcomed by the user community.

4.1. The `otherwise` operator

A common idiom in XPath is to see constructs like `(@discount, 0)[1]` to mean: take the value of the `@discount` attribute if present, or the default value 0 otherwise.

There are two drawbacks with this construct: firsly, unless you've come across it before, the meaning is far from obvious; and secondly, it only works if the first value is a singleton, rather than an arbitrary sequence.

I propose the syntax `@discount otherwise 0` as a more intuitive way of expressing this. The expression returns the value of the first operand, unless it is an empty sequence, in which case it returns the value of the second operand.

4.2. Adding `@select` to `<xsl:when>` and `<xsl:otherwise>`

Most XSLT instructions that allow a contained sequence constructor also allow a `select` attribute as an alternative. The `<xsl:when>` and `<xsl:otherwise>` elements are notable exceptions, and I propose to remedy this. For example this instruction:

```
<xsl:choose>
   <xsl:when test="@a=2">
      <xsl:sequence select="17"/>
   </xsl:when>
   <xsl:when test="@a=3">
      <xsl:sequence select="19"/>
   </xsl:when>
```

```
    <xsl:otherwise>
       <xsl:sequence select="23"/>
    </xsl:otherwise>
 </xsl:choose>
```

can be rewritten as:

```
 <xsl:choose>
    <xsl:when test="@a=2" select="17"/>
    <xsl:when test="@a=3" select="19"/>
    <xsl:otherwise select="23"/>
 </xsl:choose>
```

which makes it significantly more readable.

4.3. Adding `@then` and `@else` attributes to `<xsl:if>`

For the `xsl:if` instruction, rather than adding a `select` attribute, I propose to add two attributes, `then` and `else`. If either attribute is present then the contained sequence constructor must be empty. If one attribute is present and the other absent, the other defaults to () (the empty sequence).

This enables a construct like:

<xsl:if test="@a='yes' then="0" else="1"/>

This is likely to be particularly useful for delivering function results, in place of `xsl:sequence`; it will often enable a 2-way `xsl:choose` to be replaced with a 2-way `xsl:if`.

Consider this example from the XSLT 3.0 specification:

```
<xsl:choose>
  <xsl:when test="system-property('xsl:version') = '1.0'">
    <xsl:value-of select="1 div 0"/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:value-of select="xs:double('INF')"/>
  </xsl:otherwise>
</xsl:choose>
```

which can (in all likelihood) be rewritten

```
<xsl:if test="system-property('xsl:version') = '1.0'"
      then="1 div 0"
      else="xs:double('INF')"/>
```

Of course, we could also use an XPath conditional here. But when the expressions become a little longer, many users dislike using complex multi-line XPath expressions (partly because some editors ruin the layout, whereas editors offer good support for XML layout).

For another example, the function given earlier in this paper:

```
<xsl:function name="f:mangle" as="xs:integer">
  <xsl:param name="a" as="xs:string"/>
  <xsl:param name="options" as="map(*)" select="map{}"/>
  <xsl:sequence select="if ($options?upper) then upper-case($a) else
$a"/>
</xsl:function>
```

can now be written:

```
<xsl:function name="f:mangle" as="xs:integer">
  <xsl:param name="a" as="xs:string"/>
  <xsl:param name="options" as="map(*)" select="map{}"/>
  <xsl:if test="$options?upper" then="upper-case($a)" else="$a"/>
</xsl:function>
```

4.4. `xsl:message/@test` attribute

Users have become familiar with the ability to "compile out" instructions using a static `use-when` expression, for example

```
<xsl:message use-when="$debug"/>
```

Currently this only works if `$debug` is a static variable; if it becomes necessary to use a non-static variable instead, the construct has to change to the much more cumbersome

```
<xsl:if test="$debug">
   <xsl:message/>
</xsl:if>
```

I propose that `<xsl:message>` should have a `test` attribute, bringing it into line with `<xsl:assert>`.

Verbose wrapping of instructions in `<xsl:if>` is also seen when constructing output elements, for example one might see a long sequence of instructions of the form:

```
<xsl:if test="in:maturity-date">
   <out:maturityDate>{maturity-date}</out:maturityDate>
</xsl:if>
```

I considered proposing that all instructions should have a `test` or `when` attribute, defining a condition which allows the instruction to be skipped. Having experimented with such a capability, however, I'm not convinced it improves the language.

4.5. Equality Operators

There are in effect four different equality operators for comparing atomic values, all with slightly different rules:

- The "=" operator is implicitly existential, and converts untyped atomic values to the type of the other operand: this leads to curiosities such as the fact that `A = B` being different from `not(A = B)`, and to non-transitivity (if `X` is `xs:untypedAtomic`, then `X = '4'` and `X = 4` can both be true, but `4 = '4'` gives a type error).

- The "eq" operator eliminates the existential behaviour, and converts untyped atomic values to strings. This avoids some of the worst peculiarities of the "=" operator, but the type promotion rules mean that it edge cases, it is still not transitive. The result of the operator is context-sensitive; for example the result of comparing two `xs:dateTime` values can depend on the implicit timezone.

  The comparison performed by `xsl:sort` and `xsl:merge` is based on the "eq" and "le" operators, but NaN is considered equal to itself. The lack of transitivity with edge cases involving mixed numeric types creates a potential security weakness in that it might be possible to construct an artificial input sequence to `xsl:sort` that causes the instruction not to terminate.

- The operator used by the `deep-equal()` function, and also (by reference) by `distinct-values()`, `index-of()`, `fn:sort()`, and `<xsl:for-each-group>`, differs from "eq" primarily in that it returns false rather than throwing an error when comparing unrelated types; it also compares `NaN` as equal to itself. Because it handles conversion among numeric types in the same way as "eq", it is still non-transitive in edge cases, which is particularly troublesome when the operator is used for sorting or grouping. Like "eq", the result is context-sensitive.

- The "same key" operator used implicitly for comparing keys in maps (for example in `map:contains()`) is designed to be error-free, context-free, and transitive. So it always returns false rather than throwing an error; the result is never context-sensitive; and it is always transitive.

It's difficult to sort all of this out while retaining an adequate level of backwards compatibility, but I propose that:

- Type promotion when comparing numeric types should be changed to use the rules of the "same key" operator throughout. In effect this means that all numeric comparisons are done by converting both operands to infinite-precision `xs:decimal` (with special rules for infinity and NaN). This change makes "eq" transitive. Although this creates a minor backwards incompatibility in edge cases, I believe this change can be justified on security grounds; the current rules mean there is a risk that sorting will not terminate for some input sequences. These rules extend to other functions that compare numeric values, for example `min()` and `max()`, but the promotion rules for arithmetic are

unchanged: adding an `xs:double` and an `xs:decimal` still delivers an xs:double.

- All four operators should handle timezones in the way that the "same key" operator does: that is, a date/time value with a timezone is not considered comparable to one without. This change makes the result of a comparison independent of the dynamic context in which it is invoked, which enables optimizations that are disallowed in 3.0 simply because of the remote possibility that the input data will contain a mix of timezoned and untimezoned dates/times.

  This change is perhaps more significant from the perspective of backwards compatibility, and perhaps there needs to be a 3.0-compatible mode of execution that retains the current behaviour.

5. Template Rules and Modes

Template rules and modes are at the heart of the XSLT processing model. The `xsl:mode` declaration in XSLT 3.0 usefully provides a central place to define options and properties for template rule processing. Packages also help to create better modularity. But anyone who has to debug a large complex stylesheet with 20 or more modules knows what a nightmare it can be to find out where a particular bit of logic is located, so further improvements are possible.

5.1. Enclosed Modes

I propose to allow template rules to be defined by using `xsl:template` as a child of `xsl:mode`. An xsl:mode declaration that contains template rules is referred to as an enclosed mode. Such template rules must have no `mode` attribute (it defaults to the name of the containing mode). They must also have no `name` attribute. If a mode is an enclosed mode, then all template rules for the mode must appear within the `xsl:mode` declaration, other than template rules declared using `xsl:override` in a different package. Specifying `mode="#all"` on a template rule outside the enclosed mode is interpreted as meaning "all modes other than enclosed modes". The default mode for `xsl:apply-templates` instructions within the enclosed mode is the enclosing mode itself.

This feature is designed to make stylesheets more readable: it becomes easier to get an overview of what a mode does, and it becomes easier to find the template rules associated with a mode. It makes it easier to copy-and-paste a mode from one stylesheet to another. It means that to find the rules for a mode, there are fewer places you need to look: the rule will either be within the mode itself, or (if the mode is not declared `final`) within an `xsl:override` element in a using package.

To further encourage the use of very simple template rules, I propose allowing `xsl:template` to have a `select` attribute in place of a sequence constructor. This allows for example:

```
<xsl:mode name="border-width" as="xs:integer">
<xsl:template match="aside" select="1"/>
<xsl:template match="footnote" select="2"/>
<xsl:template match="*" select="0"/>
</xsl:mode>
```

A template rule with a `select` attribute must not contain any `xsl:param` or `xsl:context-item` declarations.

5.2. Typed modes

It is often the case that all template rules in a mode return the same type of value, for example nodes, strings, booleans, or maps. This is almost a necessity, since anyone writing an `xsl:apply-templates` instruction needs to have some idea of what will be returned.

I propose therefore that the `xsl:mode` declaration should acquire an `as` attribute, whose value is a sequence type. If present, this acts as the default for the `as` attribute in `xsl:template` rules using that mode. Individual template rules may have an `as` attribute that declares a more precise type, but only if it is a true subtype.

The presence of this attribute enables processors to infer a static type for the result of the `xsl:apply-templates` instruction.

In the interests of forcing good practice, the `xsl:mode/ @as` attribute is required in the case of an enclosed mode.

5.3. Default Namespace for Elements

Anyone who follows internet programming forums such as StackOverflow will know that the number one beginner mistake with XSLT is to assume that an unprefixed name, used in a path expression or match pattern, will match an unprefixed element name in the source document. In the presence of a default namespace declaration, of course, this is not the case.

What's particularly annoying about this problem is that the consequences bear no obvious relationship to the nature of the mistake. It generally means that template rules don't fire, and path expressions don't select anything. Those are tough symptoms for beginners to debug, when they have no idea where to start looking.

It's worth noting that only a minority of documents actually use multiple namespaces, and in those that do, there is rarely any ambiguity in the sets of local names used. It's therefore unsurprising that beginners imagine that namespaces are something they can learn about later if they need to.

The `xpath-default-namespace` attribute in XSLT 2.0 was an attempt to tackle this problem; but unfortunately it only solved the problem if you already knew that the problem existed.

I want to propose a more radical solution:

- Unprefixed element names in path expressions and match patterns should match by local name alone, regardless of namespace; that is, `NNNN` is interpreted as `*:NNNN`.

  This is a radical departure and for backwards compatibility, it must be possible to retain the status quo. My guess is that the vast majority of stylesheets will still work perfectly well with this change.

- The syntax `:local` (with a leading colon) becomes available to force a no-namespace match, regardless of default namespace.

- The option to match by local name can be explicitly enabled (for any region of the stylesheet) by specifying `xpath-default-namespace="##any"`, while the option for unprefixed names to match no-namespace names can be selected by setting the attribute to either a zero-length string (as in XSLT 3.0) or, for emphasis, to `"##local"` (a notation borrowed from XSD).

- The "default default" for `xpath-default-namespace` becomes implementation-defined, with a requirement that it be configurable; implementors can choose how to configure it, and what the default should be. (This includes the option to use the default namespace declared in the source document, if known).

This gives implementors the option to provide beginners with an interface in which unprefixed element names match the way that beginners expect: by local name only. Users who understand namespaces can then switch to the current behaviour if they wish, or can qualify all names (using the new syntax `:name` for no-namespace names if necessary), to make sure that the problem does not arise.

This proposal is also motivated by the challenges posed by the way namespaces are handled in HTML5. The HTML5 specification defines a variation on the XPath 1.0 specification that changes the way element names in path expressions match. The proposal to make unprefixed element names match (by default) by local name alone removes the need for HTML5 to get special treatment.

6. Processing Maps and Arrays

The introduction of maps and arrays into the data model has enabled more complex applications to be written in XSLT, as well as allowing JSON to be processed alongside XML. But experience with these new features has revealed some of their limitations, and a second round of features is opportune.

6.1. Array construction

The XSLT instruction `xsl:array` is added to construct an array.

The tricky part is how to construct the array members (in general, a sequence of sequences). The same problem exists for the square and curly array constructors in XPath, and I propose to solve the problem in the same way.

First I propose a new function

```
array:of((function() as item()*)*) => array(*)
```

which takes a sequence of zero-arity functions as its input, and evaluates each of those functions to return one member of the array. For example

```
array:of((_{1 to 5}, _{7 to 10}))
```

returns the array `[(1,2,3,4,5), (7,8,9,10)]`

(The underscore syntax for writing simple functions – in this case, zero-arity functions – was described earlier in the paper).

For a more complex example,

```
array:of(for $x in 1 to 5 return _{1 to $x})
```

returns the array `[(1), (1,2), (1,2,3), (1,2,3,4), (1,2,3,4,5)]`.

Now I propose an instruction `xsl:array` that accepts either a `select` attribute or a contained sequence constructor, and processes the resulting sequence in the same way as the `array:of()` function, with one addition: any item in the result that is not a zero-arity function is first wrapped in a zero-arity function. For example:

```
<xsl:array select="1 to 5"/>
```

returns the array `[1,2,3,4,5]`; while

```
<xsl:array>
  <a/>
  <b/>
  <c/>
</xsl:array>
```

returns the array `[<a/>, <b/>, <c/>]`, and

```
<xsl:array select="1, 2, 3, _{}, ${4,5,6}"/>
```

returns the array `[1, 2, 3, (), (4,5,6)]`

6.2. Map construction

The `<xsl:map>` instruction acquires an attribute `on-duplicates`. The value of the attribute is an XPath expression that evaluates to a function; the function is called when duplicate map entries are encountered. For example, `on-duplicates="_{$1}"` selects the first duplicate, `on-duplicates="_{$2}"` selects

the last, `on-duplicates="_{$1, $2}"` combines the duplicates into a single sequence, and `on-duplicates="_{string-join(($1, $2), '|')}"` concatenates the values as strings with a separator.

6.3. The Lookup Operator ("?")

In 3.0, the right-hand side of the lookup operator (in both its unary and binary versions) is restricted to be an NCName, an integer, the token "*", or a parenthesized expression.

To provide slightly better orthogonality, I propose relaxing this by allowing (a) a string literal, and (b) a variable reference. In both cases the semantics are equivalent to enclosing the value in parentheses: for example `$array?$i` is equivalent to `$array?($i)` (which can also be written `$array($i)`), and `$map?"New York"` is equivalent to `$map?("New York")` (which can also be written `$map("New York")`).

6.4. Iterating over array members

The lookup operator `$array?*` allows an array to be converted to a sequence, and often this is an adequate way of iterating over the members of the array. But where the members of the array are themselves sequences, this loses information: the result of `array{(1,2,3), (4,5,6)}?*` is `(1,2,3,4,5,6)`.

To make processing such arrays easier, I introduce a new clause for FLWOR expressions: `for member $var in array-expression` which binds `$var` to each member of the array returned by the *array-expression*, in turn.

For example:

```
for member $var in array{(1,2,3), (4,5,6)} return sum($var)
```

returns `(6, 15)`

As with `for` and `let`, I allow `for member` as a free-standing expression in XPath.

Currently the only way to achieve such processing is with higher-order functions: `array:for-each($array, sum#1)`.

We can also consider an XSLT instruction `<xsl:for-each-member>` but the question becomes, how should the current member be referenced? I'm no great enthusiast for yet more `current-XXX()` functions, but stylistic consistency is important, and this certainly points to the syntax:

```
<xsl:for-each-member select="array{(1,2,3), (4,5,6)}">
  <total>{sum(current-member())}</total>
</xsl:for-each-member>
```

6.5. Rule-based recursive descent with maps and arrays

The traditional XSLT processing model for transforming node trees relies heavily on the interaction of the `xsl:apply-templates` instruction and match patterns.

The model doesn't work at all well for maps and arrays, for a number of reasons. The reasons include:

- We don't have convenient syntax for matching maps and arrays in patterns; all we have is general predicates, which are cumbersome to use.
- Because there is no parent or ancestor axis available when processing maps and arrays, a template rule for processing part of a complex structure cannot get access to information from higher in the structure unless it is passed down in the form of parameters. In addition, there is no mechanism for defining a template rule to match a map or array in a way that is sensitive to the context in which it appears.
- There is no built-in template corresponding to the shallow-copy template that works effectively for maps and arrays, allowing the stylesheet author to define rules only for the parts of the structure that need changing
- Template rules always match items. But with a map, the obvious first level of decomposition is not into items, but into entries (key-value pairs). Similarly, with arrays, the first level of decomposition is into array members, which are in general sequences rather than single items.

The following sections address these issues in turn.

6.5.1. Type-based pattern matching

In 3.0 it is possible to use a pattern of the form `match=".[. instance of T]"` to match items by their type. This syntax is clumsy, to say the least. I therefore propose some new kinds of patterns with syntax closely aligned with item type syntax. The following new kinds of pattern are introduced (by example):

- `atomic(xs:date)`
  Matches an atomic value of type `xs:date`.
- `union(xs:date, xs:dateTime, xs:time)`
  Matches an atomic value belonging to a union type.
- `map(xs:string, element())`
  Matches a map belonging to a map type.
- `tuple(first, middle, last)`
  Matches a map belonging to a tuple type.
- `array(xs:integer)`
  Matches an array whose members are of a given type
- `type(T)`
  Matches an an item belonging to a named type (declared using `xsl:item-type`).

In each case the item type can be followed by predicates. For example, strings starting with "#" can be matched using the pattern `atomic(xs:string)[starts-with(., '#')]`, while tuples representing female employees might be matched with the pattern `tuple(ssn, lastName, firstName, *)[?gender='F']`

The following rules are proposed for the default priority of these patterns (in the absence of predicates):

- For patterns corresponding to the generic type `function(*)` the priority is -0.75; for `map(*)` and `array(*)` it is -0.5.

- For atomic patterns such as `atomic(xs:string)`, the priority is 1 - 0.5^N, where N is the depth of the type in the type hierarchy. For example, `xs:decimal` is 0.5, `xs:integer` is 0.75, `xs:long` is 0.875. In all cases the resulting priority is between zero and one.

  `atomic(xs:anyAtomicType)` gets a priority of 0.

  The rule extends to user-defined atomic types.

  The rule ensures that if S is a subtype of T, then the priority of S is greater than the priority of T.

- For union patterns such as `union(xs:integer, xs:date)`, the priority is the product of the priorities of the atomic member types. So for this example, the priority is 0.375.

  Again, this rule ensures that priorities reflect subtype relationships: for example `union(xs:integer, xs:date)` has a lower priority than `atomic(xs:integer)` but a higher priority than `union(xs:decimal, xs:date)`.

  The rule does not ensure, however, that overlapping types have equal priority; for example when matching an integer, the pattern `union(xs:integer, xs:date, xs:time)` will be chosen in preference to `union(xs:integer, xs:double)`. The rules will not, therefore, be a reliable way of resolving ambiguous matches.

- For a specific array type `array(M)`, the priority is the normalized priority of the item type of *M* (the cardinality of *M* is ignored). Normalized priority is calculated as follows: if the priority is *P*, then the normalized priority is *(P +1)/2*. That is, base priorities in the range -1 to +1 are compressed into the range 0 to +1.

- For a specific map type `map(K, V)`, the priority is the product of the normalized priorities of *K* and the item type of *V* (the cardinality of *V* is ignored).

- For a specific function type `function(A1, A2, ...) as V`, the priority is the product of the normalized priorities of the item types of the arguments. The cardinalities of the argument types, and the result type, are ignored.

Enterprising users may choose to exploit the fact that `function(xs:integer)` has a higher priority than `function(xs:decimal)` as a way of implementing polymorphic function despatch.

- For a non-extensible tuple type `tuple(A as t1, B as t2, ...)`, the priority the product of the normalized priorities of the item types of the defined fields.
- For an extensible tuple type `tuple(A as t1, B as t2, ..., *)`, the priority is -0.5 plus (0.5 times the priority of the corresponding non-extensible tuple type).

  This rule has the effect that an extensible tuple type is never considered for a match until all non-extensible tuple types have been eliminated from consideration.

Like the existing rules for the default priority of node patterns, these rules are a little rough-and-ready, and will not always give the result that is intuitively correct. However, they follow the general principle that selective patterns have a higher priority than non-selective patterns, so it's likely that they will resolve most cases in the way that causes least surprise. When things get complex, users can always define explicit priorities.

The existing rules for node patterns often ensure that overlapping rules have the same priority, thus leading to warnings or errors when more than one pattern matches. That remains true for the new rules when predicates are used, but in the absence of predicates, there are many cases where overlapping patterns do not have the same priority.

The most important use case for the new kinds of pattern is to match maps (objects) when processing JSON input, and in this case using tuples that name the distinguishing fields/properties of each object should achieve the required effect, regardless whether extensible or inextensible tuple types are used.

6.5.2. Decomposing Maps

I propose a function `map:entries($map)` which returns a sequence of maps, one per key-value pair in the original map. The map representing each entry contains the following fields:

- key: the key (an atomic value)
- value: the associated value (any sequence)
- container: the map from which this entry was extracted.

That is, the result matches the type `tuple(key as xs:anyAtomicType, value as item()*, container as map(*))`. To process a map using recursive-descent template rule processing, it is possible to use an instruction of the form `<xsl:apply-templates select="map:entries($map)"/>`, and then to process each entry in the map using a separate template rule. The presence of the

`container` field compensates for the absence of an ancestor axis: it gives access to entries in the containing map other than the one being processed. For example:

```
<xsl:template match="tuple(key, value)[?key='ssn']">
   <xsl:if test="?container?location='London'" then="'UK'||?value"
else="'US'||?value"/>
</xsl:template>
```

This makes the immediate context of a map entry available to the called template rule. For more distant context, it is generally necessary to pass the information explicitly, typically using tunnel parameters. (Navigating further back using multiple container steps is feasible in theory, but clumsy in practice.)

An alternative to use of tunnel parameters is to add information to the map being processed: instead of `<xsl:apply-templates select="map:entries($map)"/ >`, you can write `<xsl:apply-templates select="$map:entries($map) ! map:put(., 'country-name': $country)"/>`, and the extra data will then be available in the called templates as `?country-name`.

7. New Functions

In this section, I propose various new or enhanced functions to add to the core function library, based on practical experience. (Other new functions, such as `array-of()`, have been proposed earlier in the paper).

7.1. `fn:item-at`

The function `fn:item-at($s, $i)` returns the same result as `fn:subsequence($s, $i, 1)`. It is useful in cases where the positional filter expression `$s[EXPR]` is unsuitable because the subscript expression `EXPR` is focus-dependent.

7.2. `fn:stack-trace`

I propose a new function `fn:stack-trace()` to return a string containing diagnostic information about the current execution state. The detailed content and format of the returned string is implementation-dependent.

I also propose a standard variable `$err:stack-trace` available within `xsl:catch` to contain similar information about the execution state at the point where a dynamic error occurred.

7.3. `fn:deep-equal` with options

An extra argument is added to `fn:deep-equal`; it is a map following the "option parameter conventions". The options control how the comparison of the two operands is performed. Options should include:

- Ignore whitespace text nodes
- Normalize whitespace in text and attribute nodes
- Treat comments as significant
- Treat processing instructions as significant
- Treat in-scope namespace bindings as significant
- Treat namespace prefixes as significant
- Treat type annotations as signficant
- Treat `is-ID`, `is-IDREF` and `nillable` properties as signficant
- Treat all nodes as untyped
- Use the "same key" comparison algorithm for atomic values (as used for maps), rather than the "eq" algorithm
- Ignore order of sibling elements

7.4. `fn:differences()`

A new function, like `fn:deep-equal()`, except that rather than returning a true or false result, it returns a list of differences between the two input sequences. If the result is an empty sequence, the inputs are deep-equal; if not, the result contains a sequence of maps giving information about the differences. The map contains references to nodes within the tree that are found to be different, and a code indicating the nature of the difference, plus a narrative explanation. The specification will leave the exact details implementation-defined, but standardised in enough detail to allow applications to generate diagnostics.

For example, `fn:differences(<a x='3'/>, <a x='4'/>)` might return `map{0: $1/@x, 1: $2/@x, 'code': 'different-string-value', 'explanation': "The string value of the @x attribute differs ('3' vs '4')"}`

The values of entries 0 and 1 here are references to the attribute nodes in the supplied input sequences.

7.5. `fn:index-where($input, $predicate)`

Returns a sequence of integers (monotonically ascending) giving the positions in the the input sequence where the predicate function returns true.

Example: `subsequence($in, 1, index-where($in, .{exists(self::h1)})` returns the subsequence of the input up to and including the first `h1` element.

Equivalent to

```
(1 to count($input)) [$predicate(subsequence($input, ., 1)]
```

7.6. fn:items-before(), fn:items-until(), fn:items-from(), fn:items-after()

These new higher-order functions all take two arguments: an input sequence, and a predicate that can be applied to items in the sequence to return a boolean.

If `N` is the index of the first item in the input sequence that matches the predicate, then:

- `fn:items-before()` returns items with `position() lt N`
- `fn:items-until()` returns items with `position() le N`
- `fn:items-from()` returns items with `position() ge N`
- `fn:items-after()` returns items with `position() gt N`

7.7. map:index($input, $key)

Returns a map in which the items in `$input` are indexed according to the atomized value of the `$key` function. For example `map:index(// employee, .{@location})` returns a map `$M` such that `$M?London` will return all employees having `@location='London'`.

The `$key` function may return a sequence of values in which case the corresponding item from the input will appear in multiple entries in the index.

7.8. map:replace($map, $key, $action)

If the map `$map` contains an entry for `$key`, the function calls `$action` supplying the existing value associated with that key, and returns a new map in which the value for the key is replaced with the result of the `$action` function.

If the map contains no entry for the `$key`, calls `$action` supplying an empty sequence, and returns a new map containing all existing entries plus a new entry for that key, associated with the value returned by the `$action` function.

For example, `map:replace($map, 'counter', _{($1 otherwise 0) + 1})` sets the value of the `counter` entry in the map to the previous value plus 1, or to 1 if there is no existing value (and returns the new map).

7.9. `fn:highest()` and `fn:lowest()`

Currently given as example user-written functions in the 3.1 specification, these could usefully become part of the core library. For example, `highest(// p, string-length#1)` returns the longest paragraph in the document.

7.10. `fn:replace-with()`

The new function `fn:replace-with($in, $regex, $callback, [$flags])` is similar to `fn:replace()`, but it computes the replacement string using a callback function. For example, `replace-with($in, '[0-9]+', .{string(number()+1)})` adds one to any number appearing within the supplied string: "Chapter 12" becomes "Chapter 13".

7.11. `fn:characters()`

Splits a string into a sequence of single-character strings. Avoids the clumsiness of `string-to-codepoints(x)!codepoints-to-string()`.

7.12. `fn:is-NaN()`

Returns true if and only if the argument is the `xs:float` or `xs:double` value `NaN`.

7.13. Node construction functions

Once you start using higher-order functions extensively, you discover the problem that in order for a user-written function to create nodes, your code has to be written in XSLT rather than in XPath. This is restrictive, because it means for example that the logic cannot be included in static expressions, nor in expressions evaluated using `xsl:evaluate` (I've seen people using `fn:parse-xml()` to get around this restriction, for example `fn:parse-xml("<foo/>")` to create an element node named `foo`). A set of simple functions for constructing new nodes would be very convenient. Specifically:

- `fn:new-element(QName, content)` – constructs a new element node with a given name; `$content` is a sequence of nodes used to form the content of the element, following the rules for constructing complex content.
- `fn:new-attribute(QName, string)` – constructs a new attribute node, similarly
- `fn:new-text(string)` - constructs a new text node
- `fn:new-comment(string)` - constructs a new comment node
- `fn:new-processing-instruction(string, string)` - constructs a new processing instruction node

- `fn:new-document(content)` - constructs a new document node node
- `fn:new-namespace(content)` - constructs a new namespace node

Despite their names, these functions are defined to be *non-deterministic with respect to node identity*: if called twice with the same arguments, it is system-dependent whether or not you get the same node each time, or two different nodes. In practice, very few applications are likely to care about the difference, and leaving the system to decide leaves the door open for optimizations such as loop-lifting.

Here's an example to merge the attributes on two sequences of elements, taken pairwise:

```
<out>
  <xsl:sequence select="for-each-pair($seq1, $seq2,
                                      _{new-element(node-name($1),
($1/@*, $2/@*))})"/>
</out>
```

The functional approach to node construction is useful when elements are created conditionally. Consider this example from the XSLT 3.0 specification:

```
<xsl:for-each-group select="node()"
            group-adjacent="self::ul or self::ol">
    <xsl:choose>
        <xsl:when test="current-grouping-key()">
            <xsl:copy-of select="current-group()"/>
        </xsl:when>
        <xsl:otherwise>
            <p>
                <xsl:copy-of select="current-group()"/>
            </p>
        </xsl:otherwise>
    </xsl:choose>
</xsl:for-each-group>
```

This can now be written:

```
<xsl:for-each-group select="node()"
            group-adjacent="self::ul or self::ol">
  <xsl:if test="current-grouping-key()"
          then="current-group()"
          else="new-element(QName("", "p"), current-group())"/>
</xsl:for-each-group>
```

References

[1] Michael Kay. *Transforming JSON using XSLT 3.0*. Presented at XML Prague, 2016. Available at `http://archive.xmlprague.cz/2016/files/`

xmlprague-2016-proceedings.pdf and at http://www.saxonica.com/papers/xmlprague-2016mhk.pdf

[2] Michael Kay. *An XSD 1.1 Schema Validator Written in XSLT 3.0*. Presented at Markup UK, 2018. Available at http://markupuk.org/2018/Markup-UK-2018-proceedings.pdf and at http://www.saxonica.com/papers/markupuk-2018mhk.pdf

[3] Michael Kay, John Lumley. *An XSLT compiler written in XSLT: can it perform?*. Presented at XML Prague, 2019. Available at http://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf and at http://www.saxonica.com/papers/xmlprague-2019mhk.pdf

[4] John Lumley, Debbie Lockett and Michael Kay. *Compiling XSLT3, in the browser, in itself*. Presented at Balisage: The Markup Conference 2017, Washington, DC, August 1-4, 2017. In Proceedings of Balisage: The Markup Conference 2017. Balisage Series on Markup Technologies, vol. 19 (2017). Available at https://doi.org/10.4242/BalisageVol19.Lumley01

[5] *XSL Transformations (XSLT) Version 3.0*. W3C Recommendation, 8 June 2017. Ed. Michael Kay, Saxonica. http://www.w3.org/TR/xslt-30

(Re)presentation in XForms

Steven Pemberton
CWI, Amsterdam
<steven.pemberton@cwi.nl>

Alain Couthures
AgenceXML, France

Abstract

XForms [6][7] is an XML-based declarative programming language. XForms programs have two parts: the form or model, contains descriptions of the data used, and constraints and relationships between the values that are automatically checked and kept up to date by the system; and the content, which displays data to the user, and allows interaction with values.

Content is presented to the user with abstract controls, which bind to values in the model, reflecting properties of the values, and in general allowing interaction with the values in various ways. Controls are unusual in being declarative, describing what they do, but not how they should be represented, nor precisely how they should achieve what is required of them. The abstract controls are concretised by the implementation when the XForm application is presented to the user, taking into account modality, features of the client device, and instructions from style sheets.

This has a number of advantages: flexibility, since the same control can have different representations depending on need and modality, device independence, and accessibility.

This paper discusses how XForms content presentation works, and the requirements for controls, discusses how one implementation, XSLTForms, implements content presentation, and the use of CSS styling to meet the requirements of controls, and future improvements in both.

Keywords: XML, XForms, presentation, CSS, styling, skinning

1. XForms

XForms is a declarative markup for defining applications. It is a W3C standard, and in worldwide use, for instance by the Dutch Weather Service, KNMI, many Dutch and UK government websites, the BBC, the US Department of Motor Vehicles, the British National Health Service, and many others. Largely thanks to its declarative nature, experience has shown that you can produce applications in

much less time than with traditional procedural methods, typically a tenth of the time [5].

2. Principles

XForms programs are divided into two parts: the *form* or *model*, which contains the data, and describes the properties of the data, the types, constraints, and relationships with other values, and the *content*, which displays values from the model, and allows interaction with those values. This can be compared with how HTML separates styling from content, or indeed how a recipe first lists its ingredients, before telling you what to do with them.

The model consists of any number of *instances*, collections of data that can either be loaded from external data:

```
<instance src="data.xml"/>
```

or can contain inline data:

```
<instance>
   <payment xmlns="">
      <amount/>
      <paymenttype/>
      <creditcard/>
      <address>
         <name/>
         <street1/>
         <street2/>
         <city/>
         <state/>
         <postcode/>
         <country/>
      </address>
   </payment>
</instance>
```

Properties can then be assigned to data values using *bind* elements. Properties can be:

types (which can also be assigned with schemas):

```
<bind ref="amount" type="decimal"/>
```

relevance conditions:

```
<bind ref="creditcard" relevant="../paymenttype = 'cc'"/>
```

required/*optional* conditions:

```
<bind ref="postcode" required="true()"/>
<bind ref="state" required="../country = 'USA'/>
```

read-only conditions:

```
<bind ref="ordernumber" readonly="true()"/>
```

constraints on a value:

```
<bind ref="age" constraint=". &gt; 17 and . &lt; 65"/>
<bind ref=creditcard" constraint="is-card-number(.)"/>
```

or *calculations*:

```
<bind ref="total" calculate="sum(instance('order')/item/price)"/>
```

XForms controls are used in the content to display and allow interaction with values, such as output:

```
<output ref="amount" label="Amount to pay"/>
```

input:

```
<input ref="creditcard" label="Credit card number"/>
```

or selecting a value:

```
<select1 ref="paymenttype" label="How will you pay?">
   <item label="Cash on delivery">cod</item>
   <item label="Credit card">cc</item>
   <item label="By bank">bank</item>
</select1>
```

XForms controls bind to values in instances, and are unusual in that in contrast with comparable systems, they are not visually oriented, but specify their intent: *what* they do and not *how*. Visual requirements are left to styling.

This has an important effect: the controls are as a result device- and modality-independent, and accessible, since an implementation has a lot of freedom in how they can be represented. The controls are an abstract representation of what they have to achieve, so that the same control can have different representations according to need.

3. The effect of data properties on presentation of controls

Since implementations have a degree of freedom in how they represent controls, they can take the properties of the values into account in deciding how to do it.

The major effect is based on *relevance*, and demanded by the language: if a value is not relevant, then the control it is bound to is not displayed. So for instance, if the buyer is not paying by credit card, then the control for input of the credit-card number

```
<input ref="creditcard" label="Credit card number"/>
```

will not be displayed. Note that most XForms data properties depend on a boolean expression, and so the property can change accordingly at run time.

The display of values that are not even present in the data, which can be seen as a sort of super-nonrelevance, is similar: controls that are bound to values that are not present are also not displayed. This is in particular useful for data coming from external sources, where certain fields may be optional in the schema. Note that a value may later become available, for instance as a result of insertions, so that the control has nevertheless to be ready to accept a value.

Another property of importance is *type*, where the implementation may adapt the input control to the type of data that it represents. The classic example of this is a control bound to a value of type *date*, which allows the implementation to pop up a date picker rather than requiring the user to type in a complete date. Another classic example is a control bound to a value of type *boolean*, allowing the control to be represented as a check box.

The remaining properties, while not affecting the form of the control, affect other styling aspects.

If a control is bound to a value that is *required*, then it gives the implementation the opportunity to indicate that fact to the user is a consistent manner, for instance by putting a small red asterisk next to the label, or colouring the background red, or both.

If a control is bound to a value that is *readonly*, then the control will look similar, but should be represented in a way that makes it clear to the user that the value is not changeable.

The final property of interest here is general *validity*, both type validity as well as adherence to a *constraint* property. If the value is non-valid, the implementation can display the control in such a way as to make that clear. Additionally, all controls can have an *alert* message associated with them, that the implementation displays when the value is invalid:

```
<input ref="creditcard" label="Credit card number"
       alert="Not a valid credit card number"/>
```

4. Implementation approaches

XForms was deliberately designed to allow different implementation strategies. For instance:

- Native: The XForm is directly served to a client that processes it directly;
- Server-side: The server, possibly after inspecting what the client can accept, transforms or compiles the XForm into something that the client can deal with natively; the client may have to communicate with the server during processing in order to achieve some of the functionality;
- Hybrid: some combination of the above.

As an example, one widely used implementation, XSLTForms [9], works by using an XSLT stylesheet [8] to transform the XForm in the browser, client-side, into a

combination of HTML and Javascript, so that all processing takes place on the client. This has an additional advantage, over a pure server-side implementation, that 'Show Source' shows the XForms source, and not the transformation.

Such an approach requires the design of equivalent constructs in HTML+Javascript to implement the XForm constructs. Since XForms controls contain a lot of implicit functionality, even apparently simple cases can require quite complex transformations.

As an example, the transformation of

```
<input ref="creditcard" label="Credit card number"
       alert="Not a valid credit card number"/>
```

gives the following HTML:

```
<span class="xforms-control xforms-input xforms-appearance xforms-
optional
             xforms-enabled xforms-readwrite xforms-valid"
      xml:id="xsltforms-mainform-input-2_10_2_4_3_">
   <span class="focus"> </span>
   <label class="xforms-label" xml:id="xsltforms-mainform-
label-2_2_10_2_4_3_"
          for="xsltforms-mainform-input-input-2_10_2_4_3_"
         >Credit card number</label>
   <span class="value">
      <input class="xforms-value"
             xml:id="xsltforms-mainform-input-input-2_10_2_4_3_"
type="text"
             style="text-align: left;"/>
   </span>
   <span class="xforms-required-icon">*</span>
   <span class="xforms-alert">
      <span class="xforms-alert-icon"> </span>
      <span xml:id="xsltforms-mainform-alert-4_2_10_2_4_3_"
            class="xforms-alert-value"
           >Not a valid credit card number</span>
   </span>
</span>
```

plus a number of *event listeners* to implement the semantics.

This exposes two essential aspects of the transformation: enclosing `<span>` elements for the control as a whole, and each of its subparts – label, input field, support for the required property and alert value; and the use of `class` values to record properties of the control and its bound value. In this case you can see that it is recorded as being a *control*, in particular an *input* control, that the value is *optional* not *required*, the control is *enabled*, the value is *readwrite*, and (currently) *valid*. Since these last four values are dynamic, depending on a boolean expres-

sion and the type, they can change during run-time, for instance `xforms-valid` can become `xforms-invalid`.

Here is another example for a similar control, but bound to a value of type boolean:

```
<input ref="truth" label="boolean"/>
```

which gives:

```
<span class="xforms-control xforms-input xforms-appearance
             xforms-optional xforms-enabled xforms-readwrite
             xforms-valid" xml:id="xsltforms-mainform-input-2_6_2_4_3_">
   <span class="focus"> </span>
   <label class="xforms-label"
          xml:id="xsltforms-mainform-label-1_2_6_2_4_3_"
          for="xsltforms-mainform-input-input-2_6_2_4_3_">boolean</label>
   <span class="value">
      <input type="checkbox"
             xml:id="xsltforms-mainform-input-input-2_6_2_4_3_"/>
   </span>
   <span class="xforms-required-icon">*</span>
   <span class="xforms-alert">
      <span class="xforms-alert-icon"> </span>
   </span>
</span>
```

Note that since *type* is not a dynamic property, the system does not have to be prepared for types changing.

5. Integration in HTML+CSS

One advantage of using HTML as target code is that you have the power of Cascading Style Sheets (CSS) [3] at your disposal to support presentation. In particular the CSS can use the `class` values as shown in the examples above to affect the presentation.

The most obvious case is for when a value becomes non-relevant, and therefore the control becomes disabled. CSS can be used to remove the control from the presentation:

```
.xforms-disabled {display: none}
```

In fact, because of CSS cascading rules, it is essential in this case to override the cascade:

```
.xforms-disabled {display: none !important}
```

Another case is dealing with whether the value is *required* or not. There is an element in the markup that holds an icon to be displayed if the value is required:

```
<span class="xforms-required-icon">*</span>
```

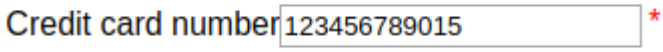
The default is not to display it:

```
.xforms-required-icon {
    display: none;
}
```

unless the value is *required*:

```
.xforms-required .xforms-required-icon {
    display: inline;
    margin-left: 3px;
    color: red;
}
```

giving



A further case is if a value is *invalid*. All information about presentation for invalidity is contained in the `span` element of class `xforms-alert`:

```
<span class="xforms-alert">
      <span class="xforms-alert-icon"> </span>
      <span xml:id="xsltforms-mainform-alert-4_2_10_2_4_3_"
            class="xforms-alert-value"
           >Not a valid credit card number</span>
   </span>
</span>
```

Similarly to *required*, the default is not to display it:

```
.xforms-alert {
    display: none;
}
```

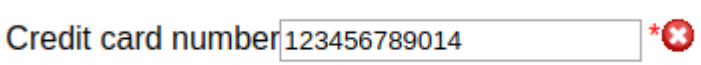
and then if the value becomes invalid, to display it

```
.xforms-invalid .xforms-alert {
    display: inline;
}
```

along with the alert icon:

```
.xforms-alert-icon {
   background-image: url(../img/icon_error.gif);
   background-repeat : no-repeat;
}
```

giving:

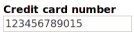


Using CSS properties, hovering over the alert icon pops up the alert text:

6. Improvements

For a planned new version of XSLTForms, we are working on a number of improvements in the visual approach, as well as in the use of the CSS, and the format of the transformed HTML, the aim being to make the default styling more attractive, and more flexible. (What is presented here is work in progress.)

For a start, labels will be styled bold, and by default above the control:



This helps in lining up controls vertically and generally makes the style more restful to the eye.

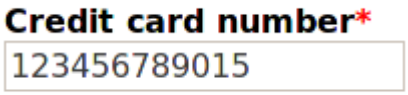
This is simply done by making the label element a block, with bold font:

```
.xforms-label {font-weight: bold; display: block}
```

In the case of a value being *required*, although the transformed HTML contains a representation of the asterisk to be included, in the element with class `xforms-required-icon`, since CSS offers the ability to insert text, it gives more flexibility to ignore the required icon, and instead insert it from the CSS:

```
.xforms-required-icon {display: none}
 .xforms-required .xforms-label:after {content: '*'; color: red}
```

giving:



This also means that in the future transformed HTML, the `span` element with class of `required-icon` no longer needs to be included.

If a value is *invalid*, either due to its type or a constraint, using the same technique a large red X can be displayed after the label:

```
.xforms-invalid .xforms-label:after
    {content: ' ✖'; color: red}
```

However, because of CSS cascading rules, only one of these rules can match at any one time, so that if a value is both *required* and *invalid* a rule has to be added to match that case as well:

```
.xforms-required.xforms-invalid .xforms-label:after
    {content: '*✖'; color: red}
```

For invalid input values, the background of the input field will additionally be coloured a light red:

```
.xforms-invalid .value input
    {background-color: #fcc; border-style: solid; border-width: thin}
```

Finally for invalid values the alert text has to be displayed. Normally alerts will not be displayed:

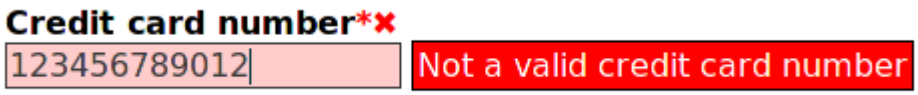
```
.xforms-alert-icon {display: none}
 .xforms-alert {display: none; position: relative;}
```

(Again the `alert-icon` element is no longer needed in the transformed HTML.)

On becoming *invalid*, the alert text can be popped up:

```
.xforms-invalid .xforms-alert {display: inline}
 .xforms-alert-value {
    color: white;
    background-color: red;
    margin-left: 0.5ex;
    border: thin solid black;
    padding: 0.2ex
}
```

the end result being:

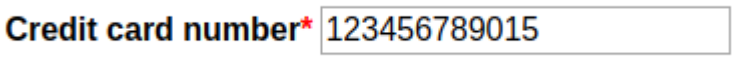


7. Skinning

Unfortunately, CSS in general doesn't allow the reordering of content, but nevertheless there is some freedom to how labels of controls can be positioned. Since the label element is textually before the input field in the transformed HTML, it is easy to position the label above or to the left of the control. For instance, instead of above the control as in the last example, to the left:

```
.xforms-label
   {display: inline-block; width: 12ex; text-align: right}
```

giving



With care, labels can be positioned to the right of the control, by floating the label element, or with even more care, below, using relative positioning.

To give the user some freedom in how XForms are displayed, but without having to know details of CSS, a skinning technique will be used [1] [2]. This is where a top-level element is given classes that indicate presentation requirements of the enclosed content. For instance, the enclosing `body` element can indicate the positioning required for labels:

```
<body class="xforms-labels-left">
```

CSS rules then key off this value to provide different presentations for different cases:

```
.xforms-label {font-weight: bold}

 .xforms-labels-top .xforms-label
    {display: block; margin: 0}

 .xforms-labels-left .xforms-label
    {display: inline-block; width: 20ex; text-align: right}
```

Thanks to containment hierarchy, this offers quite a lot of flexibility, since even in one XForm different sets of controls can be formatted differently:

```
<group class="xforms-labels-left">
   ...
</group>
<group class="xforms-labels-top">
   ...
</group>
```

8. Future Transformation

HTML5 [4] allows you to define custom elements for a document.

Although these wouldn't offer any additional functionality, transforming to an HTML using themwould mean that the transformed HTML can be kept far closer to the original XForm. As an example, a control such as

```
<input ref="creditcard" label="Credit card number"
       alert="Not a valid credit card number"/>
```

could be transformed to

```
<xforms-input xf-ref="@creditcard">
    <xforms-label>Credit card number</xforms-label>
    <xforms-alert>Not a valid credit card number</xforms-alert>
</xforms-input>
```

9. Conclusion

XForms offers a lot of flexibility in how it can be implemented. One of the advantages of implementing it by transforming to HTML means that the power of CSS is available for presentation ends. However, to avoid requiring the XForms programmer to necessarily know CSS, skinning techniques can be used to offer flexibility to the presentations available. A new XForms implementation is in preparation that will use those techniques.

10. References

Bibliography

[1] *Bootstrap*. https://getbootstrap.com/css/ .

[2] *Bulma*. http://bulma.io/documentation/overview/classes/ .

[3] W3C. *CSS*. 2020. https://www.w3.org/Style/CSS/ .

[4] W3C. *HTML5*. http://www.w3.org/TR/html5/. .

[5] Steven Pemberton. *An Introduction to XForms*. XML.com. 2018. https://www.xml.com/articles/2018/11/27/introduction-xforms/ .

[6] John Boyer (ed). *XForms 1.1*. 2009. W3C. https://www.w3.org/TR/xforms11 .

[7] Erik Bruchez et al. (eds). *XForms 2.0*. W3C. 2020. https://www.w3.org/community/xformsusers/wiki/XForms_2.0 .

[8] W3C. *XSLT*. https://www.w3.org/TR/xslt/all/ .

[9] Alain Couthures. *XSLTForms*. AgenceXML. 2014. http://www.agencexml.com/xsltforms .

Greenfox – a schema language for validating file systems

Hans-Juergen Rennau
parsQube GmbH
<hans-juergen.rennau@parsqube.de>

Abstract

Greenfox is a schema language for validating file systems. One key feature is an abstract validation model inspired by the SHACL language. Another key feature is a view of the file system which is based on the XDM data model and thus supports a set of powerful expression languages (XPath, foxpath, XQuery). Using their expressions as basic building blocks, the schema language unifies navigation within and between resources and access to the structured contents of files with different mediatypes.

Keywords: Validation, SHACL, XSD, JSON Schema, Schematron

1. Introduction

How to validate data against expectations? Major options are visual inspection, programmatic checking and validation against a schema document (e.g. XSD, RelaxNG, Schematron, JSON Schema) or a schema graph (e.g. SHACL). Schema validation is in many scenarios the superior approach, as it is automated and declarative. But there are also limitations worth considering when thinking about validation in general.

First, schema languages describe instances of a particular format or mediatype only (e.g. XML, JSON, RDF), whereas typical projects involve a mixture of mediatypes. Therefore schema validation tends to describe the state of resources which are pieces from a jigsaw puzzle, and the question arises how to integrate the results into a coherent whole.

Second, several schema languages of key importance are grammar based and therefore do not support "incremental validation" – starting with a minimum of constraints, and adding more along the way. We cannot use XSD, RelaxNG or JSON Schema in order to express some very specific key expectation, without saying many things about the document as a whole, which may be a task requiring disproportional effort. Rule based schema languages (like Schematron) do support incremental validation, but they are inappropriate for comprehensive validation as accomplished by grammar based languages.

As a consequence, schema validation enables isolated acts of resource validation, but it cannot accomplish the integration of validation results. Put differently,

schema validation may contribute to, but cannot accomplish, system validation. The situation might change in an interesting way if we had a schema language for validating *file system contents* – arbitrary trees of files and folders. This simple abstraction suffices to accommodate any software project, and it can accommodate system representations of very large complexity.

This document describes an early version of **greenfox**, a schema language for validating file system contents. By implication, it can also be viewed as a schema language for the validation of *systems*. Such a claim presupposes that a meaningful reflection of system properties, state and behaviour can be represented by a collection of data (log data, measurement results, test results, configurations, …) distributed over a set of files arranged in a tree of folders. It might then sometimes be possible to translate meaningful definitions of system validity into constraints on file system contents. At other times it may not be possible, for example if the assessment of validity requires a tracking of realtime data.

The notion of system validation implies that extensibility must be a key feature of the language. The language must not only offer a scope of expressiveness which is immediately useful. It must at the same time serve as a *framework*, within which current capabilities, future extensions and third-party contributions are uniform parts of a coherent whole. The approach we took is a generalization of the key concepts underlying SHACL [7], a validation language for RDF data. These concepts serve as the building blocks of a simple metamodel of validation, which offers guidance for extension work.

Validation relies on the key operations of navigation and comparison. File system validation must accomplish them in the face of divers mediatypes and the necessity to combine navigation within as well as between resources. In response to this challenge, greenfox is based on a *unified data model* (XDM) [7] and a *unified navigation model* (foxpath/XPath) [3] [4] [5], [9] [11] built upon it.

Validation produces results, and the more complex the system, the more important it may become to produce results in a form which combines maximum precision with optimal conditions for integration with other resources. This goal is best served by a *vocabulary* for expressing validation results and schema contents in a way which does not require any context (like a document type) for being understood. We choose an RDF based definition of validation schema and validation results, combined with a bidirectional mapping between RDF and more intuitive representations, XML and JSON. For practical purposes, we assume the XML representation to be the form most frequently used. Concerning schemas, this document discusses only the XML representation. Concerning results, XML and RDF are dealt with.

Before providing an overview of the greenfox language, a detailed example should give a first impression of how the language can be used.

2. Getting started with greenfox

This section illustrates the development of a greenfox schema designed for validating a file system tree against a set of expections. Such a validation can also be viewed as validation of the system "behind" the file system tree, represented by its contents.

2.1. The system – system S

Consider **system S** – an imaginary system which is a collection of web services. We are going to validate a *file system representation* which is essentially a set of test results, accompanied by resources supporting validation (XSDs, codelists and data about expected response messages). The following listing shows a file system tree which is a representation of system S, as observed at a certain point in time:

```
system-s
. resources
. . codelists
. . . codelist-foo-article.xml
. . xsd
. . . schema-foo-article.xsd
. testcases
. . test-t1
. . . config
. . . . msg-config.xml
. . . input
. . . . getFooRQ*.xml
. . . output
. . . . getFooRS*.xml
. . +test-t2   (contents: see test-t1)
. . usecases
. . . usecase-u1
. . . . usecase-u1a
. . . . . +test-t3   (contents: see test-t1)
```

The concrete file system tree must be distinguished from the *expected file system* tree, which is described by the following rules.

Table 1. Rules defining "validity" of the considered file system.

File or folder	File path	Expectation
folder	resources/codelists	Contains one or more codelist files
file	resources/codelists/*	A **codelist file**; name not constrained; must be an XML document containing `<codelist>` elements with a @name attribute and `<entry>` children
folder	resources/xsd	Contains one or more XSDs describing services messages
file	resources/xsd/*	An **XSD schema file**; name not constrained
folder	.//test-*	A test case folder, containing `input`, `output` and `config` folders; apart from these only optional `log-*` files are allowed
folder	.//test-*/config	Test case config folder, containing file `msg-config.csv`
file	.//test-*/config/msg-config.csv	A **message configuration file**; CSV file with three columns: request file name, response file name, expected return code
folder	.//test-*/input	Test case input folder, containing request messages
file	.//test-*/input/*	A **request message file**; name extension `.xml` or `.json`; mediatype corresponding to name extension
folder	.//test-*/output	Test case output folder, containing response messages
file	.//test-*/output/*	A **response message file**; name extension `.xml` or `.json`; mediatype corresponding to name extension

The number and location of testcase folders (`test-*`) are unconstrained. This means that the testcase folders may be grouped and wrapped in any way, although they must not be nested. So the use of a `testcases` folder wrapping all testcase folders - and the use of `usecase*` folders adding additional substructure - is allowed, but must not be expected. The placing of XSDs in folder `resources/xsd`, on the other hand, is obligatory, and likewise the placing of codelist documents in folder `resources/codelists`. The names of XSD and codelist files are not constrained.

Apart from these static constraints, the presence of some files implies the presence of other files:

- For every request message, there must be a response message with a name derived from the request file name (replacing substring `RQ` with `RS`).

Expectations are not limited to the presence of files and folders - they include details of file contents, in some cases relating the contents of different files with different mediatypes:

- For every response message in XML format, there is exactly one XSD against which it can be validated
- Every response message in XML format is valid against the appropriate XSD
- Response message items (XML elements and JSON fields) with a particular name (e.g. `fooValue`) must be found in the appropriate XML codelist discovered in a set of codelist files
- Response message return codes (contained by XML and JSON documents) must be as configured by the corresponding row in a CSV table

2.2. Building a greenfox schema "system S"

Now we create a greenfox schema which enables us to validate the file system against these expectations. An initial version only checks the existence of non-empty XSD and codelists folders:

```
<greenfox greenfoxURI="http://www.greenfox.org/ns/schema-examples/system-s"
          xmlns="http://www.greenfox.org/ns/schema">

  <!-- *** System file tree *** -->
  <domain path="\tt\greenfox\resources\example-system\system-s"
          name="system-s">

      <!-- *** System root folder shape *** -->
      <folder foxpath="." id="systemRootFolderShape">

        <!-- *** XSD folder shape *** -->
        <folder foxpath=".\\resources\xsd" id="xsdFolderShape">
            <targetSize count="1"
                        countMsg="No XSD folder found"/>

            <file foxpath="*.xsd" id="xsdFileShape">
                <targetSize minCount="1"
                            minCountMsg="No XSDs found"/>
            </file>
        </folder>
```

```
        <!-- *** Codelist folder shape *** -->
        <folder foxpath=".\\resources\codelists" id="codelistFolderShape">
            <targetSize count="1"
                        countMsg="No codelist folder found"/>

            <file foxpath="*[is-xml(.)]" id="codelistFileShape">
                <targetSize minCount="1"
                            minCountMsg="No codelist files found"/>
            </file>
        </folder
      </folder>
   </domain>
</greenfox>
```

The <domain> element represents the root folder of a **file system tree** to be validated. The folder is identified by a mandatory @path attribute.

A <folder> element describes a set of folders selected by a *target declaration*. Here, the target declaration is a foxpath expression, given by a @foxpath attribute. Foxpath [3] [4] is an extended version of XPath 3.0 which supports file system navigation, node tree navigation and a mixing of file system and node tree navigation within a single path expression. Note that file system navigaton steps are connected by a backslash operator, rather than a slash, which is used for node tree navigation steps. The foxpath expression is evaluated in the context of a folder selected by the target declaration of the *containing* <folder> element (or the @path of <domain>, if there is no containing <folder>). Evaluation "in the context of a folder" means that the *initial context item* is the file path of that folder, so that relative file system path expressions are resolved in this context (see [3] for details). For example, the expression

```
.\\resources\xsd
```

resolves to the xsd folders contained by a resources folder found at any depth under the context folder, which here is

```
\tt\greenfox\resources\example-system\system-s\.
```

Similarly, a <file> element describes the set of files selected by its *target declaration*, which is a foxpath expression evaluated in the context of a folder selected by the containing <folder> element's target declaration. So here we have a file element describing all files found at the relative path

```
*.xsd
```

evaluated in the context of any folder selected by

```
\tt\greenfox\resources\example-system\system-s\\resources\xsd
```

A <folder> element represents a **folder shape**, which is a set of **constraints** applying to a **target**. The target is a (possibly empty) set of folders, selected by a

target declaration, e.g. a foxpath expression. The constraints of a folder shape are declared by child elements of the shape element. Every folder in the target is tested against every constraint in the shape. When a folder is tested against a constraint, it is said to be the **focus resource** of the constraint.

Likewise, a `<file>` element represents a **file shape**, defining a set of constraints applying to a target, which is a set of files selected by a target declaration. Folder shapes and file shapes are collectively called **resource shapes**.

The expected number of folders or files belonging to the target of a shape can be expressed by declaring a **constraint**. A constraint has a kind (called the **constraint component**) and a set of arguments passed to the **constraint parameters**. Every kind of constraint has a "signature", a characteristic set of mandatory and optional constraint parameters, defined in terms of name, type and cardinality. A *constraint component* can therefore be thought of as a library function, and a *constraint declaration* is like a function call, represented by elements and/or attributes. Here, we declare a `TargetMinCount` constraint, represented by a @minCount attribute on a `<targetSize>` element. When a resource is validated against a constraint, the imaginary function consumes the constraint parameter values, inspects the resource and returns a validation result. If the constraint is violated, the validation result is a `<gx:red>` element which contains an optional message (either supplied by an attribute or constructed by the processor), along with a set of information items identifying the violating resource (@filePath), the constraint (@constraintComp and @constraintID) and its parameter values (@minCount). In the case of a `TargetMinCount` constraint, the violating resource is the folder providing the context when evaluating the target declaration. Example result:

```
<gx:red msg="No XSDs found"
    filePath="C:/tt/greenfox/resources/example-system/system-s/resources/
xsd"
    constraintComp="TargetMinCount"
    constraintID="TargetSize_2-minCount"
    resourceShapeID="xsdFileShape"
    minCount="1"
    valueCount="0"
    targetFoxpath="*.xsd"/>
```

In a second step we extend our schema with a folder shape whose target consists of *all testcase folders in the system*:

```
<!-- *** Testcase folder shape *** -->
<folder foxpath=".\\test-*[input][output][config]"
id="testcaseFolderShape">
  <targetSize minCount="1"
              minCountMsg="No testcase folders found">
```

```
  <!-- # Check - test folder content ok? -->
  <folderContent
       closed="true"
       closedMsg="Testcase member(s) other than input/output/config, log-
*.">
    <memberFolders names="input, output, config"/>
    <memberFile name="log-*" count="*"/>
  </folderContent>
    …
</folder>
```

The target includes all folders found at any depth under the current context folder (`system-s`), matching the name pattern `test-*` and having (at least) three members `input`, `output` and `config`. The `TargetMinCount` constraint checks that the system contains at least one such folder. The contents of these testcase folders are subject to several constraints defined by the `<folderContent>` element. There must be three subfolders `input`, `output` and `config`, and there may be any number of `log-*` elements, but not any other members (`FolderContentClosed` constraint).

We proceed with a file shape which targets the `msg-config.csv` file in the `config` folder of the test case:

```
<!-- *** msg config file shape *** -->
<file foxpath="config\msg-config.csv" id="msgConfigFileShape" ...>
  <targetSize count="1"
              countMsg="Config file missing"/>
  ...
</file>
```

The `TargetCount` constraint makes this file mandatory, but we want to be more specific: to constrain the *file contents*. The file must be a CSV file, and the third column (which according to the header row is called `returnCode`) must contain a value which is "OK" or "NOFIND" or matches the pattern "ERROR_*". We add attributes to the `<file>` element which specify how to **parse the CSV file into an XML representation** (@mediatype, @csv.separator, @csv.header). As with other non-XML mediatypes (e.g. JSON or HTML), an XML view enables us to leverage XPath and *express* a selection of content items, preparing the data material for fine-grained validation.

We add to the file shape an `<xpath>` element which describes a *selection* of content items and defines a *constrait* which these items must satisfy (expressed by the `<in>` child element):

```
<!-- *** msg config file shape *** -->
<file foxpath="config\msg-config.csv" id="msgConfigFileShape"
      mediatype="csv" csv.separator="," csv.withHeader="yes">
  ...
```

```
    <!-- # Check - configured return codes ok? -->
    <xpath expr="//returnCode"
           inMsg="Config file contains unknown return code">
      <in>
        <eq>OK</eq>
        <eq>NOFIND</eq>
        <like>ERROR_*</like>
      </in>
    </xpath>
  </file>
```

The item selection is defined by an XPath expression (provided by @expr), and an `XPathValueIn` constraint is specified by the `<in>` child element: an item must either be equal to one of the strings "OK" or "NOFIND", or it must match the glob pattern "ERROR_*".

It is important to understand that the XPath expression is evaluated in the context of the **document node** of the document obtained by parsing the file. Here comes an example of a conformant message definition file:

```
request,response,returnCode
getFooRQ1.xml,getFooRS1.xml,OK
getFooRQ2.xml,getFooRS2.xml,NOFIND
getFooRQ3.xml,getFooRS3.xml,ERROR_SYSTEM
```

while this example violates the `XPathValueIn` constraint:

```
request,response,returnCode
getFooRQ1.xml,getFooRS1.xml,OK
getFooRQ2.xml,getFooRS2.xml,NOFIND
getFooRQ3.xml,getFooRS3.xml,ERROR-SYSTEM
```

The second example would produce the following validation result, identify resource and constraint, describing the constraint and exposing the offending value:

```
<gx:red msg="Config file contains unknown return code"
    filePath="C:/tt/greenfox/resources/example-system/system-s/resources/
xsd"
    constraintComp="ExprValueIn"
    constraintID="xpath_1-in"
    valueShapeID="xpath_1"
    exprLang="xpath"
    expr="//returnCode">
    <gx:value>ERROR-SYSTEM</gx:value>
</red>
```

According to the conceptual framework of greenfox, the `<xpath>` element does not, as one might expect, represent a constraint, but a **value shape**. A value shape is a container combining a single **value mapper** with a set of constraints: the

value mapper maps the focus resource to a value - called a **resource value** - which is validated against each one of the constraints. Greenfox supports two kinds of value mapper – XPath expression and foxpath expression, and accordingly there are two variants of a value shape – **XPath value shape** (represented by an `<xpath>` element) and **Foxpath value shape** (`<foxpath>`). See Section 5 for more information about value shapes.

Now we are going to check *request message files*: for each such file, there must be a response file in the `output` folder, with a name derived from the request file name (replacing the last occurrence of substring "RQ" with "RS"). This is a constraint which does not depend on file contents, but on file system contents found "around" the focus resource. A check requires **navigation of the file system**, rather than file contents. We solve the problem with a Foxpath value shape:

```
<!-- *** Request file shape *** -->
<file foxpath="input\(*.xml, *.json)" id="requestFileShape">
  ...
  <!-- # Check - request with response ? -->
  <foxpath
      expr="..\..\output\*\file-name(.)"
      containsXPath=
           "$fileName ! replace(., '(.*)RQ(.*)$', '$1RS$2')"
      containsXPathMsg="Request without response"
  ...
<file>
```

A Foxpath value shape combines a foxpath expression (@expr) with a set of constraints. The expression maps the focus resource to a resource value, which is validated against all constraints. Here we have an expression which maps the focus resource to a list of file names found in the `output` folder. A single constraint, represented by the @containsXPath attribute, requires the foxpath expression value to contain the value of an XPath expression, which maps the request file name to the response file name. The constraint is satisfied if and only if the response file is present in the `output` folder.

As with XPath value shapes, it is important to be aware of the evaluation context. We have already seen that in an XPath value shape the initial context item is the *document node* obtained by parsing the text of the focus resource into an XML representation. In a Foxpath value shape the initial context item is the *file path* of the focus resource, which here is the file path of a request file. The foxpath expression starts with two steps along the parent axis (`..\..`) which lead to the enclosing testcase folder, from which navigation to the response files and their mapping to file names is trivial:

```
..\..\output\*\file-name(.)
```

A Foxpath value shape does not require the focus resource to be parsed into a document, as the context is a file path, rather than a document node. Therefore, a Foxpath value shape can also be used in a folder shape. We use this possibility in order to constrain the `codelists` folder to contain non-empty `<codelist>` elements with unique names:

```
<folder foxpath=".\\resources\codelists" id="codelistFolderShape">
    ...
    <!-- # Check - folder contains codelists? -->
    <foxpath expr=".\*.xml//codelist[entry]/@name"
             minCount="1"
             minCoutMsg="Codelist folder without codelists"
             itemsUnique="true"
             itemsUniqueMsg="Codelist names must be unique"/>
    ...
</folder>
```

Note the unified view of file system contents offered by the foxpath language: a single expression starts with file system navigation, visiting all `.xml` files in the current folder, enters their XML content and selects the @name attributes of non-empty `codelist` elements, which may occur at any depth inside the content trees.

Now we turn to the *response message files*. They must be "fresh", that is, have a timestamp of last modification which is after a limit timestamp provided by a call parameter of the system validation. This is accomplised by a `LastModified` constraint, which references the parameter value. Besides, response files must not be empty (`FileSize` constraint):

```
<!-- *** Response file shape *** -->
<file foxpath="output\(*.xml, *.json)" mediatype="xml-or-json">
    ...
    <!-- # Check - response fresh? -->
    <lastModified ge="${lastModified}"
                  geMsg="Stale output file"

    <!-- # Check - response non-empty? -->
    <fileSize gt="0"
              gtMsg="Empty output file"
    ...
</file>
```

The placeholder `${lastModified}` is substituted with the value passed to the greenfox processor as input parameter and declared in the schema as a *context parameter*:

```
<greenfox ... >
  <!-- *** External context *** -->
```

```
  <context>
    <field name="lastModified"
  </context>
  ...
</greenfox>
```

We have several expecations related to the contents of response files. If the response is an XML document (rather than JSON), it must be valid against some XSD found in the XSD folder. XSD validation is triggered by an `XSDValid` constraint, with a foxpath expression locating the XSD(s) to be used:

```
<!-- *** Response file shape *** -->
<file foxpath="output\(*.xml, *.json)" mediatype="xml-or-json">
    ...
    <!-- # Check - schema valid? (only if XML) -->
    <ifMediatype eq="xml">
        <xsdValid msg="Response msg not XSD valid"
                  xsdFoxpath="$domain\resources\xsd\\*.xsd"/>
    </ifMediatype>
    ...
</file>
```

It is not necessary to specify an individual XSD – the greenfox processor inspects all XSDs matching the expression and selects for each file to be validated the appropriate XSD. This is achieved by comparing name and namespace of the root element with local name and target namespace of all element declarations found in the XSDs selected by the foxpath expression. If not exactly one element declaration is found, an error is reported, otherwise XSD validation is performed. Note the variable reference `$domain`, which can be referenced in any XPath or foxpath expression and which provides the file path of the domain folder.

The next condition to be checked is that certain values from the response (selected by XPath `//*:fooValue`) are found in a particular codelist. Here we use an XPath value shape with an `ExprValueInFoxpath` constraint, represented by the @inFoxpath attribute:

```
<!-- *** Response file shape *** -->
<file foxpath="output\(*.xml, *.json)" mediatype="xml-or-json">
    ...
   <!-- # Check - known article number? -->
   <xpath expr="//*:fooValue"
          inFoxpath="$domain\\codelists\*.xml
                     /codelist[@name eq 'foo-article']/entry/@code"
          inFoxpathMsg="Unknown foo article number"/>
</file>
```

As always with an XPath value shape, the XPath expression (@expr) selects the content items to be checked. The `ExprValueInFoxpath` constraint works as fol-

lows: it evaluates the foxpath expression provided by constraint parameter @inFoxpath and checks that every item of the value to be checked also occurs in the value of the foxpath expression. As here the foxpath expression returns all entries of the appropriate codelist, the constraint is satisfied if and only if every `<fooValue>` element in the response contains a string found in the codelist.

Note that this value shape works properly for both, XML and JSON responses. Due to the @mediatype annotation on the file shape, which is set to `xml-or-json`, the greenfox processor first attempts to parse the file as an XML document. If this does not succeed, it attempts to parse the file as a JSON document and transform it into an equivalent XML representation. In either case, the XPath expression is evaluated in the context of the document node of the resulting XDM node tree. In such cases one has to make sure, of course, that the XPath expression can be used in both structures, original XML and XML capturing the JSON content, which is the case in our example.

As a last constraint, we want to check the return code of a response. The expected value can be retrieved from the message config file, a CSV file in the `config` folder: it is the value found in the third column (named `returnCode`) of the row in which the second column (named `response`) contains the file name of the response file. We use a Foxpath value shape with an expression fetching the expected return value from the CSV file. This is accomplished by a mixed navigation, starting with file system navigation leading to the CSV file, then drilling down into the file and fetching the item of interest. The value against which to compare is retrieved by a trivial XPath expression (@eqXPath):

```
<!-- *** Response file shape *** -->
<file foxpath="output\(*.xml, *.json)" mediatype="xml-or-json">
    ...
    <!-- # Check - return code expected? -->
    <foxpath expr="..\..\config\msg-config.csv\csv-doc(., ',', 'yes')
                  //record[response eq $fileName]/returnCode"
             eqXPath="//*:returnCode"
             eqXPathMsg="Return code not the configured
value"
</file>
```

The complete schema is shown in Appendix A. To summarize, we have developed a schema which constrains the presence and contents of folders, the presence and contents of files, and relationships between contents of different files, in some cases belonging to different mediatypes. The development of the schema demanded familiarity with XPath, but no programming skills beyond that.

3. Basic principles

The "Getting started" section has familiarized you with the basic building blocks and principles of greenfox schemas. They can be summarized as follows.

- A file system is thought of as containing two kinds of resources, **folders** and **files**.
- Resources are validated against **resource shapes.**
- There are two kinds of resource shapes – **folder shapes** and **file shapes.**
- A resource shape is a set of **constraints** which apply to each resource validated against the shape.
- A resource which is validated against a shape is called a **focus resource**.
- A resource shape may have a **target declaration** which selects a set of focus resources.
- A target declaration of a resource shape can be a file path or a foxpath expression.
- A target declaration of a resource shape is resolved in the context of all resources obtained from the target declaration of the containing resource shape.
- Every violation of a constraint produces a **validation result** describing the violation and identifying the focus resource and the constraint.
- Constraints can apply to **resource properties** like the last modification time or the file size.
- Constraints can apply to a **resource value**, which is a value to which the resource is mapped by an expression, or by a chain of expressions.
- A **value shape** combines an expression mapping the focus resource to a resource value, or a resource value to another resource value, and a set of constraints against which to validate the resource value obtained.
- The expression used by a value shape may be an **XPath expression** or a **foxpath expression**.
- The **foxpath context item** used by a value shape mapping a focus resource to a resource value is the file path of the focus resource. The foxpath context item used by a value shape mapping a preceding resource value to another resource value is a single item of the preceding resource value.
- The **XPath context item** used by a value shape mapping a focus resource to a resource value is the root of an XDM node tree representing the content of the focus resource, or the file path of the focus resource if an XDM node tree cannot be constructed. The XPath context item used by a value shape mapping a preceding resource value to another resource value is a single item of the preceding resource value.
- **XDM node tree representations** of file resources can be controlled by mediatype related attributes on a file shape.
- When validating resources against resource shapes, the heterogeneity of mediatypes can be hidden by a **unified representation as XDM node trees.**
- When validating resources against resource shapes, the heterogeneity of navigation (within resource contents and between resources) can be hidden by a **unified navigation language.** (foxpath)

4. Information model

This section describes the information model underlying the operations of greenfox.

4.1. Part 1: resource model

A **file system tree** is a tree whose nodes are file system resources – folders and files.

A **file system resource** has an identity, resource properties, derived resource properties and resource values.

The **resource identity** of a file system resource can be expressed by a combination of file system identity and a file path locating the resource within the file system.

A **resource property** has a name and a value which can be represented by an XDM value.

A **derived resource property** is a property of a resource property value, or of a derived resource property value, which can be represented by an XDM value.

A **resource value** is the XDM value of an expression evaluated in the context of a resource property, or of a derived resource property, or of an item from another resource value.

4.1.1. Folder resources

The table below summarizes the **resource properties** of a folder resource, as currently evaluated by greenfox. More properties may be added in the future, e.g. representing access rights.

Table 2. Resource properties of a folder resource.

Property name	Value type	Description
[name]	`xsd:string?`	The folder name; optional – the file system root folder does not have a name
[parent]	Folder resource	The XDM representation of resource identity is its file path
[children]	Folder and file resources	The XDM representation of resource identity is its file path
[last-modified]	`xsd:dateTime`	May be out of sync when comparing values of resources from different machines

A folder has the following **derived resource properties**.

Table 3. Derived resource properties of a folder resource.

Property name	Value type	Description
[filepath]	`xsd:string`	The names of all ancestor folders and the folder itself, separated by a slash

Resource values of a folder are obtained by evaluating a foxpath expression in the context of [filepath]. They can also be obtained by evaluating an XPath or a foxpath expression in the context of an item taken from another resource value. See Appendix D for implications of this recursive definition.

4.1.2. File resources

A file has the following **resource properties**, as currently evaluated by greenfox.

Table 4. Resource properties of a file resource.

Property name	Value type	Description
[name]	`xsd:string`	Mandatory – a file must have a name
[parent]	Folder resource	The XDM representation of resource identity is its file path
[last-modified]	`xsd:dateTime`	May be out of sync when comparing values of resources from different machines
[size]	`xsd:integer`	File size, in bytes
[sha1]	`xsd:string`	SHA-1 hash value of file contents
[sha256]	`xsd:string`	SHA-256 hash value of file contents
[md5]	`xsd:string`	MD5 hash value of file contents
[text]	`xsd:string?`	The text content of the file (empty sequence if not a text file)
[encoding]	`xsd:string?`	The encoding of the text content of the file (empty sequence if not a text file)
[octets]	`xsd:base64-Binary`	The binary file content

A file has the following **derived resource properties**, as currently evaluated by greenfox.

Table 5. Derived resource properties of a file resource.

Property name	Value type	Description
[filepath]	`xsd:string`	The names of all ancestor folders and the folder itself, separated by a slash
[xmldoc]	`document-node()?`	The result of parsing [text] into an XML document
[jsondoc-basex]	`document-node()?`	The result of parsing [text] into a JSON document represented by a document node in accordance with the rules defined by BaseX documentation
[jsondoc-w3c]	`document-node()?`	The result of parsing [text] into a JSON document represented by a document node in accordance with XPath function `fn:json-to-xml`
[htmldoc]	`document-node()?`	The result of parsing [text] into an XML document represented by a document node in accordance with the rules defined by TagSoup documentation
[csvdoc]	`document-node()?`	The result of parsing [text] into an XML document represented by a document node, as controlled by the CSV parsing parameter values derived from a file shape, in accordance with the rules defined by BaseX documentation

Resource values of a file are obtained by evaluating a foxpath expression in the context of [filepath], or evaluating an XPath expression in the context of a [*doc] or [*doc-*] property. They can also be obtained by evaluating an XPath or a foxpath expression in the context of an item taken from another resource value. See Appendix D for implications of this recursive definition.

For information about CSV parsing parameters, see [1], section `# wiki/CSV_Module`.

4.2. Part 2: schema model

File system validation is a mapping of a file system tree and a greenfox schema to a set of validation results.

A **greenfox schema** is a set of shapes.

A **shape** is a resource shape or a value shape.

A **resource shape** is a set of constraints applicable to a file system resource. It has an optional target declaration.

A **target declaration** specifies the selection of a target.

A **target** is a set of focus resources, or a focus value.

A **focus resource** is a resource to be validated against a resource shape.

A **focus value** is a resource value providing a context in which to evaluate value shapes (rather than in the context of a resource's file path or node tree representation). A focus value is typically a set of nodes selected from the resource's node tree representation.

A resource shape is a **folder shape** or a **file shape**.

A **value shape** is a mapping of a focus resource, or of a resource value, to a resource value and a set of constraints which apply to the value.

A **constraint** maps a resource property or a resource value to a validation result.

A constraint is defined by a **constraint declaration**.

A constraint declaration is provided by a shape. It identifies a constraint component and assigns values to the constraint parameters.

A **constraint component** is a set of constraint parameter definitions and a validator.

A **constraint parameter definition** specifies a name, a type, a cardinality range and value semantics.

A **validator** is a set of rules how a resource property or a resource value and the values of the constraint parameters are mapped to a validation result.

A **validation result** describes the outcome of validating a resource against a constraint. It contains a boolean value signaling conformance, an identification of the resource and the constraint, constraint parameter values and optional details about the violation.

4.3. Part 3: validation model

File system validation is a mapping of a file system tree and a greenfox schema to a set of validation results, as defined in the following paragraphs.

Validation of a file system tree against a greenfox schema: Given a file system tree and a greenfox schema, the validation results are the union of results of the validation of the file system tree against all shapes in the greenfox schema.

Validation of a file system tree against a shape: Given a file system tree and a shape in the greenfox schema, the validation results are the union of the results of the validation of all focus resources that are in the target of the shape.

Validation of a focus resource against a shape: Given a focus resource in the file system tree and a shape in the greenfox schema, the validation results are the union of the results of the validation of the focus resource against all constraints declared by the shape.

Validation of a focus resource against a constraint: Given a focus resource in the file system tree and a constraint of kind C in the greenfox schema, the validation results are defined by the validator of the constraint component C. The validator typically takes as input a resource property or a resource value of the focus resource and the arguments supplied to the constraint parameters.

5. Schema building blocks

This section summarizes the **building blocks** of a greenfox schema. Building blocks are the parts of which a schema serialized as XML is composed. The serialized schema should be distinguished from the logical schema, which is independent of a serialization and can be described as a set of logical components (as defined by the information model) and parameter bindings.

Each building block is represented by XML elements with a particular name. There is not necessarily a one-to-one correspondence between building blocks and logical components as defined by the information model. An Import declaration, for example, is a building block without corresponding logical component. Constraints, on the other hand, are logical components which in many cases are not represented by a separate building block, but by attributes attached to a building block. Note also that the information model includes logical components built into the greenfox language and without representation in any given schema (e.g. validators).

Table 6. The building blocks of a greenfox schema.

Building block	Role	XML representation
Import declaration	Declares the import of another greenfox schema so that its contents are included in the current schema	`gx:import`
Context declaration	Declares external schema variables, the values of which can be supplied by the agent launching the validation. Each variable is represented by a `gx:field` child element.	`gx:context`
Shapes library	A collection of shapes without target declaration, which can be referenced by other shapes	`gx:shapes`
Constraints library	A collection of constraint declaration nodes, which can be referenced by shapes	`gx:constraints`

Building block	**Role**	**XML representation**
Constraint components library	A collection of constraint component definitions, for which constraints can be declared	`gx:constraintComponents`
Constraint component definition	A user-defined constraint component. It declares the constraint parameters and provides a validator. Parameters are represented by `gx:param` child elements, the validator by a `gx:validatorXPath` or `gx:validatorFoxpath` child element, or a @validatorXPath or @validatorFoxpath attribute	`gx:constraintComponent`
Domain	A container element wrapping the shapes used for validating a particular file system tree, identified by its root folder	`gx:domain`
Resource shape	A shape applicable to a file system folder or file	`gx:folder` `gx:file`
Value shape	A shape applicable to a resource value	`gx:xpath` `gx:foxpath`
Focus mapper	Maps a resource to a focus value, or the items of a focus value to another focus value; contains value shapes to be applied to the focus value; may contain other Focus mappers using the focus value items as input	`gx:focusNode`
Base shape declaration	References a shape so that its constraints are included in the shape containing the reference	`gx:baseShape`

Constraint declaration node	An element representing one or several constraints declared by a shape. Constraint parameters are represented by attributes and/or child elements	`gx:fileSize`
		`gx:folderContent`
		`gx:hashCode`
		`gx:lastModified`
		`gx:mediaType`
		`gx:resourceName`
		`gx:targetSize`
		`gx:xsdValid`

Building block	Role	XML representation
Conditional node	A set of building blocks associated with a condition, so that the building blocks are only used if the condition is satisfied	`gx:ifMediatype`

6. Schema language extension

This section describes **user-defined constraint components**. Such components are defined within a greenfox schema by a `gx:constraintComponent` element, which specifies the constraint component name, declares the constraint parameters and provides an implementation. The implementation is an XPath or a foxpath expression, which accesses the parameter values as pre-bound variables. User-defined constraint components are used like built-in components: a constraint is declared by an element with attributes (or child elements) providing the parameter values and optional messages.

As an illustrative example, consider the creation of a new constraint component characterized as follows.

Constraint component IRI: `ex:grep`

Constraint parameters:

Name	Type	Meaning	Mandatory?	Default value
regex	`xsd:string`	A regular expression	+	-
flags	`xsd:string`	Evaluation flags	-	Zero-length string

Semantics:

"A constraint is satisfied if the focus resource is a text file containing a line matching regular expression `$regex`, as controlled by the regex evaluation flags given by `$flags` (e.g. case-insensitively)."

The **implementation** may be provided by the following element, added to the schema as a child element of `gx:constraintComponents`:

```
<constraintComponent constraintElementName="ex:grep">
    <param name="pattern" type="xs:string"/>
    <param name="flags" type="xs:string?"/>
    <validatorXPath>
        exists(unparsed-text-lines($this)[matches(., $pattern, $flags)])
    </validatorXPath>
</constraintComponent>
```

The *context item* supplied to the validator is assigned by the greenfox processor according to the following rules:

- If the constraint is used by a value shape: an item from the resource value
- If the constraint is used by a folder shape: the file path of the focus resource
- If the constraint is used by a file shape, the validator is an XPath expression and the file can be parsed into an XDM node tree: the root node of the node tree
- Otherwise the file path of the focus resource

Because of these rules, the example code uses the built-in variable `$this` which is always bound to the file path, rather than the context item (`.`) which may be the file path or a document node, dependent on the mediatype of the file.

The constraint can be used like this:

```
<file foxpath="...">
    ex:grep pattern="fbIx?" flags="i"
            msg="File does not contain string '$pattern'."
            msgOK="File contains string '$pattern'."/>
</file>
```

Note the variable references in the message text, which the greenfox processor replaces with the actual parameter values.

7. Validation results

This section describes the results produced by a greenfox validation.

7.1. Validation reports and representations

The primary result of a greenfox validation is an RDF graph called the **white validation report**. This is mapped to the **red validation report**, an RDF graph obtained by removing from a white report all triples not related to constraint violations. For red and white validation reports a **canonical XML representation** is defined. Apart from that, there are **derived representations**, implementation-dependent reports which may use any data model and mediatype.

The **white validation report** is an RDF graph with exactly one instance of `gx:ValidationReport`. The instance has the following properties:

- `gx:conforms`, with an `xsd:boolean` value indicating conformance
- `gx:result`, with one value ...
  - for each constraint violation (“red and yellow values”)
  - for each constraint check which was successful (“green values”)
  - for each observation, which is a result triggered by a value shape in order to record a resource value not related to constraint checking (“blue values”)

The **red validation report** is an RDF graph obtained by removing from the white validation report all green and blue result values. Note that the validation report defined by SHACL [7] corresponds to the red validation report defined by greenfox.

The **canonical XML representation** of a white or red validation report is an XML document with a `<gx:validationReport>` root element. It contains for each `gx:result` value from the RDF graph one child element, which is a `<gx:red>`, `<gx:yellow>`, `<gx:green>` or `<gx:blue>` element, according to the `gx:result/gx:severity` property value being `gx:Violation`, `gx:Warning`, `gx:Pass` or `gx:Observation`).

A **derived representation** is any kind of data structure, using any mediatype, representing information content from the white or red validation report in an implementation-defined way.

7.2. Validation result

A **validation result** is a unit of information which describes the outcome of validating a focus resource against a constraint: either constraint violation ("red" or "yellow" result), or conformance ("green" result).

A validation result is an RDF resource with several properties as described in Appendix C. Key features of the result model are:

- Every result is related to an individual file system resource (file or folder)
- Every result is related to an individual constraint (and, by implication, a shape)

This allows for meaningful aggregation by resource, by constraint and by shape, as well as any combination of aggregated resources, constraints and shapes. Such aggregation may be useful, e.g. for integrating validation results into a graphical representation of the file system, or for analysis of impact.

See Appendix C for a detailed description of the validation result model – RDF properties, SHACL equivalent and XML representation.

8. Implementation

An implementation of a greenfox processor is available on github [6]. The processor is provided as a command-line tool (`greenfox.bat`, `greenfox.sh`). Example call:

```
greenfox "val?gfox=/projects/greenfox/example-schemas/gfox-system-s.xml,
          domain=/projects/greenfox/example-systems/system-s"
```

The implementation is written in XQuery and requires the use of the BaseX [1] XQuery processor.

9. Discussion

Due to the rigorous framework on which it is based, the functionality of greenfox can be extended easily. Any number of new constraint components can be added without increasing the complexity of the language, as the *usage* of any constraint component follows the same pattern: select the component and assign the parameter values. Validation *results* likewise retain their simplicity, as their structure is immutable: a collection of result objects, reporting the validation of a single resource against a single constraint, expressed in a small and stable core vocabulary. New constraint components can be enhancements of the core language or extensions defined by user-defined schemas. Library schemas may give access to domain-specific sets of constraint components.

Another aspect of extension concerns the reuse of existing constraints and shapes. Reuse should be facilitated by refining the syntax and semantics of parameterizing and extending existing components. The value gain is immediate and the purity of the conceptual framework is not endangered.

The remainder of this discussion deals with the possibility to extend greenfox beyond adding new constraint components and refining techniques of component reuse. Care must be taken to avoid a hodgepodge of features increasing complexity and reducing uniformity, making further extension increasingly difficult and risky. Ideally, the future development of the language should be guarded by an architectural style as defined by Roy Fielding [2] – a set of architectural constraints. A good starting point is an attempt to take an abstract and fundamental view of the language.

Greenfox is **tree-oriented**, as a tree-structured perception of a file system is natural: a folder contains folders and files, a file (often) contains tree-structured information (XML, JSON, HTML, CSV, …). The expressiveness of greenfox can in large parts be attributed to the expressiveness of tree navigation languages (XPath, XQuery, foxpath), in combination with the suitability of the XDM model [8] for turning different mediatypes into a unified substrate for those languages.

On the other hand, greenfox is based on a rigorous conceptual framework which has been defined by SHACL [7], a **validation language for graphs** – without any relationship to tree structures. This apparent contradiction is resolved by identifying the *fundamental* concepts shared by the SHACL and greenfox languages, distinguishing them from derived concepts accounting for all the outward differences. Such fundamental concepts are:

1. itemization of information
2. identification of a subset of items with resources
3. constraint check: resource + constraint parameters = true/false + details
4. itemization of validation: one resource against one constraint
5. itemization of validation results: one unit per pair of resource and constraint
6. resource interface model: resource properties and resource values

7. resource value model: a mapping of resource property or resource value to a value

The degree of abstraction makes it unnecessary to prescribe the data model (RDF / XDM), the alignment between items and resources (RDF-nodes / Files +Folders), the value mapping languages (SPARQL / XPath+foxpath). The conceptual foundation is equally well-suited for supporting an RDF or an XDM based view.

This perception can give guidance for the further development of greenfox. Greenfox differs from other validation languages in its main goal which is a **unified view on system validity**, integrating any resources which can be accommodated in a file system. Greenfox is intent on hiding outward heterogeneity (e.g. of mediatype) behind rigorous abstractions. In this field, RDF has *very much* to offer, as it separates information content from its representation in a most radical way. There is no reason not to also consider the use of RDF nodes as resource values, or to use RDF expressions as vehicles of mapping and navigation. The integration of graph and tree models, the combination of their complementary strengths, holds considerable promise for anyone interested in unified views of information. In spite of its deep commitment to a tree-oriented data model and expression languages built upon it, the greenfox language might in due time integrate with graph technology in order to offer yet more comprehensive answers to problems of validity.

A. Greenfox schema "system S"

This appendix lists the complete schema developed in Section 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<greenfox greenfoxURI="http://www.greenfox.org/ns/schema-examples/system-s"
   xmlns="http://www.greenfox.org/ns/schema">

   <!-- *** External context *** -->
   <context>
      <field name="lastModified" value="2019-12-01"/>
   </context>

   <!-- *** System file tree *** -->
   <domain path="\tt\greenfox\resources\example-system\system-s"
           name="system-s">

      <!-- *** System root folder shape *** -->
      <folder foxpath="." id="systemRootFolderShape">

         <!-- *** XSD folder shape *** -->
         <folder foxpath=".\\resources\xsd" id="xsdFolderShape">
```

```
   <targetSize count="1"
               countMsg="No XSD folder found"/>
   <file foxpath="*.xsd" id="xsdFileShape">
      <targetSize minCount="1"
                  minCountMsg="No XSDs found"/>
   </file>
</folder>

<!-- *** Codelist folder shape *** -->
   <folder foxpath=".\\resources\codelists"
           id="codelistFolderShape">
      <targetSize count="1"
                  countMsg="No codelist folder found"/>

      <!-- # Check - folder contains codelists? -->
      <foxpath
         expr="*.xml/codelist[entry]/@name"
         minCount="1"
         minCountMsg="Codelist folder without codelists"
         itemsUnique="true"
         itemsUniqueMsg="Codelist names must be unique"/>

      <file foxpath="*[is-xml(.)]" id="codelistFileShape">
         <targetSize minCount="1"
                     minCountMsg="No codelist files found"/>
      </file>
   </folder>

   <!-- *** Testcase folder shape *** -->
   <folder foxpath=".\\test-*[input][output][config]"
           id="testcaseFolderShape">
      <targetSize minCount="1"
                  minCountMsg="No testcase folders found"/>

      <!-- # Check - test folder content ok? -->
      <folderContent
         closed="true"
         closedMsg="Testcase contains member other than
                    input, output, config, log-*.">
         <memberFolders names="input, output, config"/>
         <memberFile name="log-*" count="*"/>
      </folderContent>

      <!-- *** msg config shape *** -->
      <file foxpath="config\msg-config.csv" id="msgConfigFileShape"
            mediatype="csv" csv.separator="," csv.withHeader="yes">
```

```
   <targetSize count="1"
               countMsg="Config file missing"/>

   <!-- # Check - configured return codes expected? -->
   <xpath expr="//returnCode"
          inMsg="Config file contains unknown return code">
      <in>
         <eq>OK</eq>
         <eq>NOFIND</eq>
         <like>ERROR_*</like>
      </in>
   </xpath>
</file>

<!-- *** Request file shape *** -->
<file foxpath="input\(*.xml, *.json)"
      id="requestFileShape">
   <targetSize
      minCount="1"
      minCountMsg="Input folder without request msgs"/>

   <!-- # Check - request with response? -->
   <foxpath
      expr="..\..\output\*\file-name(.)"
      containsXPath=
           "$fileName ! replace(., '(.*)RQ(.*)$', '$1RS$2')"
      containsXPathMsg="Request without response"/>
</file>

<!-- *** Response file shape *** -->
<file foxpath="output\(*.xml, *.json)"
      id="responseFileShape"
      mediatype="xml-or-json">
   <targetSize
      minCount="1"
      minCountMsg="Output folder without request msgs"/>

   <!-- # Check - response fresh? -->
   <lastModified ge="${lastModified}"
                 geMsg="Stale output file"

   <!-- # Check - response non-empty? -->
   <fileSize gt="0"
             gtMsg="Empty output file"/>

   <!-- # Check - schema valid? (only if XML) -->
```

```
                    <ifMediatype eq="xml">
                       <xsdValid xsdFoxpath="$domain\resources\xsd\\*.xsd"
                                 msg="Response msg not XSD valid"/>

                    </ifMediatype>

                    <!-- # Check - known article number? -->
                    <xpath
                       expr="//*:fooValue"
                       inFoxpath="$domain\\codelists\*.xml
                                  /codelist[@name eq 'foo-article']/entry/
@code"
                       inFoxpathMsg="Unknown foo article number"
                       id="articleNumberValueShape"/>

                    <!-- # Check - return code ok? -->
                    <foxpath
                       expr="..\..\config\msg-config.csv\csv-doc(., ',',
'yes')
                             //record[response eq $fileName]/returnCode"
                       eqXPath="//
*:returnCode"
                       eqXPathMsg="Return code not the configured
value"/>

                </file>
             </folder>
          </folder>
       </domain>
</greenfox>
```

B. Alignment of key concepts between greenfox and SHACL

This appendix summarizes the conceptual alignment between greenfox and SHACL. The striking correspondence reflects our decision to use SHACL as a blueprint for the conceptual framework underlying the greenfox language. Greenfox can be thought of as a combination of SHACL's abstract validation model with a view of the file system through the prism of a unified value model (XDM), supporting powerful expression languages (XPath/XQuery + foxpath).

The alignment is described in two tables. The first table provides an aligned definition of the validation process as a decomposable operation as defined by greenfox and SHACL. The second table is an aligned enumeration of some building blocks of the conceptual framework underlying greenfox and SHACL.

Table B.1. Greenfox/SHACL alignment, part 1: validation model

Greenfox operation	**SHACL operaration**
Validation of a file system against a greenfox schema	Validation of a data graph against a shapes graph
= Union of the results of the validation of the file system against all shapes	= Union of the results of the validation of the data graph against all shapes
Validation of a file system against a shape	Validation of a data graph against a shape
= Union of the results of all focus resources in the target of the shape	= Union of the results of all focus nodes in the target of the shape
Validation of a focus resource against a shape = Union of the results of the validation of the focus resource against all constraints declared by the shape	Validation of a focus node against a shape = Union of the results of the validation of the focus node against all constraints declared by the shape
Validation of a focus resource against a constraint = function(constraint parameters, focus resource, resource values)	Validation of a focus node against a constraint = function(constraint parameters, focus node, property values)
Resource values = XPath|foxpath, applied to a resource	Property values = SPARQL property path, applied to a node

Table B.2. Greenfox/SHACL alignment, part 2: conceptual building blocks

Greenfox concept	**SHACL**	**Remark**
Resource shape: • Folder shape • File shape	Node shape	Common key concept: shape = set of constraints for a set of resources
Focus resource	Focus node	Common view: validation can be decomposed into instances of validation of a single focus against a single shape
Target declaration • Foxpath expression • Literal file system path	Target declaration • Class members • Subjects of predicate IRI • Objects of predicate IRI • Literal IRI	Difference: in greenfox a target declaration is essentially a navigation result, in SHACL it tends to be derived from class membership (ontological)

Greenfox concept	**SHACL**	**Remark**
Resource value	Value node	Common view: non-trivial validation requires mapping resources to values
Mapping resource to value: • XPath expression • Foxpath expression	Mapping resource to property: • SPARQL property path	Common view: the mapping of a resource to a value is an expression
Value shape: • XPath shape • Foxpath shape	Property shape	Common view: usefulness of an entity combining a *single* mapping of the focus resource to a value with a *set of constraints* for that value
Constraint declaration • Constraint component • Constraint parameters	Constraint declaration • Constraint component • Constraint parameters	Common view: a constraint declaration can be thought of as a function call
Constraint component • Signature • Mapping semantic	Constraint component • Signature • Mapping semantic	Common view: a constraint component can be thought of as a library function
Validation report • Constraint violations • Constraint passes • Observations	Validation report • Constraint violations	Common view: a result is an RDF resource; difference: in greenfox also successful constraint checks produce results (“green results”); difference: in greenfox also observations can be produced, results unrelated to constraint checking (“blue results”)
Extension language: • XPath/XQuery expression • foxpath expression	Extension language: • SPARQL SELECT queries • SPARQL ASK queries	Common view: extension of functionality is based on an expression language for mapping resources to values and values to a result

Greenfox concept	SHACL	Remark
Mediatype integration: • Common data model • Common navigation model	-	Difference: in contrast to SHACL, greenfox faces a heterogeneous collection of validation targets, calling for integration concepts

C. Validation result model

This appendix defines the validation result model.

In the table below, the XML representation is rendered as an XPath expression to be evaluated in the context of the XML element representing the result, which is a `<gx:red>`, `<gx:yellow>`, `<gx:green>` or `<gx:blue>` element. Apart from the result properties shown in the table, individual constraint components may define additional properties.

Table C.1. The validation result model – RDF properties, description, corresponding SHACL result property and XML representation.

RDF property	Description	SHACL result property	XML representation
`gx:severity`	The possible values: • `gx:Violation` • `gx:Warning` • `gx:Pass` • `gx:Observation` While `gx:Observation` is a value not related to a constraint check, the other ones represent constraint violations or a successful check	`sh:severity`	Local name of the result representing element: • `red` = `gx:Violation` • `yellow` = `gx:Warning` • `green` = `gx:Pass` • `blue` = `gx:Observation`

RDF property	**Description**	**SHACL result property**	**XML representation**
`gx:fileSystem`	Identifies the file system validated	An aspect of `sh:focusNode`	`ancestor::gx:validationReport/@fileSystemURI`
`gx:focusResource`	File path of a file or folder resource	An aspect of `sh:focusNode`	`@filePath`
`gx:focusNode`	XPath of a node within an XDM node tree representing the contents of a file resource	`sh:focusNode`	`@nodePath`
`gx:xpath`	The XPath expression of a value shape	`sh:resultPath`	`@expr` or `./expr` + `@exprLang="XPath"`
`gx:foxpath`	The foxpath expression of a value shape	`sh:resultPath`	`@expr` or `./expr` + `@exprLang="foxpath"`
`gx:value`	A resource value, or single item of a resource value, causing a violation	`gx:value`	`@value` or `value` A value consisting of several items is represented by a sequence of `value` child elements
`gx:valueCount`	Number of resources in a target, or of resource value items, causing a violation	-	`@valueCount`

RDF property	**Description**	**SHACL result property**	**XML representation**
`gx:sourceShape`	The value shape or resource shape defining the constraint; the value is the @id value on the shape element in the schema if present, or a value assigned by the greenfox processor otherwise	`gx:sourceShape`	`@resourceShapeID`, or `@valueShapeID`
`gx:constraint-Component`	Identifies the kind of constraint	`sh:constraint-Component`	`@constraintComp`
`gx:message`	A message communicating details to humans; the value is the `@msg` or `@...Msg` attribute or `<msg>` or `<...Msg>` child element value on the shape or constraint element in the schema, or a value assigned by the greenfox processor. In the above, … is a prefix identifying the constraint to which the message relates. Examples: `@minCountMsg`, `@exprValueEqMsg`.	`sh:message`	`@msg` or `msg` + `msg/@xml:lang` A message with a language tag is represented by a child element with language attribute.

D. Note on the generation of resource values by expression chains

The recursive definition of *resource values* allows the construction of resource values through **chains of expressions**. When a chain is used, each combination of items from all expressions except the last one is mapped to a distinct resource

value, which itself may have zero, one or more items. As an example, consider a first expression mapping a folder to a sequence of files, a second expression mapping each file to all `<row>` elements found in its node tree representation, and a final expression mapping each `<row>` element to its `<col>` child elements. This chain generates one resource value for each combination of file and row, consisting of zero, one or more `<col>` elements. These values are resource values of the folder to which the expression chain was applied.

Bibliography

[1] *BaseX*. 2020. BaseX GmbH. `http:// basex.org`

[2] *Architectural Styles and the Design of Network-based Software Architectures.*. 2000. Roy Fielding. `https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`

[3] *FOXpath - an expression language for selecting files and folders.*. 2016. Hans-Juergen Rennau. `http://www.balisage.net/Proceedings/vol17/html/Rennau01/BalisageVol17-Rennau01.html`

[4] *FOXpath navigation of physical, virtual and literal file systems.*. 2016. Hans-Juergen Rennau. `https://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf`

[5] *foxpath - an extended version of XPath 3.0 supporting file system navigation.*. Hans-Juergen Rennau. 2017. `https://github.com/hrennau/shax`

[6] *Greenfox - a schema language for validating file system contents and, by implication, real-world systems.*. Hans-Juergen Rennau. 2020. `https://github.com/hrennau/greenfox`

[7] *Shapes Constraint Language (SHACL)*. 2017. World Wide Web Consortium (W3C). `https://www.w3.org/TR/shacl/`

[8] *XQuery and XPath Data Model 3.1*. 2017. World Wide Web Consortium (W3C). `https://www.w3.org/TR/xpath-datamodel-31/`

[9] *XML Path Language (XPath) 3.1*. 2017. World Wide Web Consortium (W3C). `https://www.w3.org/TR/xpath-31/`

[10] *XPath and XQuery Functions and Operators 3.1*. 2017. World Wide Web Consortium (W3C). `https://www.w3.org/TR/xpath-functions-31/`

[11] *XQuery 3.1: An XML Query Language*. 2017. World Wide Web Consortium (W3C). `https://www.w3.org/TR/xquery-31/`

Use cases and examination of XML technologies to process MS Word documents in a corporate environment

Toolset to test and improve the quality and consistency of styling in MS Word

Colin Mackenzie
Mackenzie Solutions
<colin@mackenziesolutions.co.uk>

Abstract

In recent years XML has been replaced by JSON as the preferential format of the API community and most non-SQL database vendors are not focused on using XML for storage of non-tabular data. This has meant that the use of XML and its accompanying technologies has retreated somewhat back to its origin as a method of structuring and processing documents. The majority of specialist XML developers using XML tools work within traditional publishers, divisions of government or technical publishers successfully delivering quality and diverse publications though complex workflows. While the obvious preference is for authors to generate semantically rich documents using XML editors, many of these XML publishers are faced with converting and improving documents originated in MS Word and utilize XML technologies as part of this process. But the majority of professional Word documents are not generated for publishers but instead are created within corporate environments. If XML technology can be applied to this problem space it could provide a significant boost to the continued adoption of these technologies.

This paper will investigate some of the use cases for processing Word documents found in the corporate environment (focusing on improving quality) and will demonstrate using a toolset developed in XProc and XSLT3 that open standard XML technologies can provide significant advantages.

Keywords: XML, XSLT3, XProc, OOXML, Ms Word, Quality

1. The problem with styles and Word

There are few ubiquitous tools in IT, but Microsoft Word™ probably comes as close as there is. With only a few exceptions (where a web-only deliverable means

the content is authored directly into HTML or where complex re-use and professional publication requirements mandates the use of XML) we all use Word to author important documents. Whether the documents are internal reports, legal contracts or consultancy proposals it is vital that the documents:

- reflect the latest corporate brand;
- are consistent with other documents being delivered;
- uses the agreed numbering system (via auto numbered paragraphs that can be dynamically referenced and chapter/appendix prefixes);
- automatically create the correct table of contents (and table of figures/tables if required);
- can be easily edited by others; and
- are able to have content extracted and re-used in other documents or libraries of information.

This is only achievable in Word via the consistent use of styles in well managed templates. However, even if your organization has developed and maintained these templates, documents will frequently have their consistency (and therefore quality) reduced due to:

- use of old templates;
- creation of Word documents from existing documents that do not use the latest template;
- editing of the document outside of the organization-controlled environment (e.g. sending contracts to "the other side"); and
- user error where formatting is applied manually (via buttons, format painter etc.) or where ad-hoc styles are created and used.

It is vital that we do not underestimate the issue of user error. Most business users are never trained in Word as, in its simplest form, it is so easy to use. But is not easy to use Word in the right way to achieve consistency (especially in documents that require complex numbering and multi-level lists) even in the most macro-heavy templated environment. Many of us have had to take over complex Word documents from business users in order to try to decipher what has gone wrong and make last minute edits before deadlines. In many other cases these last-minute edits are made blind "I just changed things till it looked right" at the cost of consistency and any other users of the document.

With typical Word workflows, errors in the styles being applied will directly result in presentation errors in the final delivered documents (as the delivery format is Word or PDF). In more complex publishing workflows, the Word documents may be:

- converted and formatted using InDesign;
- converted to HTML for web publishing; or

- converted to XML for enrichment and/or multi-format delivery.

In all of these more complex workflows the correct use of Word styles is pivotal to the success of the process in order to convert, brand or structure the data appropriately with missing or misused styles leading to invalid or substandard content.

Typical issues include:

- application of styles to wrong content/in wrong order;
- use of manual mark-up instead of styles (or overriding styles to mimic other styles);
- creation and use of unsupported styles (styles not defined in master template);
- use of out of date styles/templates;
- manual numbering (and chapter/appendix prefixing); and
- lack of metadata (missing or incomplete properties or fields).

So how do you know if your document has issues never mind being able to correct them? Lack of style consistency/quality across thousands of documents would substantially increase the cost of any project designed to utilize that library as a consistent data set (and may even call the financial viability of the project into doubt).

2. Non-XML solutions

Despite the volume of licenses sold to the corporate market, Microsoft have not really focused on providing product features to increase the quality/consistency of styling in documents created by Word. While manual procedures are available these are not ideal as manual means "subject to human error" and they do not tell you if the latest/correct version of the style itself is in use.

Historically styling solutions were all based around macros/plug-ins within Word or client-side automation using Word itself. Typical approaches taken to ensure quality of styles mostly fall into the following categories:

- Template management: forcing the user to pick from one of a number of centrally managed templates or auto-loading a central template from a network drive when creating a new document.
  - But what if the user opens an old document or one sent in from a third-party and then saves it with a new name?
- Customized editing experience: providing custom ribbons and dialogue boxes that aide the user by applying the correct style (somehow made more obvious via an icon?) of the many approved styles to a given paragraph.
  - But what if a user applies styles or formatting manually (if users are not trained in Word they will almost certainly get little training in any add-ons), does not apply any style or even does not enter content that is consid-

ered mandatory in a given scenario (e.g. all groups of "Warning Paras" must be preceding by a "Warning Title")?

- Document analysis and repair – Provides reports on style use and a custom user interface to allow users to manually apply a selected style to one or more paragraphs. Some of these tools can also spot hard coded textual references (e.g. "see clause 4.2" and replace them with dynamic Word cross references).
  - Can the "rules" for the styles be easily kept up to date as the template(s) changes?

Those who utilize these solutions find that over time there tends to be an issue maintaining them. Issues have included:

- The solution no longer works since Word was upgraded (incompatible macros/plug-ins).
- The solution no longer works since the template was upgraded (the template designer does not understand the style solution and IT do not understand complex Word templates).
- Security changes (in Windows or in the organization) mean that the client-side code no longer runs.

If the code that is trying to identify style issues and/or fix those issues also has to contain the business rules then the process logic and business logic gets muddled. Some tools utilize configuration files listing style names that are allowed and old style names that should be mapped to the new style names. However, logic such as "do not allow a 'Clause Level 2' unless it is preceded by a 'Clause Level 1'" is not easily expressed in a simple look-up table never mind more complex logic that may look up multiple paragraphs in order to decide what is valid and may also utilize text pattern matching (e.g. if a heading matches the pattern "Appendix [A-Z]" then it should use "Appendix heading" style).

It would surely be preferable to utilize XML technologies to:

- Use a standard language to define what the rules are for the styling and content of a document (and how they can be fixed) in a way that supports the maintenance of the logic separately from both Word and the program that utilizes these rules.
- Check that the latest style and numbering definitions themselves are in use.
- Find a process that does not need to be installed on the client machine so it is easier to maintain.
- Apply fixes wherever automatable.
- Report issues back to users using standard Word features.
- Provide reports on libraries of documents summarizing the level of compatibility to current style rules.

3. A standards-based solution

In 2003 Microsoft created a public standard for an XML specification (Microsoft Office XML) that could be imported or exported from MS Word 2003. For the first time, developers could safely generate (or more easily adapt/transform) Word documents outside of the MS Word application. This allowed automation solutions to be developed for business challenges such as:

- conversion from Word to XML for publishers;
- creation of customized contracts (with appropriate clauses inserted based-on information gathered) and whose style reflects the corporate Word template; and
- personalized reporting/marketing material (e.g. "your pension performance explained").

The single file format became a favorite for XML developers to transform via XSLT to whatever output was required but this approach was rarely adopted outside the publishing community or bespoke products.

Microsoft replaced that standard in later years with the ISO standard "Office Open XML" (OOXML) ultimately becoming the default read and write formats for MS Word (i.e. ".docx"). As most XML developers know, Docx files are a zipped set of folders containing XML files for the text, style, comments (plus graphics) required for a Word document. This new format allows developers to work directly with the core document format of MS Word but needs the developer to have the ability to "unpack" the files, update multiple files before repackaging as a ".docx". This meant many XSLT developers (as XSLT cannot yet open ZIP files) stuck to the old format.

When investigating the suitability of the use of the latest XML toolsets for processing Word, we decided to develop a solution for checking, reporting and fixing Word issues. In order to have access to the complete Word data set, we decided to use OOXML and therefore turned to XProc. XProc provides many built-in steps that makes it perfect for processing Docx files. These steps include the ability to unzip, validate, compare, merge and manipulate XML, transform via XSLT and zip the results back to a ".docx" file.

Having dealt with the zipping and unzipping of documents, we needed a way to check the consistency and quality of the document style and content. While it is easy to validate the individual Word XML files against a schema (the "Office Open XML" schema), this only checks that the XML structure within the file matches what is expected but does not check compliance against any business-specific rules such as style conformance or mandatory text content.

Fortunately, Schematron allows a document analyst to define whatever simple or complex rules that are required to check the quality of a document and to provide information back to the business users on how to correct any issues. An

example of a Schematron rule to test that a paragraph with a paragraph style "Heading 3" must be immediately preceded by a paragraph with style "Heading 2" is as follows.

```
<sch:rule context="p:para[cm:getParaStyle(.)='Heading3'">
            <sch:let name="vPrecedingStyleName"
value="ms:getParaStyle(ms:getPreviousPara(.))"/>
            <sch:assert test="$vPrecedingStyleName ='Heading2'"
id="H3afterH2">Heading 3 must be immediately preceded by Heading2 (para
before actually has style '<sch:value-of select="$vPrecedingStyleName"/
>')</sch:assert>
</sch:rule>
```

As these rules are declarative and separate from any logic used to process the Word file itself, a document analyst is free to develop and maintain these rules without having to be an expert programmer. The Schematron format is an open standard (with plenty of documentation and training material on the web) that utilizes the XPath standard as the way to identify content in order to test its validity. Developers can simplify commonly-used complex paths by defining custom variables or functions such as "getParaStyle". These rules can check for the existence and validity of fields, metadata or that content of a certain type has text that fits a particular pattern (using regular expressions). If required, a library of these tests can be created and re-used as required.

Once a document has been processed by the tool, the errors or warnings from Schematron are presented back to the user as Word comments (from pseudo "Error" or "Warning" users) with the location of the comment providing the context for the error. Users can utilize Word's review toolbar to navigate their way through the comments.

As you can see from Figure 1, errors can be reported not only on erroneous application of styles or formatting but also where the text itself does not match the "house style" for this sort of document (e.g. the use of punctuation in lists).

Once a user remedies the issue (e.g. by changing style to the correct style or by moving an existing paragraph into the correct position) the file can be reprocessed allowing the existing errors/warnings to be stripped and any new or remaining issues to be created as new comments. This is not the first solution to suggest using Schematron with Office documents with author feedback provided as comments (see [1]). Our goes further by:

- Implementing the process in XProc allowing further steps and options to be developed;
- Focusing on business cases other than those of supporting XML conversion from Word.
- Enhancing the usability of the feedback provided to the users.

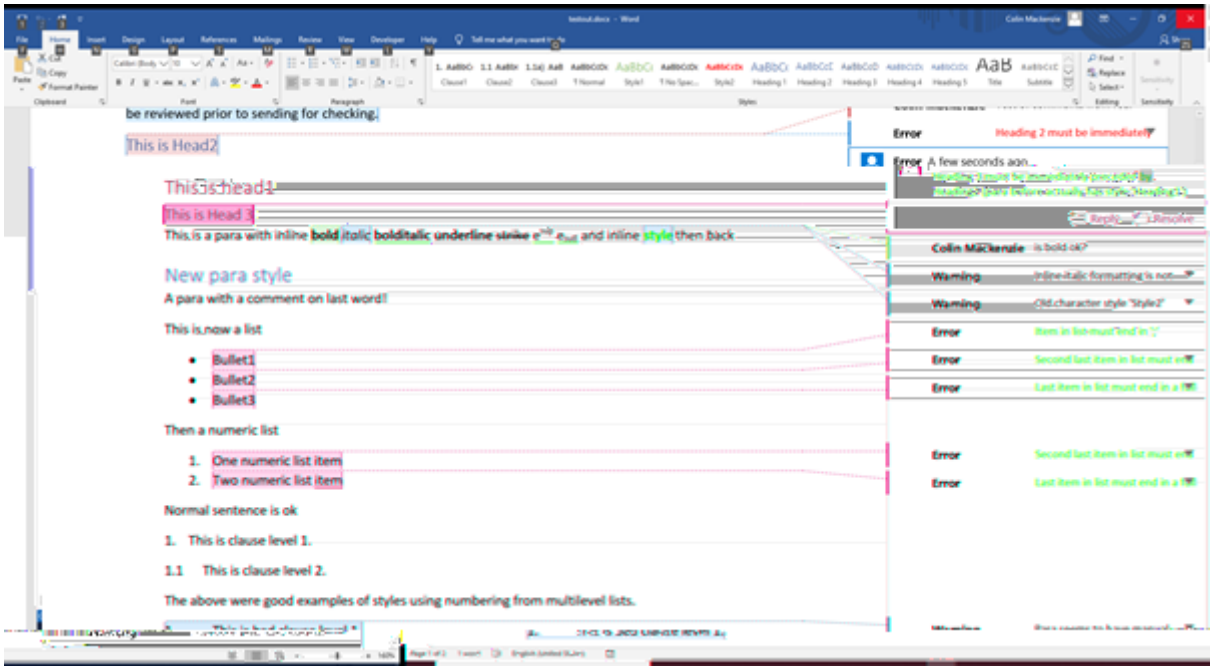


Figure 1. Screenshot of document with Schematron errors

- Performing the checks on native ".docx" files.
- Detecting the type of document and selecting the correct Schematron rule files to use to check that file (therefore supporting general rules, corporate rules and template/content specific rules).
- Checking that the styles and numbering used in the document matches those in a reference master style file.
- Providing options to strip existing user generated comments (important before final delivery of a document) or to keep those comments.
- Running configurable pre and post quality XSLT transformation pipelines based on the template used for the document.
- Providing users with a choice of fixes that can be manually applied or, in some cases, automated during re-processing (in XSLT steps prior to checking quality).

The ability to apply different quality check and resolution XSLTs per template enables the solution to be run across a gamut of corporate documents types (and versions of those templates) with different business rules per template but without duplicating or disseminating logic. The format for the configuration files is as follows.

```
<validateConfig>
    <entry default="true">
        <template>specification.dotm</template>
        <template>proposal.dotm</template>
```

```
            <schematron>testWordStyle.sch</schematron>
            <schematronFix>testWordStyleFix.sch</schematronFix>
            <pre-xslt>wordContentFixes.xsl</pre-xslt>
            <post-xslt>postFix1.xsl</post-xslt>
            <masterStyleFile>contractMasterStyles.xml</masterStyleFile>
            <masterNumberFile>contractMasterNumbering.xml</masterNumberFile>
        </entry>
        <entry>
            <template>contract.dotm</template>
            <template>contractNew.dotm</template>
            <schematron>testWordStyle2.sch</schematron>
            <schematronFix>testWordStyleFix.sch</schematronFix>
            <pre-xslt>preFix1.xsl</pre-xslt>
            <post-xslt>preFix2.xsl</post-xslt>
            <masterStyleFile>masterStyles2.xml</masterStyleFile>
            <masterNumberFile>masterNumbering.xml</masterNumberFile>
        </entry>
    </validateConfig>
```

By providing the rules developer (and XSLT fix developer) with access to the source document (plus the definition of its styles and numbering all combined into one XML file) along with the master template's styles and numbering, the solution can try to ensure that not only is the appropriate style name being used but that the style definition (and any associated autonumbering) matches the correct corporate standard. Errors that are not related to a particular line of content (such as mismatch in the style definitions or in Word Properties) are added automatically as paragraphs at the start of the file (see Figure 2).

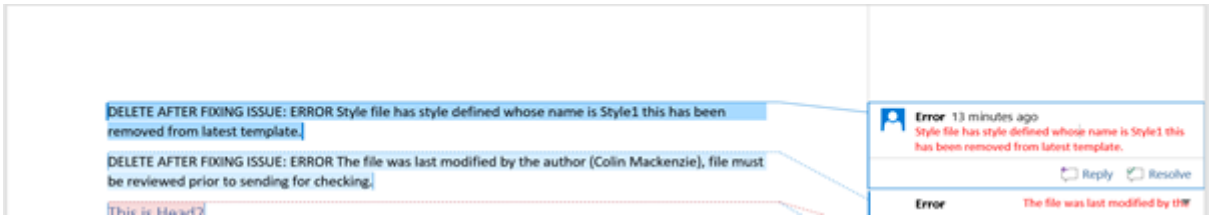


Figure 2. Screenshot of style and metadata errors

While the content of styles (as opposed to the application of particular style names to content) may not be important for those simply converting Word to XML, for those use cases where the Word document will go on to be edited or combined with other Word documents it is important that the style/numbering information has not been overridden locally (in this para) or within the styles defined in this particular document.

Further, where manual numbering has been applied, this can be identified and suggestions made to the author as to what corporate styles are available that

have numbering formats that match the manual numbers applied by the author. This is shown in the Schematron rule below.

```
<sch:rule context="w:p">
            <sch:let name="vNumStr" value="ms:getManualNumber(.)"/>
            <sch:let name="vSuggestedStyleNames"
value="ms:getSuggestedStyleNames(.,$vNumStr)"/>
            <sch:report test="$vNumStr " id="ManualNumber1"
role="warning">Para seems to have manual number '<sch:value-of
select="$vNumStr"/>': consider replacing using styles <sch:value-of
select="$vSuggestedStyleNames"/></sch:report>
        </sch:rule>
```

This rule includes calls to some helper functions (provided with the solution) to make the task of defining custom rules easier. The logic for the function to identify manual numbering (in this case looking for certain numbering patterns at the start of the text followed by a tab) is relatively simple but ensures the analyst does not need to have a deep knowledge of OOML. The logic to find suggested style names based on the hard-coded numbering is much more complex and would be beyond the ability of a corporate developer as it requires an in-depth understanding of the list and style definitions within OOXML. As this code dynamically checks what styles are recommended that meets the numbering required, the code does not need to be updated as new styles/multi-level lists are defined in the master template making maintenance much easier.

The XProc process also supports recording the quality of the document in an XML log file so that an entire library of documents can be checked for style conformance which is especially important when beginning a new project that requires consistency of content. The log(s) can be queried (e.g. using xQuery) or transformed (e.g. for loading into Excel) to provide business intelligence on a batch of documents.

```
<log date="2019-11-12">
   <entry stylesMatch="true"
          errorCount="4"
          warnCount="1"
          issues="H1notfirst H3afterH2 Bullet2 NumOne"
          warnings="NoI"
          filename="test.docx"
          startDateTime="2019-11-12T16:37:49.614Z"
          endDateTime="2019-11-12T16:37:49.621Z"/>
```

This XProc process can be invoked in a number of ways depending on the business requirement and IT limitations:

- Run on current Word file from custom macro or Add-in to Word (with the solution client-side or posted to a server application).

- Invoked from a workflow, content management or publishing solution as part of a "check" stage using Java or by running a BAT file.
- Run from PowerShell when a file arrives in a specific network folder.
- Run from a Bat file on a hierarchical folder full of Word files.
- Run from XML processing tools such as Oxygen.

4. Providing fixes using a "QuickFix" like approach

We have already described how the solution provides Word users with visibility of business logic errors that have been defined using Schematron and how normal development approaches (applying a configurable list of XSLTs to the entire document) could be used to fix those errors. However, there are circumstances where an interaction with the author is required in order to fix a problem in a way that does not result in more work rather than less. If there is a rule where a para with a "Heading 2" style must be immediately preceded by a "Heading 1" (or another "Heading 2") then a number of fixes are possible including:

- Insert a "Heading 1" before the current "Heading 2" ready for the author to put in the main heading; or
- Change the current "Heading 2" to be a "Heading 1" (say if the preceding para is not a "Heading1" already); or
- Change the current "Heading 2" to be a non heading paragraph.

It *may* be possible to determine the best approach (based on the styles of surrounding paragraphs) but in many cases it is not possible. Fortunately, there is a precedent for providing users with choices of fixes that can then be automatically applied – Schematron QuickFix (SQF).

We considered defining fixes using the standard QuickFix grammar and then include an existing QuickFix processor into our pipeline but decided for the initial solution to define and implement the fixes by dynamically calling functions that abstract away the complexity of OOXML. The following code is the Schematron for a "Heading 2" example that illustrates where XSLT can be embedded (possibly an undocumented feature of the Schematron processor) to choose a fix action or offers the choice to the user of multiple fix options.

```
<sch:rule context="w:p[ms:getParaStyle(.)='Heading3']">
            <sch:let name="vPrecedingStyleName"
value="ms:getParaStyle(ms:getPreviousPara(.))"/>
            <sch:assert test="$vPrecedingStyleName='Heading2'"
id="H3afterH2">Heading 3 must be immediately preceded by Heading2 (para
before actually has style '<sch:value-of
select="$vPrecedingStyleName"/>')
                <cmqf:fixes>
                    <xsl:choose>
```

```
                        <xsl:when test="$vPrecedingStyleName='Heading1'">
                            <cmqf:fix id="ChangeStyle-Heading2">
                                <cmqf:description><cmqf:title>Change
style to Heading2</cmqf:title></cmqf:description>
                            </cmqf:fix>
                        </xsl:when>
                        <xsl:otherwise>
                            <cmqf:fix id="ChangeStyle-Heading1">
                                <cmqf:description><cmqf:title>Change
style to Heading1 OR</cmqf:title></cmqf:description>
                            </cmqf:fix>
                            <cmqf:fix id="ChangeStyle-Normal">
                                <cmqf:description><cmqf:title>Change
style to Normal</cmqf:title></cmqf:description>
                            </cmqf:fix>
                        </xsl:otherwise>
                    </xsl:choose>
                </cmqf:fixes>
            </sch:assert>
        </sch:rule>
```

For the example where manual numbering was applied to a paragraph rather than using one of the suggested styles that would achieve that numbering, we would add fixes where we remove the manual numbering then additionally apply a suitable style.

```
<sch:rule context="w:p">
            <sch:let name="vNumStr" value="ms:getManualNumber(.)"/>
            <sch:let name="vSuggestedStyleNames"
value="ms:getSuggestedStyleNames(.,$vNumStr)"/>
            <sch:report test="$vNumStr" id="ManualNumber1"
role="warning">Para seems to have manual number '<sch:value-of
select="$vNumStr"/>': consider replacing using styles <sch:value-of
select="$vSuggestedStyleNames"/>
             <cmqf:fixes>
                <cmqf:fix id="RemoveManualNumber">
                    <cmqf:description><cmqf:title>Remove manual number
<xsl:value-of select="$vNumStr"/></cmqf:title></cmqf:description>
                </cmqf:fix>
                <xsl:for-each select="$vSuggestedStyleNames">
                    <cmqf:fix id="ChangeStyle-{.}">
                        <cmqf:description><cmqf:title>Change style to
<xsl:value-of select="."/></cmqf:title></cmqf:description>
                    </cmqf:fix>
                </xsl:for-each>
            </cmqf:fixes></sch:report>
        </sch:rule>
```

In order to provide the fix suggestions back to the user we again use the Word comment facility by generating comments by a pseudo user called "Fix" where the comment text makes sense to the user but also contains enough information to allow the pipeline to implement the fix by dynamically constructing a function call with suitable arguments.

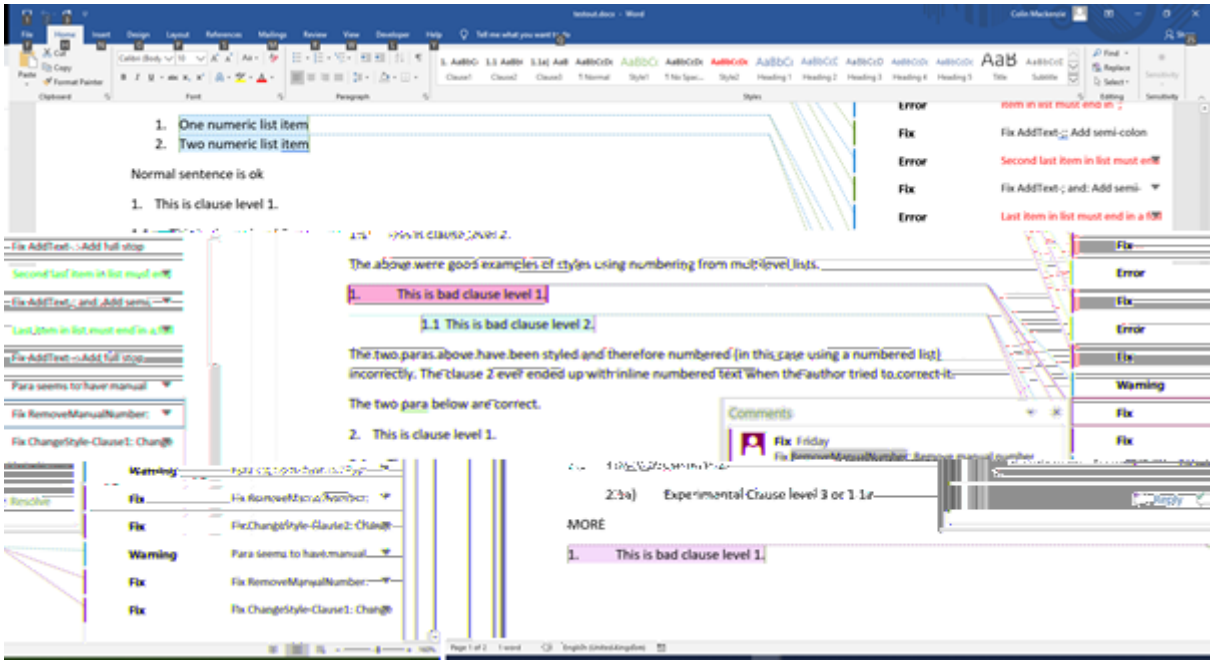


Figure 3. Screenshot showing RemoveManualNumber fix

Figure 4. Screenshot showing ChangeStyle fix

In the examples shown in Figure 3 and Figure 4, the fixes will be applied using two functions:

- "RemoveManualNumber" – this function will ensure that the manual number is dropped from the text; and
- "ChangeStyle" – this function will change the style to the style name passed as an argument (in this case "Clause1").

Other scenarios (e.g. adding punctuation to simple lists) will use other helper functions provided with the framework (e.g. “AddText”) or new custom functions defined by the client’s document analyst.

Word authors can still make any manual change required and simply delete any “Fix” comment that is no longer applicable or is not their preferred option (where there is a choice of fixes).

When the fixes are applied, any existing Schematron error/warning/fix comments are dropped and the fixed document is revalidated by Schematron in case any new issues have arisen.

If the user leaves these fix comments in place then reprocesses the document though the solution, the following output would be achieved.

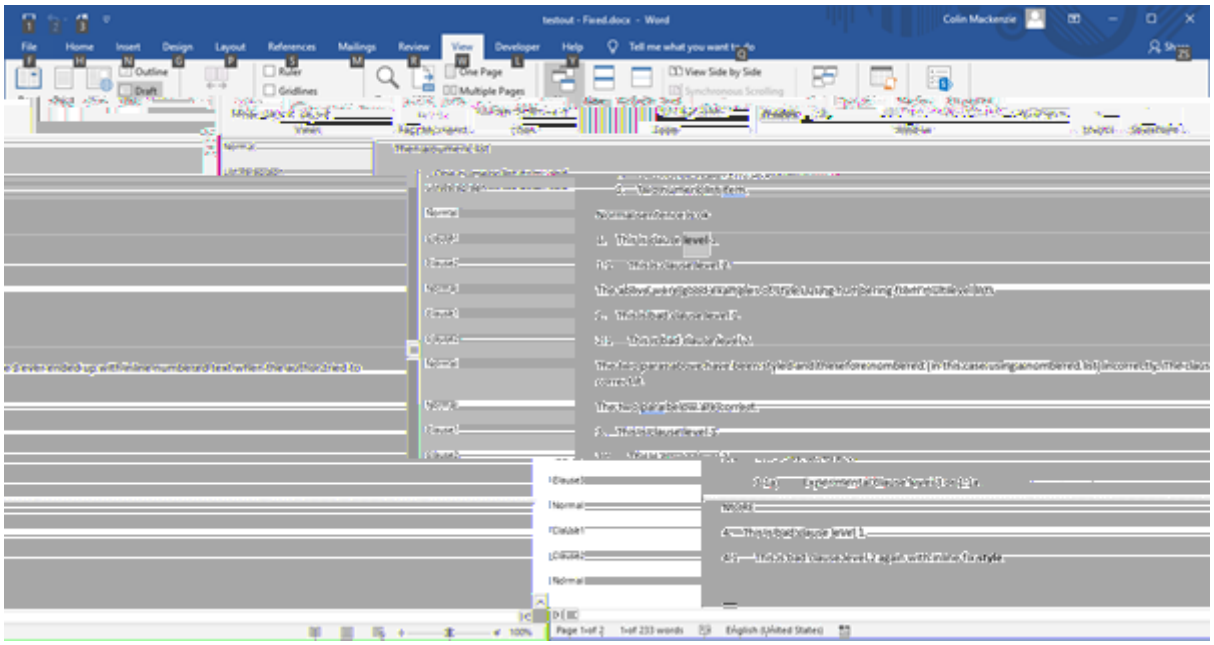


Figure 5. Screenshot of corrected document

5. Technical challenges and solutions

In this section, we will describe some of the approaches taken and difficulties encountered when developing this solution.

5.1. XProc

While XProc provided all of the functions necessary for the unpacking and repackaging of the documents there is an opportunity for the XML community to improve documentation and examples to the level of those available for other such technologies (e.g. there is no XProc book, documentation/examples for steps like the zip/unzip steps could be improved for updates of existing archives). Limitations of Xproc (that will be removed by XProc3) slowed development including

the need for variables to be the first thing in a group, the lack of Attribute Value Templates (AVT) to populate XML structures with XProc variables and especially the inability to have anything other than atomic values in variables (e.g. to store sequences of elements to iterate over or to pass to XSLT as parameters).

Debugging complex XProc can also be a frustrating process as run-time error messages from Calabash have no line numbers and, in many cases, (e.g. where the input to a step is required but for some reason is not present) there is no indication as to which of your custom steps is throwing the error.

Some tasks that seem easily achievable actually turn out to be a little more complicated. One of the features of the tool is to optionally run a series of XSLTs (that are named in the config file) before and after the Schematron processing step. While it is easy to iterate over the XSLT filename list it is not trivial to then pass the content through the dynamic pipeline (with the output of the previous XSLT being the input to the next XSLT). While this challenge has already been solved by and code provided via Open Source solution (see [2]), I wanted to define my own solution (as an intellectual challenge and to make sure I could easily change the functionality for my own use case). The answer, as is so often the case in XSLT, is recursion where a custom step is called with the list of XSLT filenames. The step will run the first XSLT in the list on the initial input XML and then call itself passing the output of the step along with the XSLT filename list minus the XSLT filename that has just been run.

5.2. XSLT

Despite the fact that XSLT3 has been a W3C recommendation since 2017, most of our customers have not yet adopted it (many client developers do not yet fully utilize the use of XSLT2 features such as functions). For my own project, I was therefore keen to utilize XSLT3 for the transformations in the solution to evidence it's benefits to my clients. XSLT3 provides powerful new features such as streaming, maps and support for non-XML sources and the following features were appropriate for this solution.

XSLT3 supports the `xsl:evaluate` element that can dynamically evaluate an XPath provided as a string. We used this capability for providing fixes via functions (see Section 5.3) and also to evaluate the location paths for issues identified by Schematron validation. These error location paths were dynamically evaluated in order to identify on which Word elements we need to insert comment references. Without the use of `xsl:evaluate`, previous XSLT2-based solutions have had to process the SVRL (the XML format describing the Schematron errors and their location) to create another XSLT (with matches for the defined location steps) to achieve the same result. As the XPaths being evaluated are in this case limited to those created by the SVRL, there is no danger of a security breach through an injection attack.

The use of {Attribute Value Templates} (AVT) has been expanded in XSLT3 to support Text Value Templates (TVT) which aids the creation of clean minimal stylesheet code. I did find whitespace handling limitations when TVT was used in a function that returned a string (this was not a problem if the function were to return an integer). This was encountered while trying to create a function that uses TVT to create strings as IDs. This can be illustrated simply when comparing the output of the following test functions.

```
<xsl:function name="ms:getInt" as="xs:integer">
        <xsl:param name="pElement" as="element()"/>
        {$pElement/position()}
    </xsl:function>
```

Returns "1".

```
<xsl:function name="ms:getString" as="xs:string">
        <xsl:param name="pElement" as="element()"/>
        {$pElement/position()}
    </xsl:function>
```

Returns "`
 1
 `" which includes the whitespace used to format the function that would not have been included had `xsl:value-of` been used instead of the TVT. This is understandable as the rules as to what is significant whitespace are complicated and the TVT example certainly includes no elements to help decide what is correct.

I also stumbled upon an obscure Saxon bug when experimenting with disable-output-escaping (DOE) where templates or the stylesheet had TVT turned on (using expand-text="true"). In these cases, the TVT worked but the DOE did not. Once reported, the issue was instantly diagnosed and kindly fixed by Michael Kay[2].

5.3. Dynamic calls to functions

As was briefly discussed in Section 4, we decided to implement the definition and application of fixes using a new approach rather than implementing the Quick-Fix[3] vocabulary. While this may change over time, we decided to investigate the suitability of applying fixes by calling user-defined functions dynamically using the same `xsl:evaluate` approach we took when injecting Schematron errors into the Word document from the SVRL output.

When a document that contains "Fix" comments is reprocessed by the solution, an XSLT (as configured in the config.xml for this particular Word template in order to support difference in rules/fixes per document type) is run on the combined Word content XML. The XSLT includes the main framework code that will

[2]See `https://saxonica.plan.io/issues/4412`
[3]See `http://www.schematron-quickfix.com/`

invoke the fix functions along with the corporate/document specific fixes (or included libraries of shared fixes). As the functions are invoked dynamically, there is no need for a corporate user to ever have to edit the main framework code to add calls to the function (as the function name and arguments are related to the content via the Word comment injected by the fix mark-up and the functions themselves are defined in the users own XSLT files). This will help avoid errors and ensure easier upgrades by separating the fix from the core code especially if the core templates are moved into an XSLT3 package to avoid them from being overridden.

An example of a fix function (and additional template matches invoked) to change a paragraph style would be as follows.

```
<xsl:function name="ms:ChangeStyle" as="element(w:p)"
visibility="public">
        <xsl:param name="pOriginalElement" as="element(w:p)"/>
        <xsl:param name="pThisPara" as="element(w:p)"/>
        <xsl:param name="pToStyleId" as="xs:string"/>

        <xsl:apply-templates select="$pThisPara" mode="ChangeStyle">
            <xsl:with-param name="pToStyleId" select="$pToStyleId"
tunnel="yes"/>
        </xsl:apply-templates>
    </xsl:function>

    <xsl:template match="w:p[not(w:pPr)]" mode="ChangeStyle">
        <xsl:param name="pToStyleId" as="xs:string" tunnel="yes"/>
        <xsl:copy>
            <xsl:copy-of select="@*"/>
            <w:pPr>
                <w:pStyle w:val="{$pToStyleId}"/>
                <w:rPr>
                    <w:lang w:val="en-GB"/>
                </w:rPr>
            </w:pPr>
            <xsl:copy-of select="* except (w:pStyle|w:ind)"/>
        </xsl:copy>
    </xsl:template>

    <xsl:template match="w:pPr" mode="ChangeStyle">
        <xsl:param name="pToStyleId" as="xs:string" tunnel="yes"/>
        <xsl:copy>
            <xsl:copy-of select="@*"/>
            <w:pStyle w:val="{$pToStyleId}"/>
            <!-- currently we always remove any locally applied indent --
>
            <xsl:copy-of select="* except (w:pStyle|w:ind)"/>
```

```
        </xsl:copy>
    </xsl:template>
```

A function to delete a paragraph would be much simpler (and even simpler without error checking).

```
<xsl:function name="ms:DeleteCurrentPara" as="element(w:p)?"
visibility="public">
        <xsl:param name="pOriginalElement" as="element(w:p)"/>
        <xsl:param name="pThisPara" as="element(w:p)"/>
        <xsl:choose>
            <!-- simply return nothing -->
            <xsl:when test="$pThisPara/self::w:p"/>
            <xsl:otherwise>
                <!-- must have been used wrongly so just keep as is -->
                <xsl:sequence select="$pThisPara"/>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:function>
```

Fixes can be applied to a para or a sequence of elements (e.g. "runs" of text).

While the use of these functions could solve the majority of common styling and content issues for corporate Word documents there is a limitation in that they can only remove/change the content items that are passed to the function (and/or insert new content before or after it). The functions cannot affect sibling content or any other part of the document. This limitation can be partially avoided by careful drafting of the Schematron rules to make sure the test is applied to the element that needs to be fixed rather than on another element.

Running code that is generated via comment text dynamically using `xsl:evaluate` could of course lead to run-time errors and attempts at injection attacks. The framework code protects the pipeline from errors in user functions (as much as is possible) by surrounding their invocation using `xsl:try` and `xsl:catch` and testing that a suitable function with the correct number of arguments can be found (using "function-available"). Injection attacks (potentially caused by malicious editing of the function information in the fix comment) are avoided as the code constructs the call to the user function carefully such that only functions in the required namespace are called and that the only arguments that are passed are strings (in addition to automatically generated standard arguments of the original context item(s), and the current items(s) to be processed).

Note

If the function call being generated references a named variable or parameter (e.g. "pOriginalElement" in `ms:ChangeStyle($pOriginalElement, ., 'Heading1')` then the `xsl:evaluate` will error unless the parameter is additionally passed using `xsl:with-param`.

```
<xsl:evaluate xpath="$vFunctionToRun" context-item="$pOutput[1]">
    <xsl:with-param name="pOriginalElement"
select="$pOriginalElement" as="element()" />
</xsl:evaluate>
```

5.4. OOXML

The greatest challenges developing this solution were caused by the complexity of the OOXML format. To add a comment requires not just the creation of the reference in the document XML but also the creation of other referenced elements in multiple other files. If the original source file had no comments then not only do the supporting files have to be created but the package "rels" file also needs to be updated to point to these new documents. Any bug in the core framework code creating the word content can swiftly lead to a resulting document that cannot be opened in Word or that could only be opened in repair mode (but with little detailed feedback as to what the issue actually is).

Ideally, we would have created the error, warning and fix comments where the only text was intended to be read by the user. Ultimately, we also had to include the Schematron rule ID and the name/arguments of fixes. This was necessary only because we could find no way of smuggling custom XML into the OOXML in a way that Word would then successfully open the document and keep the extra information. Processing instructions were dropped by Word. While the OOXML specification supports the `w:customXml` element, Word itself no longer supports this element (following a court case in 2009[4]).

6. Conclusion

While it is perfectly possible to achieve many of the same results in C# or VB .Net (especially using the Open XML SDK) a standards-based solution can be deployed more flexibly anywhere from a local machine to a cloud-based service. Further, developing using open standards inspires us to think of new standard-based approaches that may not have been considered by desktop developers that can deliver real business benefits. As the document quality directly depends on input from the author it would be wise to consider linking solutions back to the GUI to provide a more interactive experience while leaving the business rules declared in XML, XPath and Schematron and not "spaghetti code" embedded in templates. However, it should also be noted that most "Word template experts" within corporate environment will typically be some combination of Word "super-users" or macro developers and will not be familiar with XML (including OOXML), validation and Schematron. This would mean that for the solution to be

[4]See `https:// www.zdnet.com/ article/ microsoft-loses-its-appeal-in-200-million-plus-custom-xml-patent-infringement-case/`

a success an extensive library of Schematron tests and fix helper functions covering most common scenarios would have to be developed then made available along with some basic training.

By processing the native format (OOXML), the developer has full access to all of the data rather than a subset of data provided by APIs therefore opening up the opportunity for powerful applications, however with this power comes greater complexity.

While the development of the particular solution discussed in this paper would certainly have been easier using XProc3, this solution shows that it is possible to deliver powerful functionality in an easily extendable manner to process MS Office documents using current open standard technology.

Bibliography

[1] Andrew Sales: *The application of Schematron schemas to word-processing documents*, 2015 `https://xmllondon.com/2015/presentations/sales`

[2] Nic Gibson: *XProc Tools*. `https://github.com/Corbas/xproc-tools`

XML-MutaTe

A declarative approach to XML Mutation and Test Management

Renzo Kottmann
KoSIT
<renzo.kottmann@finanzen.bremen.de>

Fabian Büttner
KoSIT
<fabian.büttner@finanzen.bremen.de>

Abstract

Correctness of XML language designs is important in XML based data standardisation efforts. A general approach to testing of XML Schema and Schematron designs is to write own test frameworks including a set of XML instances to validate against the XML schema languages during development.

We present a new integrated test approach. It combines three concepts with a simple declarative language for annotating XML test instances. Mutation is the first concept for automatically generating many new test instances from a single original instance. The second concept of validation with expectation compares each positive or negative validation result with an expectation of a test writer. The last concept adds test metadata to XML test instances without interfering with XML schema language design and XML parsing. We also present XML-MutaTe as a prototype implementation that supports generation, execution and reporting of positive and negative test cases.

Overall, this approach and first implementation has the potential to prevent the need for custom tailored XML testing frameworks. Therefore, It simplifies test driven development of XML schema language designs for XML based data standard development.

Keywords: XML, testing, schema, test, management, schematron, generation

1. Introduction

Several aspects have to be taken into account for successfully testing XML schema language designs expressed e.g. in XML Schema Definition Language (XML

Schema) [5] or Schematron[6]. From a test management perspective these aspects are:

1. Generate test cases,
2. execute tests, and
3. summarize and report the outcome.

Currently, general practice is to implement custom test suits and frameworks (like e.g. the test framework for XML Schema testing [4]) where often one test case is equal to one XML instance. These frameworks often include custom scripts to manage the test cases, chain up validators, and generate custom test reports. A common variant is to additionally develop a custom XML language for defining a domain specific language (DSL) for testing. The aims of these custom languages include handling test metadata, provide test hints, configurations and commands. Therefore, tests are either written as stand alone documents separated from the XML instances under test or the XML instances are embedded in a custom test language. This either has the disadvantage that the test specification is separated from what is tested or that the XML instance is validated against a custom test language. Moreover, current test frameworks are tailored for either XML Schema or Schematron development but not for both. However, there are XML based data standardisation efforts which use XML Schema to define general structure and Schematron for expressing business rules.

In addition, a general shortcoming of many custom test frameworks is that tests cases are mostly written for positive testing i.e. an XML instance is validated against a schema language and expected to give a positive result. However, in order to also make sure that a schema language excludes wrong data, it would be of advantage to be able to write test cases for negative testing to make sure that wrong or missing data is always detected.

We present a new integrated test approach with a simple declarative language for annotating XML test instances with test metadata and instructions for automatic generating new test instances and validating these against test outcome expectations. We also present XML-MutaTe as a prototype implementation that supports generation, execution and reporting of positive and negative test cases.

2. Integrated Test Approach

Test management is defined as part of a software testing process that includes planning and generation of tests, their execution and storage and analysis of the tests results. The integration into a single approach requires three combined concepts.

2.1. Test Generation by Mutation

Automated test data generation is useful in order to minimize the effort to hand write XML test instances. There are several tools available to generate XML

instance documents based on a given XML Schema. These tools are very good in generating random documents within the constraints of the XML Schema i.e. generating valid instances. On the other hand, the generated content is mostly not very meaningful and does not necessarily reflect real world business requirements and business cases. Additionally, it is often important to test schema definitions against invalid instances. Both, automatic generation and manual generation of test instances are useful during initial development of XML Schema definition languages as well as for further maintenance and enhancements.

Therefore, the concept of test generation by mutation allows generating new test instances by applying changes i.e. evolving original XML test instance to a new state. Each such a new state is named mutant and can either be a valid or invalid XML instance.

The agents of defined mutation strategies are called mutators. There can be many defined mutation strategies some of which can be classified as simple and the others as arbitrary complex. Simple mutations are defined as a single syntactic change to an attribute or element whatever the complexity of the element is. This can also be referred to as atomic changes [3]. Some mutators generate many mutations with a single atomic change per generated XML test instance.

There are several simple mutators:

Table 1. Mutators

Name	# **Mutants**	**Description**
empty	1	Deletes the text content of an element or attribute.
add	1	Adds an element or attribute.
remove	1	Removes an element or attribute.
rename	1	Renames an element or attribute.
change-text	m	Changes text content of an element or attribute.
whitespace	m	Exchanges text content of an element or attribute with random whitespace content.
identity	1	Keeps element as is.
code	m	Exchanges text content of an element with a list of code words one by one.
alternative	m	Uncomments each comment one by one. Allows to e.g. test XML Schema choices [8].
random-element-order	m	Randomizes child element order of an element.

More complex mutators can be based on execution of XSLT [2] which can perform many syntactic changes at once for example.

2.2. Validation with Expectations

The usual result of a validation is either true or false which is equal to valid or invalid. However, one needs to be able to examine if a validation result is really matching a certain business requirement. Hence, each test needs to be able to answer the question: "Is the outcome of the validation as expected by the business requirement?". The term expectation is used to differentiate this concept from - and not to confuse it with - the more commonly used term "assertions" from e.g. Schematron.

An expectation can itself be expressed as true (expectation is met) or false (expectation is not met). Therefore, there a four possible test results for each single constraint or rule w.r.t. the content of an XML test instance. According to a business rule each validation procedure with an expectation **does**

- accept valid content as **True Positive (TP)**
- exclude invalid content as **True Negative (TN)**
  and **does not**
- accept invalid content as **False Positive (FP)**,
- exclude valid content **False Negative (FN)**

Table 2. Validation of expectation truth table

Validation Result/Expectation	valid	invalid
valid	+ (TP)	- (FP)
invalid	- (FN)	+ (TN)

Validation result (column) versus expectation (row)

2.3. Declarative Annotation

A declarative annotation approach with XML processing instructions allows test writers to generate a few original valid test instances which are designed by the basic question "Does the XML Schema express everything for my business need?" and annotate these with specific mutation instructions and expectations. A test writer uses a mutation instruction to declare a certain mutation strategy which should be applied to an original instance in order to generate new test instances as variations of the original instance on the fly. Moreover, a test writer can also

declare expectations about the validity of the mutated instances. And finally a test writer can add metadata about the test case at hand.

3. Simple Mutation and Testing Language

The simple mutation and testing language for the declarative annotation of XML test instances is designed as a simple list of configuration items within XML processing instructions. Because processing instructions are in effect external to the main structure of an XML document, they have no impact on the XML schema languages. Processing instructions are also ignored by all XML processors by default, except by specialized applications interpreting these.

3.1. xmute Processing Instructions

Only XML processing instructions with name `xmute` are processed and interpreted.

The general data structure of an instruction is a list of configuration items with the following key/value structure:`key="value"` as shown in this example:

```
<?xmute mutator="empty" schema-valid schematron-invalid="bt-br-03" ?>
<element>with text content</element>
```

All item keys are interpreted case-insensitive. Each item value must be surrounded by quotes ". Sometimes the `value` to a key is optional and can be omitted. This is demonstrated by the `schema-valid` key in the above example. By default an `xmute` instruction refers to the next sibling XML element.

3.2. Mutations

One and only one `mutator` key(word) is mandatory where value is the name of the mutator to be applied e.g. `mutator="empty"`.

There might be additional `key=value` items configuring the behavior of the mutator.

3.3. Test Expectations

Each mutant (i.e. mutated document) can be validated against XML Schema and Schematron rules and compared to the expectations of the test writer.

3.3.1. Expectations on XML Schema

`schema-valid` and `schema-invalid` items declare expectations about the outcome of an XML Schema validation on a mutant.

This allows to generate various tests about what an XML Schema should achieve.

Example 1: Test optional elements

One possibility to test that an XML Schema correctly allows an element to be optional is to remove an optional element from a valid XML test instance. Hence we create a schema valid document with an optional element:

```
<element>element with content</element>
```

Then we can create a test case by annotating the XML test instance with an `xmute` processing instruction using the `remove` mutator and declare that the resulting mutant has to be schema valid:

```
<?xmute mutator="remove" schema-valid ?>
<element>optional element with content</element>
```

In case the XML Schema validation result is true, it will meet the expectation. Hence, the test result will be positive, otherwise negative.

Example 2: Test required elements

In order to test that an XML Schema correctly requires an element to be always present, we can again use the `remove` mutator and declare that our expectation is that the XML Schema validation result will be invalid :

```
<?xmute mutator="remove" schema-invalid ?>
<element>required element with content</element>
```

In case the XML Schema definition still treats the element as optional, the validation result will be true, but it will *not* meet the expectation. Hence the test result will be negative. Only after changing the XML Schema definition and force the element to be required, the XML Schema validation result will be false, the expectation will be met, and the test result will be positive.

3.3.2. Expectations on Schematron Rules

The configuration items `schematron-valid="some-rule-id"` and `schematron-invalid="some-rule-id"` declare expectations about the outcome of Schematron validations. The required value can be a list of space separated schematron rule identifiers and an optional schematron symbolic name. In case one or more rule-ids are listed, the expectations of only these rules will be evaluated. In case other rules fire, they will not be reported by default.

Example 3: One simple rule

A single Schematron rule `rule-1` requires that an `element` has to have text content (independent of the above question if the `element` is optional or required by an XML Schema) i.e. it will fire a fatal if `element` is empty. We can declare another test case based on the previous example in the same document as follows:

Simple Example:

```
<?xmute mutator="empty" schema-valid schematron-invalid="rule-1" ?>
<element>element with content</element>
```

The `empty` mutator will generate a mutant similar to this one:

```
<element></element>
```

It will meet the XML Schema expectation. But only if the Schematron `rule-1` correctly fires a fatal message, it will also meet the `schematron-invalid`expectation and the whole test case will be positive. Otherwise the test case will be negative.

Example 4: Many complex rules

More than one rule can be declared:

```
<?xmute mutator="empty" schema-valid
   schematron-invalid="rule-1 rule-2 rule-3" ?>
<element>element with content</element>
```

In case a test case needs to validate against different schematron files, symbolic names can be assigned to schematron rules:

```
<?xmute mutator="empty"
   schema-valid
   schematron-invalid="ubl:rule-1, ubl:rule-2, xr:rule-1"
?>
<element>with content</element>
```

Here, there are two rules from Schematron with symbolic name `ubl` and one more rule with symbolic name `xr`. These symbolic names have to be defined as input to an xml-mutate processor.

There are two special keywords for convenience: `none` and `all` the meaning is defined as follows:

- `schematron-valid="all"`
  - All rules are expected to be valid
- `schematron-valid="none"`
  - None of the rules are expected to be valid
- `schematron-invalid="all"`
  - All rules are expected to be invalid
- `schematron-invalid="none"`
  - None of the rules are expected to be invalid i.e. `schematron-valid="all"`

3.4. Test Metadata

Three additional configuration items are defined to facilitate creation of test reports with meatdata content.

The `id` configuration item identifies the test case and `description` allows to document the purpose of the test case.

The function of the `tags` item is to allow arbitrary grouping of test cases and selective execution of only certain test cases. A list of identifying keywords is allowed.

Example 1. Metadata annotation

```
<?xmute mutator="remove"
    schema-valid
    id="test-id"
    tags="mandatory simple"
    description="A description of the test case purpose."
?>
```

4. XML-MutaTe Prototype

A functional prototype implementation exists and is called *xml-mutate* for XML **Muta**ting and **Te**sting. It is written in Java and has a command line interface and writes a mutation and test report to console. The source code is available on GitHub[1].

The current version already makes test driven XML Schema and Schematron development possible. This can be demonstrated by a real example which implements fictive business requirements on an XML design for book data. Assume the development starts with two simple requirements:

1. A book must always have a publisher, and
2. A book must always have a number of chapters.

Then a concrete XML test instance can be written as follows:

```
<book isbn="1-861002-85-8">
    <title>Professional Java XML Programming</title>
    <?xmute mutator="remove" schematron-valid="publisher-exist"
        id="publisher-exist-test"
        description="Demonstrate that an Expectation is not met."
    ?>
    <publisher>Wrox Press</publisher>
    <price>35.99</price>
    <pages>772</pages>
    <?xmute mutator="remove" schematron-invalid="chapters-exist"
        id="chapters-exist-test" tag="mandatory,simple"
        description="All Expectations are met if chapters
                     is removed then rule will detect it."
    ?>
    <chapters>13</chapters>
</book>
```

It includes two test cases written according to the Simple Mutation and Testing Language. The first test case with `id` *publisher-exist-test* tests the first business requirement. It declares a mutation where the element `publisher` is removed. It

[1] https://github.com/itplr-kosit/xml-mutate

also expects that the validation result of the schematron rule *publisher-exist* will be valid. This test case is designed for the purpose of demonstration only. It showcases the outcome if an expectation is not met and therefore raising the question if a business requirement is correctly implemented. The second test case `id` *chapters-exist-test* tests the second business requirement. It declares a mutation where the element `chapters` is removed. It also expects that the validation result of the schematron rule *chapters-exist* will be invalid. Therefore, it proofs the correctness of the schematron rule which has to fire a fatal message if the element `chapters` does not exist.

Development of Schematron rules can start with this simple `book.sch`:

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron"
queryBinding="xslt2">
    <pattern id="model">
        <rule context="//book">
            <assert test="publisher" role="fatal" id="publisher-exist">
                A book must always have a publisher.</assert>
            <assert test="chapters" role="fatal" id="chapters-exist">
                A book must always have a number of chapters.</assert>
        </rule>
    </pattern>
</schema>
```

This Schematron implementation tests for each `book` element that a `publisher` and a `chapter` element is present.

An execution of xml-mutate as follows:

```
java -jar xml-mutate.jar \
     --schema book.xsd \
     --schematron book.sch \
     --target /tmp/ book-simple.xml
```

requires a `book.sch` and `book.xsd` as parameters. `xml-mutate` takes `book-simple.xml` as input and processes all `xmute` instructions. It persists all generated mutations in `/tmp` directory and generates the following report as console output:

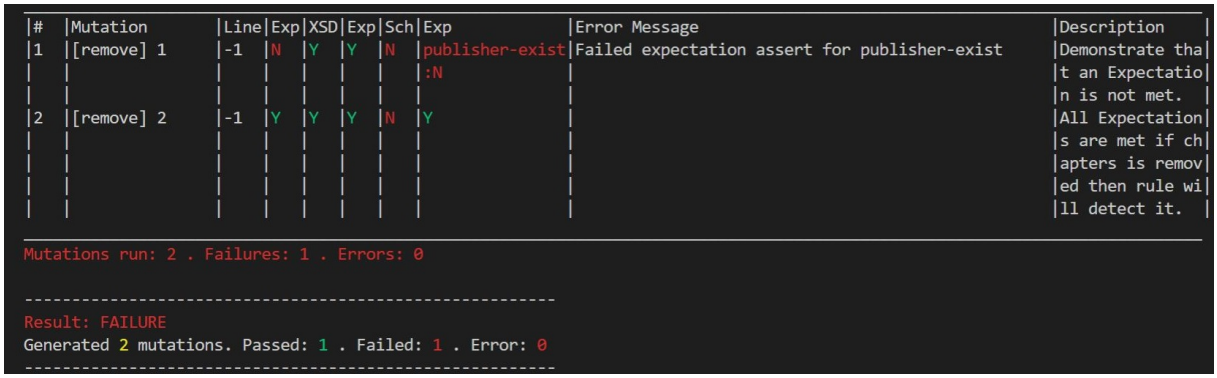


Figure 1. XML-MutaTe console output

As can be seen on the bottom line. Overall `xml-mutate` generated two more XML test instances as mutations from the original XML test instance. One validation of expectation passed and another one failed. Both mutations meet the expectation that they validate against the XML Schema (column 4 and 5). The validation result of the first mutation (`[remove] 1`) did not meet the schematron expectation. The schematron rule *publisher-exist* fired a fatal message (**N** for not valid in column 7), but the expectation was that no message is fired (**N** in column 8). The second schematron rule passed because all expectations are met. The schematron rule *publisher-exist* fired a fatal message (**N** for not valid in column 7) and the expectation was that a message is fired (**Y** in column 8).

The current status of implemented mutators is summarized in the following table:

Table 3. Mutators implemented by XML-Mutate

Name	**Implementation status**
empty	available
add	in planning
remove	available
rename	in planing
change-text	available
whitespace	available
identity	available
code	available
alternative	available
random-element-order	in planning

5. Use Case: XRechnung Standard

The main incentive for developing XML-MutaTe originates from the XML based data standard XRechnung [10]. Here, all requirements on an invoice are specified in a national specification based on- and compliant to the European Norm EN16931[11]. The European Norm did not invent an own XML Schema. It allows the use of already existing XML Schemas for invoices such as the Universal Business Language (UBL) [13]. This XML Schema defines the data structure of invoices, but not many rules on the specific content requirements. Therefore, the EN16931 is accompanied by a set of Schematron rules implementing requirements on invoices for the European market[12]. In addition XRechnung is accompanied by an additional set of Schematron rules implementing national requirements on invoices for the German market.

Technically, an invoice is an XRechnung only if it validates against all three Schemas in that order:

1. XML Schema (e.g. UBL invoice)
2. EN19931 Schematron rules
3. XRechnung Schematron rules

Altogether, there are hundreds of Schematron rules. From a test management perspective it requires to have many tests and many kinds of tests. On the smallest test scope level it requires unit tests to make sure that e.g. each XRechnung Schematron rule does what it is expected to achieve. Because of the more complex validation setting, it also needs regression tests. These make sure that if something is changed in the XML Schema or EN19931 Schematron rules all requirements are still met and no unforeseen side effect breaks other existing rules. And finally, it requires integration tests to make sure that any two rules do not contradict each other.

The value of this approach can already be demonstrated with the use of the simple `empty` mutator:

The deprecated XRechnung standard version 1.1[2] stated on p. 49f. that Business Group (=Gruppe) "Seller Contact" should exist and have `Seller contact point BT-41`, `Seller contact telephone number BT-42`, and `Seller contact email address BT-43`. This is further expressed on p.65 with “BR-DE-5 Das Element „Seller contact point“ (BT-41) muss übermittelt werden.”, “ BR-DE-6 Das Element „Seller contact telephone number“ (BT-42) muss übermittelt werden.”, and “BR-DE-7 Das Element „Seller contact email address“ (BT-43) muss übermittelt werden.”

This is expressed by the following Schematron rules on an UBL Invoice (excerpt):

[2] https://www.xoev.de/die_standards/xrechnung/xrechnung_versionen/xrechnung_version_1_1-15369

```
<param name="BG-6_SELLER_CONTACT"
   value="//ubl:Invoice/cac:AccountingSupplierParty/cac:Party/cac:Contact"/>

<param name="BR-DE-5" value="cbc:Name"/>
<param name="BR-DE-6" value="cbc:Telephone"/>
<param name="BR-DE-7" value="cbc:ElectronicMail"/>
```

Obviously, the Schematron rules only require the element to be present even if it has no content.

Now, we can use a positive example and use xml-mutate to check this issue. We take a valid UBL Invoice and annotate it with the following declarations:

```
<cac:Contact>
  <?xmute mutator="empty" schema-valid schematron-invalid="BR-DE-5" ?>
  <cbc:Name>[Seller contact person]</cbc:Name>
  <?xmute mutator="empty" schema-valid schematron-invalid="BR-DE-6" ?>
  <cbc:Telephone>+49 123456789</cbc:Telephone>
  <?xmute mutator="empty" schema-valid schematron-invalid="BR-DE-7" ?>
  <cbc:ElectronicMail>test@test.de</cbc:ElectronicMail>
</cac:Contact>
```

XML-Mutate takes each declaration and mutates the document where the content of the next element is made empty. Additionally, it expects to validate against UBL XML Schema (keyword `schema-valid`) but it expects that the XRechnung Schematron does not validate against specific rules (e.g. `schematron-invalid="BR-DE-5"`).

Therefore, the above xmute instructions test the business requirement that all these elements should also have content.

The deprecated version of the above XRechnung Schematron rules did not satisfy this business requirement which is technically expressed with declaration of `mutator="empty"` in combination with `schematron-invalid` expectation. Therefore, these three simple test cases discovered three bugs in the technical implementation of the business requirement. This was corrected in newer versions of the XRechnung standard and the XRechnung Schematron rules.

Additionally, dozens of more such tests are declared in a single valid XRechnung test instance.

6. Discussion and Conclusion

The general advantage of this approach is that it prevents the need for developing custom test frameworks. It only requires one tool, XML schema languages (XML Schema or Schematron), and XML test instances annotated with rich test instructions using processing instructions otherwise ignored by any XML processing tool.

Already now, the simple mutation and testing language seems to be feature complete and allows declaring all what is needed for an integrated test approach. Moreover, it is simple to add new features to this simple key/value based lan-

guage. With a clear separation of the simple mutation and testing language from the implementation, it is possible to implement alternative processors with different feature sets and based on different programming languages and technologies. Hence, making it possible that this approach becomes more widely accepted and adopted.

Using only XML-Mutate makes it possible to minimize the number of test instances while maximizing test coverage including negative tests. Already now, it is possible to use XML-Mutate for unit-, acceptance- and regression testing. Because all declarations are directly in the XML instances it allows test writers looking at the data to declare that XML Schema and Schematron have to validate according to the data at hand. Hence, this approach could be classified as a data-driven-development framework. This is in contrast to a unit test and behaviour driven development (BDD) framework such as XSpec for XSLT, XQuery and Schematron[9]. XSpec has a code-centric perspective. That's why both approaches complement each other.

Overall, this integrated approach and first implementation has the potential to prevent the need for custom tailored XML testing frameworks and simplifies test driven development of XML schema language designs for XML based data standards.

7. Outlook

The mutation and test approach is in its invention phase. Conceptually, it is possible to integrate computation and reporting of test coverage to better measure indicators for estimating test quality and indirectly design quality.

On the implementation level, many features are on the road map. These include Relax NG validation [1], customizable XML test instance file naming and several more mutators such as a random element order generator.

Currently, the reporting capability is limited to a simple console output. In order to allow rich reporting capabilities, Extensible Validation Report Language (XVRL) [7] is under examination to be used as the standard report data format for XML-MutaTe. This would clearly separate reporting data from presentation in many different formats (HTML, PDF etc.) and allow developers to add own reporting capabilities for individual requirements.

Bibliography

[1] Clark, James – Cowan, John – MURATA, Makoto: *RELAX NG Compact Syntax Tutorial*. Working Draft, 26 March 2003. OASIS. `http://relaxng.org/compact-tutorial-20030326.html`

[2] Kay, Michael: *XSLT 2.0 and XPath 2.0*. Wiley Publishing, 2008.

[3] *Standard Change Tracking for XML* https://www.balisage.net/Proceedings/vol13/html/LaFontaine01/BalisageVol13-LaFontaine01.html

[4] *XML Schema Test Suite* https://www.w3.org/XML/2004/xml-schema-test-suite/index.html

[5] Gao, Shudi– Sperberg-McQueen, C.M. – Thompson, Henry S.: *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. W3C Recommendation, 5 April 2012. https://www.w3.org/TR/xmlschema11-1/

[6] Jelliffe, Rick. *Schematron*, 1999. Retrieved from http://xml.ascc.net/schematron

[7] *Extensible Validation Report Language*. Retrieved from https://github.com/xproc/xvrl

[8] *XSD Choice*. Retrieved from https://www.w3.org/TR/xmlschema11-1/#element-choice

[9] *XSpec*. Retrieved from https://github.com/xspec/xspec

[10] *XRechnung*. Retrieved from https://www.xoev.de/de/xrechnung

[11] *Electronic invoicing - Part 1: Semantic data model of the core elements of an electronic invoice; German version EN 16931-1:2017*. Retreived from https://www.din.de/de/mitwirken/normenausschuesse/nia/normen/wdc-beuth:din21:274990963

[12] *Validation artefacts for the European eInvoicing standard EN 16931*. Retreived from https://github.com/ConnectingEurope/eInvoicing-EN16931

[13] *Universal Business Language Version 2.1.* 04 November 2013. OASIS Standard. http://docs.oasis-open.org/ubl/os-UBL-2.1/UBL-2.1.html.

Analytical XSLT

An Analytical Approach to Writing XSLT Transformations for Converting Documents Between DTD Versions

Liam Quin
<liam@fromoldbooks.org>

1. Abstract

People working with large XML vocabularies occasionally face the task of upgrading to a new version of a vocabulary. A similar situation arises when documents must be exchanged with an organization using a different version of a vocabulary. This paper describes an effective computer-aided approach to writing transformations in XSLT to convert documents to conform to a slightly different version of a DTD; similar techniques apply for arbitrary schema languages with caveats noted in the text. A tool to assist in this process is also described.

2. Introduction

Writing an XSLT transformation to process documents written in a large XML vocabulary can be a daunting task. Every element in the input must be handled, along with all of its attributes. When the task is to convert from one version of a vocabulary to another, one must also examine the destination vocabulary. For vocabularies represented by XML document type definitions this means comparing two DTDs.

An obvious approach to people with a background in programming is to automate as much as possible the tedious task of comparing element declarations to see what changed between two versions of a DTD. Since DTDs are stored in files one might try a text comparison utility such as Unix **diff**, but this turns out to give misleading results since it is not aware of file structure: not only inclusions, but, more importantly, conditional sections.

A program that uses an XML parser to read the two DTDs and then compare the resulting data structures is fairly easy to write, and has been done several times in the past [see references]. Although the existing DTD comparison tools are not without problems, at least some of them are open source and could be patched (or forked, if necessary). But it turns out that this is a journey in an inappropriate direction.

An analyst looking at this problem wants higher-level tools to help with the task. The author of this paper wrote Eddie 2 in order to approach this sort of problem, and has now used this tool for a number of projects. But what matters here is not the specific tool so much as the approach, and why recording the differences between two grammars, although necessary, is not sufficient.

This paper first reviews some of the existing DTD comparison tools, then briefly describes Eddie 2 to give the reader necessary context. We will then be ready to discuss the difference in approach: rather than using a DTD comparison to form the basis of an XSLT stylesheet, the analytical approach is to write a new stylesheet *informed* by a tool that detects not only differences but *incompatibilities*. The generated XSLT stylesheet is not edited itself, but is used as a tool for analysis.

It is the contention of the author that this methodology is effective; that is not to say that it could not be improved, and sharing this methodology more widely and inviting feedback is a step in that direction.

3. Existing Tools For Comparing DTDs

When a DTD is all contained in one file, and does not use conditional sections, inclusions, or extended comments, a text-base file difference utility goes a long way. On the other hand, if your single DTD file is a "flattened" copy of DocBook or BITS, the result may take you hours or days to process by hand. Putting each declaration on one line and sorting the results before using diff might help, especially if you then sort the elements in repeatable or-groups within content models. But you are working at the textual level and not thinking about the actual problem, which can be a distracting impediment.

Specialized tools for comparing DTDs and schemas exist. These may view the DTDs as grammars and list differences, or may be more text-based. A drawback of textual comparisons is that the large and complex DTDs for which tools are most useful tend to make heavy use of parameter entities and conditional processing, so that two elements might have the same textual content model but because of differing parameter entity expansions actually be very different.

3.1. DTD Diff

Early on, there was a tool for comparing two SGML DTDs written by Earl Hood [ref1]; this used a regular-expression-based parser written in Perl by Norm Walsh, DTDParse. This was updated to parse XML, although it does so from an SGML perspective, potentially leading to subtle bugs. The version the author of this paper tested did not work correctly on the JATS DTDs, but, to be fair, neither does the author's own tool, Eddie 2, described later in this paper. It provides a very simple line-oriented output; it was easier to write a new SAX-based application than to parse the output of DTD Diff.

3.2. DTD Comparator and DtdAnalyzer

The American National Center for Biotechnology Information/NLM/NIH (NCBI) offers a set of Java-based tools that use a native XML parser to produce an XML

representation of a DTD and that can produce a summary of differences, including sample XSLT.

Although there seem to be some bugs (and no work of humankind is without flaw), these could be fixed—an obvious one is that DTD Comparator does not sort the elements in or-groups, so that if all that has changed is the order, spurious differences appear to be generated. However, DTD Comparator was the most promising of the tools the author surveyed. The author of this paper does make use of the DTD Flatten utility in this package, as Eddie 2 works correctly when given a flattened DTD (that is, a DTD with parameter entities expanded).

DTD Comparator can create an XSLT stylesheet with the intent that you use it as a starting point for editing. Although this sounds useful, it has the drawback that you can't rerun the program after a small change to a DTD, once you have edited the stylesheet. Like Eddie 2, DTD Comparator produces an HTML report, but it is not designed for continuous use as part of a methodology. It is this fundamental underlying difference that led the author of this paper not to pursue contributing to the DTD Comparator project.

3.3. Style Studio

The Stylus Studio XML editor includes a visual mode for mapping from one XML schema to another. The result of this is Java code which, when executed, will perform the appropriate transformation when run in the company's proprietary database environment. This does not seem to be a productive way to write XSLT, although the visual comparison might be useful for those who can decipher it.

4. Eddie 2

The author had specific needs that none of the existing tools seem to meet:

- Generate XSLT with a template for each element containing, in comments, the differences between the two DTDs;
- Identify cases where an instance of an element conforming to the input DTD would not, if presented *in situ* in the output DTD, with no other change except possibly to its namespace URI, be valid;
- Help an XSLT developer to identify the most common problem areas and address them quickly;
- Generate and maintain a list of elements that the stylesheet author has not yet handled.

Although the DTD difference tools mentioned in the previous section could be part of this, they are not the whole solution. Their focus is on identifying differences at a moment in time, not on a process of developing a transformation.

The author wrote a new program to meet these needs, or at least to explore how to meet them: Eddie 2; this does not preclude merging with another tool in the future, but Eddie 2 was working well enough to use after a few hours. The main difficulty is always with XML Catalog files! Eddie 2 has since been developed further, when it became clear to the author that the approach was viable.

What follows is a brief overview of Eddie 2 as it currently stands; after that we can discuss how the tool supports an analysis-based methodology.

4.1. Eddie 2 Overview

The Eddie 2 program reads a configuration file (and also command-line options) uses an XML parser to construct a simple stub document with given public and system identifiers and to use this to load a DTD for each of input and output vocabularies. It then generates:

- An HTML report, with CSS and JavaScript to make it usable (for example, you can type any letter to scroll directly to the first element starting with that letter, and HTML content models are "pretty-printed" with parenthesis matching);
- An XSLT stylesheet with a template for each element; the template by default copies the element to its output (optionally discarding namespace nodes), and includes comments that show the respective content models and attribute declarations and that highlight likely incompatibilities.

4.1.1. The Generated XSLT Stylesheet

Eddie 2 writes an XSLT stylesheet that declares namespace bindings declared in its configuration file and then contains a template for every element in the source DTD.

For each element, the default behaviour is to create a template that will produce a message if a potential incompatibility with the destination DTD is detected. For example:

- an element in content that does not occur at all in the target DTD;
- an element that occurs in the target DTD but is not allowed as a child of this element;
- an attribute that is not allowed on this element in the target DTD;
- an attribute that has a value that is not allowed on this element in the target DTD—for example an unknown value from an enumeration, or a CDATA-valued attribute that is not equal to a #FIXED value.

Figure Figure Fig 1 shows an example template generated by Eddie 2 for a transformation between two different DTDs each based on a different version of JATS.

```
<xsl:template match="role">
  <!--* Notes from Eddie2
      * children of element role differ:
      *   in src not dest: index-term, index-term-range-end. inline-media
      *
      * role: Or-groups with different children
      *   src: (#PCDATA|email|ext-link|uri|inline-supplementary-material|
      *   related-article|related-object|hr|bold|fixed-case|italic|
monospace|
      *   overline|overline-start|overline-end|roman|sans-serif|sc|
strike|
      *   underline|underline-start|underline-end|ruby|alternatives|
      *   inline-graphic|inline-media|private-char|chem-struct|inline-
formula|
      *   tex-math|mml:math|abbrev|index-term|index-term-range-end|
      *   milestone-end|milestone-start|named-content|styled-content|fn|
target|
      *   xref|sub|sup|x)*
      *   dst: (#PCDATA|email|ext-link|uri|inline-supplementary-material|
      *   related-article|related-object|hr|bold|fixed-case|italic|
monospace|
      *   overline|overline-start|overline-end|roman|sans-serif|sc|
strike|
      *   underline|underline-start|underline-end|ruby|alternatives|
      *   inline-graphic|private-char|chem-struct|inline-formula|tex-
math|
      *   mml:math|abbrev|milestone-end|milestone-start|named-content|
      *   styled-content|fn|target|xref|sub|sup|x)*
      *
      * Attributes in source but not destination:
      *   degree-contribution CDATA #IMPLIED
      *   vocab CDATA #IMPLIED
      *   vocab-identifier CDATA #IMPLIED
      *   vocab-term CDATA #IMPLIED
      *   vocab-term-identifier CDATA #IMPLIED
      *
      * Destination attributes:
      *   content-type CDATA #IMPLIED
      *   id ID #IMPLIED
      *   specific-use CDATA #IMPLIED
      *   xml:base CDATA #IMPLIED
      *   xml:lang NMTOKEN #IMPLIED
      *-->
  <xsl:copy>
    <xsl:apply-templates select="@*" />
```

```
    <xsl:if test="index-term">
      <xsl:message>role: role contains child index-term
                           not in destination DTD</xsl:message>
    </xsl:if>
    <xsl:if test="index-term-range-end">
      <xsl:message>role: role contains child index-term-range-end
                           not in destination DTD</xsl:message>
    </xsl:if>
    <xsl:if test="@degree-contribution">
      <xsl:message>role: element role has attribute @degree-contribution
                           not in destination DTD</xsl:message>
    </xsl:if>
    [. . . more tests omitted for publication. . .]
    <xsl:apply-templates select="node()"/>
  </xsl:copy>
</xsl:template>
```

Figure Fig 1. Fragment of an XSLT stylesheet produced by Eddie 2

It's possible to supply condition-message pairs for a given element by editing the configuration file; by default Eddie 2 generates the messages shown in the figure. It is also possible to change the default action from using xsl:copy to using xsl:element, in order to avoid copying namespace nodes. It is not currently possible to customize the template further, however: the intent is that if you want to edit it, you mark it as "manual" in the configuration file, in which case the comment will be generated but no actual template. The reason for this is that the intent is that *eddie2.xsl* is imported into the actual stylesheet you are running. The comments make it easy to copy a template into your own stylesheet and change it; they alert you to conditions you might have to deal with in your new template.

Although the incompatibilities will all be spotted by DTD validation of the output, generating the warnings in XSLT allows them to be more specific. Two important aspects are firstly that the XSLT processing does not halt on an error, so that a complete list is generated, and, more importantly, the messages are in the domain of the input DTD, not the output. For example, if twelve templates all generate the same result element, validation of the result does not show where the faulty element was generated.

The Eddie 2 configuration file, then, can contain an override for any given element that can:

- Give a list of XPath expressions and warnings (a sort of simple Schematron-like process)
- Specify how the element is to be handled: one of:
  - Delete the element and its children;

- Expunge the element: delete it and its children, and ignore it for the purposes of content model comparisons;
- Copy the element, with default incompatibility warnings included;
- Manual: the element is handled in the main XSLT stylesheet and does not need to appear in the generated stylesheet.

It should be stressed that the XSLT stylesheet generated by Eddie 2 is intended to be imported into a hand-written stylesheet during the analysis phase; it can be used to provide initial templates (copied into the main "first" stylesheet by hand currently) but at the end of the analysis phase it is no longer used. Elements marked to be deleted in the configuration file should be matched in an empty template in the main stylesheet so as to be explicitly ignored.

The *eddie2.xsl* file, then, is an analysis tool. When input files are transformed by a stylesheet importing *eddie2.xsl*, messages produced by xsl:message will warn about likely problems. These messages can be sorted in order of frequency, so that after a few cycles of running the transform, dealing with the most common messages, editing the configuration file appropriately, and repeating, all available sample files will validate according to the target DTD. This turns out to be much more efficient than using an XML validator on the output, because *the messages are in the source domain, not the target domain*. For example, instead of, *element city not allowed at this location*, the message might be, *conference-loc contains city element not allowed in destination*; this is particularly useful when combined with custom errors.

4.1.2. The Generated Report

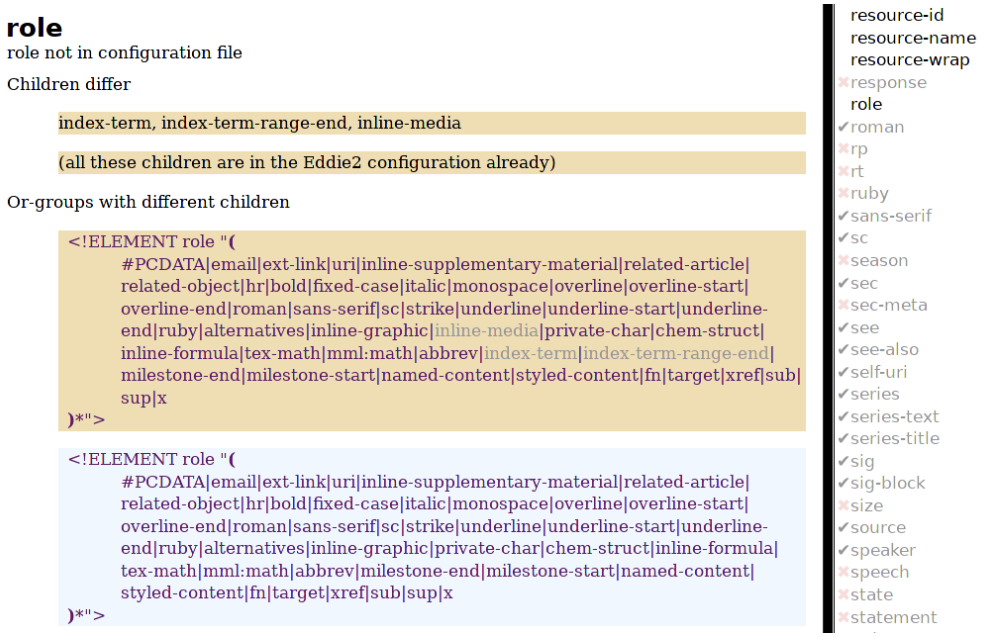
A screen-shot of a Web browser displaying part of an Eddie 2 report is given in Figures Figure 2 and Figure 3. In this example the source and destination elements have the differing content models and also differ in attributes, and these differences are highlighted. The check-marks on the right show elements that are configured; the grey element names are the same in content model and attributes in both DTDs. The screenshot has been separated into two parts for ease of printing, but of course is part of a single continuous HTML document that covers every element in the source DTD.

In the report, the light blue background indicates the destination DTD, and the yellowish background the source. In the source content model, elements not available in this element in the destination are given in grey. The list of elements on the right-hand side is a scrollable index; elements that are the same, or that have been configured, are in grey text. A check mark (✔) indicates that the element is configured and an ✖ indicates that it is not included in the Eddie 2 configuration file. Hovering over a checkmark gives hover text to describe its configuration. Typing the first letter of an element scrolls both the index and the report itself to the first element starting with that letter.

Figure 3 shows how attributes are reported, using both text and colour to describe the differences in the DTDs. One important point to note is that Eddie 2 reports differences such as where a default value changed, or where an attribute has a FIXED value in one DTD and not the other.

Not shown in the figure is that the content models are interactive in the report: hovering (or touching) within a parenthesized group shows the open and close parentheses connected by a dotted line. This is illustrated statically in Figure Figure 4. In addition, hover-text further clarifies the status of each element mentioned, and of course the element names are themselves links to their corresponding sections in the report.

Note that Eddie 2 is aimed at people writing XSLT to convert from one DTD to another; it is *not* aimed at people developing the DTDs. It does not show parameter entities, for example, and does not show which parameter entities a content model uses, nor which ones contain a reference to a given element.



The report shows both source and destination content models; this can be useful if, for example, a repeatable or-group in one DTD is a sequence in another, even if the allowed child elements are the same.

Figure 2. Eddie 2 Report: Screenshot (first part)

Attributes differ:

degree-contribution, vocab, vocab-identifier, vocab-term, vocab-term-identifier

Attributes in source but not destination:

degree-contribution CDATA #IMPLIED 0

vocab CDATA #IMPLIED 0

vocab-identifier CDATA #IMPLIED 0

vocab-term CDATA #IMPLIED 0

vocab-term-identifier CDATA #IMPLIED 0

content-type	CDATA	#IMPLIED
degree-contribution	CDATA	#IMPLIED
id	ID	#IMPLIED
specific-use	CDATA	#IMPLIED
vocab	CDATA	#IMPLIED
vocab-identifier	CDATA	#IMPLIED
vocab-term	CDATA	#IMPLIED
vocab-term-identifier	CDATA	#IMPLIED
xml:base	CDATA	#IMPLIED
xml:lang	NMTOKEN	#IMPLIED
content-type	CDATA	#IMPLIED
id	ID	#IMPLIED
specific-use	CDATA	#IMPLIED
xml:base	CDATA	#IMPLIED
xml:lang	NMTOKEN	#IMPLIED

Element role is found in:

collab, contrib, contrib-group, element-citation, mixed-citation, person-group, product, related-article, related-object, sig-block

[same in both source and destination]

Figure 3. Eddie 2 Report: Screenshot (second part)

front

front not in configuration file

Content model is the same in source and destination:

```
<!ELEMENT front "(
        journal-meta?,article-meta,(
                def-list|list|ack|bio|fn-group|glossary|notes
        )*
)">
```

Notice the two vertical lines of dots connecting a line in the element declaration with the line containing a matching parenthesis. The lines are shown only when the mouse pointer is over the parenthesized group (or when a mobile user touches that area), to avoid excessive visual clutter in complex content models.

Figure 4. Eddie 2 Report: Content model highlighting

4.2. Eddie 2 Plugins

A feature under consideration is support for external plug-ins; these can currently inject XSLT into the template for a given element, or for any element with a given attribute and/or child element. For example, a JATS Date plug-in could detect any element with both a *year* child and an ISO-format date attribute and generate XSLT to handle various cases of source and destination needing one or the other (or both). They might also inject icons into the scrollable index; see under Further Work below.

Along with a facility to rename elements, the date feature blurs the boundary between analysis and development and, although useful, is therefore experimental.

4.3. Eddie 2 in Use

The idea is that you write a stylesheet that imports the Eddie 2 XSLT; when you are finished with it, you remove the import, or make it conditional with an XSLT 3 *use-when* stylesheet parameter (which must be declared as static). Your stylesheet should also contain an identity transform at the start, or should use a default mode in XSLT 3 with on-no-match set to *shallow-copy* (which essentially does the same thing).

5. Discussion

In a way, Eddie 2 is not so different from other tools in this space. Even though the author was unaware of DTD Comparator when writing Eddie 2, there are some strong similarities. However, there are also differences, the most significant of which is a difference in approach.

One difference is the idea that Eddie 2 generates a stylesheet that pro-actively helps the developer, not by being an initial starting point but by being a continuously-updated configured part of analysis. This is where the term *An Analytical Approach* originates: instead of going through two DTDs element by element, an experimental approach is used of running sample documents and fixing the problems to get the majority of the way very quickly. Because the Eddie 2 configuration file can be updated as elements are handled, the index in the report is also a to-do list of things not yet considered.

Another part is the focus on the source domain: an edit-run-validate cycle produces messages in terms of the output, not the input. An Eddie configuration/run/review cycle produces messages in terms of the input, and hence in terms of what needs to be done to the XSLT transformation under development.

Experience suggests that using a tool such as Eddie 2 not only speeds up stylesheet development but also improves quality, by helping the developer to catch important cases.

6. Where Eddie 2 Does Not Help, and Future Work

Not all differences are measured by validation alone. DTD validation in particular is weak for finding transformation problems because it ignores actual content. A transformation which must change dates from American month/day/year format to day/month/year or international year-month-day may produce plausible but incorrect output, particularly for the first twelve days of each month. Good tests, perhaps with XSpec, can help, as can Schematron content rules; a RelaxNG or W3C XML Schema can also supply extra rules that help validation.

Sometimes elements change meaning more subtly. The term *van* in America describes a somewhat different sort of vehicle than the same term in the UK, and *SUV* is similarly somewhat different, so that the same vehicle might be in one category in one country and the other when it crosses the Atlantic. Such differences cannot be automatically detected without additional external infrastructure, and there is a real danger that a developer will skip over them.

To try to mitigate against the dangers of poor ontology matching in this way, a future version of Eddie 2 may be able to display or link to vocabulary documentation directly; this would be a good use of a plugin architecture.

Currently, although Eddie 2 works fine with DTDs thatuse parameter entities, a limitation in the XML parser that was used means DTDs must first have parameter entities expanded (flattened). A future version will use a different XML parser; the one selected had working XML Catalog support, wich remains a requirement. Note that in older SGML and XML projects it was common to support configurable element content models, so that a parameter entity in an actual document could change the grammar. There are currently no plans to support this in Eddie 2. However, vesitigal support *is* in plave for reporting on parameter entities used within content models, and this will probably be expanded in the future.

The author has also experimented with coverage reports by making Eddie 2 parse the manually-edited primary XSLT file and detect elements that have corresponding templates. Unfortunately this is, and will always be, unreliable: template match patterns are too powerful, and XPath expressions used in the *select* attributes of xsl:for-each or xsl:apply-templates are even harder: working out which elements are matched in general is equivalent to solving the halting problem, which is not possible. So a simpler approach is to read the XSLT stylesheet and report on which elements have templates that clearly match them, and to support a way of telling Eddie 2 that a particular template matches a particular set of elements. But at that point the value becomes unclear, compared to editing

the Eddie 2 configuration file; the main value is detecting discrepancies, places where a user edited the configuration file to say an element has been handled but then forgot to handle it, or made a typo in the element name. Eddie 2 does detect typos in element names in the configuration file, but not currently in the XSLT.

A future version of Eddie 2 may also accept XML Schemas (RNG or W3C) as input.

Eddie 2 is currently on gitlab, with access available on a limited bases, although that version does not support plugins or other experimental features, in order to prevent future compatibility issues.

7. Conclusions

By *analysis-driven development* the author of this paper means to suggest a process that is focussed on quantified analytical investigation and supporting tools. The idea is to work as much as possible in the problem domain and stay above implementation details as much as possible, without compromising quality.

Measurements have suggested that for a reasonably large vocabulary such as a customized version of JATS or BITS, the majority of a transformation can sometimes be completed in only a few hours, leaving only the content-based changes to handle. This compares favourably to the task of reading two versions of a DTD of course, but also compares well to experience with using other tools: the combination of domain-centered messages and a frequently-updated coverage report is very powerful.

Eddie 2 does not have any code in it that is specific to any particular DTD; writing an XSLT transformation using an analytical approach does not depend on the grammar in any way, although the larger the DTD and the more test files that are available, the greater the value of this approach.

Although Eddie 2 builds on many past ideas, the process of Analysis-based XSLT supported by tools is new. The tools mentioned in the paper are easily found; Norm Walsh may have given a paper at an SGML conference about his DTDParse; Earl Hood added the DTDDiff part and that was in part an inspiration for this work. The DTDDiff utility was alreay in use in 1999.

DtdAnalyzer was written at the USA National Center for Biotechnology Information (NCBI), a part of the National Library of Medicine (NLM) and described in a 2012 paper at JATS-Con given by Demian Hess, Chris Maloney and Audrey Hamelers.

XSLT Earley: First Steps to a Declarative Parser Generator

Tomos Hillman
eXpertML Ltd
<tom@expertml.com>

Abstract

Invisible XML [2] is a method for taking any structured text that can be parsed using a grammar, and treating it as XML. It allows the XML technology stack to be leveraged outside of XML structures.

For Invisible XML to be useful in pure XSLT transforms, a grammar-based parser available in XSLT is required: examples illustrating this are given. Parser-generators that provide parsers as XSLT are available, but they don't create parsers that work in the XSLT programming idiom, and can't parse ambiguous grammars.

An interpretation of the Earley [1] parsing algorithm may solve both of these problems: an Earley parser can parse any context-independent grammar, including any that may be ambiguous; it has also been suggested that the "Earley items" created as part of a parse operation can be reconfigured into a tree structure [5], which naturally lends itself to processing with XSLT.

This paper aims to lay the ground-work for producing a parser generator that creates XSLT which can parse string inputs given an EBNF-like grammar. Examples from previous papers on the topic will be used to manually create both an XML representation of the grammar, and the desired tree structure of Earley items. In turn, these should inform what an XSLT parser for that grammar should look like.

Finally the paper will discuss how the resulting parser can be abstracted and extended so as to parse using an arbitrary grammar, to use other grammar languages, and to investigate the possibility of generator for XSLT based parsers.

Keywords: XSLT 3.0, Earley, Invisible XML

1. Introduction

This paper is a continuation of the work in papers on Invisible XML and the Earley parser, particularly [3] and [5]. It attempts to demonstrate an implementation of the Earley algorithm [1] - or something very close to it - using the declarative programming idiom of XSLT rather than its traditional, procedural form.

The proof of concept that the paper aims to introduce is limited to a single pre-defined grammar; however it's hoped that this will form a groundwork for producing parsers and parser generators that can use not only any grammar, but grammars formed using a range of grammar languages, such as BNF and EBNF.

1.1. Invisible XML

Invisible XML was introduced by Steven Pemberton in his 2013 paper at the Balisage conference [2], and specified online [6].

It states that since all data is an abstraction, content can be equivalently expressed in a number of ways, including using XML. A simple piece of pseudo-code like:

Example 1. Proposed input

```
{a=0}
```

can be expressed without losing pertinent information in an XML format such as:

Example 2. Desired Output

```
<program>
   <block>{
      <statement>
         <assignment>
            <variable>
               <identifier>a</identifier>
            </variable>
            =
            <expression>
               <number>0</number>
            </expression>
         </assignment>
      </statement>
   }</block>
</program>
```

This is the example we will use to create our parser; it is taken from the slides of [3]

Expressing these data in an XML format allows us to use the XML technology stack to process them using tools like XQuery, XSLT, Schematron, and XSpec. For many who already have existing XML resources and expertise, this not only allows for employee proficiencies and reuse of systems, but also works within the declarative idiom.

Invisible XML also describes annotations to create attributes rather than elements, and to reduce those elements created in the parse tree that don't add

meaning to the content but are an accident of the grammar formulation. Recreating these isn't a primary goal of this paper, but doing so shouldn't present great technical difficulty.

1.2. Why Use XSLT based Parsers?

There are several features of Invisible XML that offer opportunities to process any data expressed in structured text. These can include documents (like Relax NG Compact, DTDs, XQuery, CSS, MarkDown, YAML, JSON, CSV, etc.), or formats embedded in XML (like path definitions in SVG, XSLT match patterns, or XPath statements).

Where these data are already being processed by XSLT - such as exports from content management systems, or rules based validation such as Schematron - it makes sense that an XSLT based parser can be used without introducing any new technological dependencies.

A useful example would be in rules-based validation; [4] gives the example of validating SVG paths, which use structured text within an attribute:

Example 3. An SVG Path [4]

```
<path d="M100,200 C100,100 250,100 250,200 S400,300 400,200"/>
```

Usually, checking the content of such an attribute value would be achieved by regular expression matching and checking. This is often the quickest and simplest solution, and it might be the best solution for simple structured text examples. Sometimes, however, even quite straightforward structured text grammars can require quite complicated and opaque regular expressions, leading to complex, verbose code which is hard to read and maintain.

An Invisible XML approach not only provides validation through successful parsing of the structured text, but also allows validation of specific data and relationships within and between both XML and non-XML structured text. Kosek was able to demonstrate the ability to extend Schematron by including a parser based on the grammar of these paths as an XSLT inclusion, checking both validity via parse-ability:

Example 4. Schematron rule testing SVG Path validity [4]

```
<sch:rule context="svg:path">
  <sch:report test="p:parse-svg_path(@d)/
                    self::ERROR">
    <sch:value-of select="p:parse-svg_path(@d)"/>
  </sch:report>
</sch:rule>
```

as well as more specific rule constraints, such as ensuring paths are contained within a defined coordinate space:

Example 5. Schematron rule testing path coordinate ranges [4]

```
<sch:rule context="svg:path">
  <sch:let name="path"
           value="p:parse-svg_path(@d)"/>
  <sch:assert
    test="every $c in $path//(signed-coordinate |
                      unsigned-coordinate)/number
          satisfies abs(number) le 1000">
  </sch:assert>
</sch:rule>
```

Having the parser available as XSLT therefore empowers developers who use any of the tools in the XSLT tool chain.

The other possibility that an XSLT parser allows is that of an extensible parser: this is discussed in more detail below.

1.3. Why not LL1 Parsers

There is a limited availability of XSLT based parsers; at the time of writing, there is one well known parser generator which can produce a parser in XSLT from EBNF grammars: [7].

Whilst this freely available tool has been invaluable in enabling approaches like the one above, it has a few limitations. One of these is that the parsers produced by [7] are LL1 or similar based parsers, and can't parse all possible *context free grammars*. In particular, they cannot parse *ambiguous* grammars; those which potentially allow for multiple valid parsed results, or multiple parsing routes resulting in the same results.

Example A.1 chosen for this proof of concept was chosen precisely because the grammar chosen won't work with LL parsers [3]: the first available symbols in `assignment` and `block` are both `identifier`, and therefore the grammar is ambiguous. It is perfectly possible in this case to rewrite the grammar so that it's not ambiguous through some clever abstractions, but this means that:

- using grammars may not be possible without careful editing;
- editing of the grammars may not be obvious, straightforward, or result in a concise representation of the underlying concepts;
- some grammars may not be used at all.

1.4. Writing Extensible Parsers

There is another limitation to using the [7]: the parser that it produces is not only an LL1 parser, but one that produces hundreds of state transitions that are

designed to be understood by its generated functions, rather than by a human developer. Because the code that is produced is impenetrable to ordinary humans, it is impossible for a human developer to take it and extend it to deal with extra features, let alone doing so applying the inherent approaches and strengths of XSLT.

The XSLT idiom involves match templates, "apply template" operations and native sequences. The procedural idiom of LL1 parsers involves passing state objects between functions. As well as being hard to understand, the latter is almost impossible to extend using XSLT's native import and precedence features.

Consider a proposed XSLT based parser for DTD documents. The DTD language is hard to process because it mixes a relatively simple EBNF grammar for the syntax with a mechanism for macro substitutions. It is easy to parse the grammar, but there is no way for EBNF to convey the meaning of entities and their expansions: entities will be treated as just another structure in the parse tree, without parsing any of the data which they represent. Expanding and including the entities would involve recursive operations on the results of each parse.

Better approaches may involve parsing the entities as they are defined, and including the results in the resulting parse tree; doing so would mean extending the generated parser with some bespoke code. One of the goals of this paper is to establish whether (or not!) it is possible to write a generated parser that would allow extension using the well established XSLT methods of doing so: over-riding templates in including stylesheets, priorities, and use of instructions like `xsl:next-match` or `xsl:apply-imports`.

1.5. The Earley Parser (very) briefly explained

The Earley parser is known as a chart parser: it works by compiling an intermediate data structure, originally conceived as a chart or set of *Earley Items*. Each of these items represents a step in a partial parse, evaluating one rule of the grammar on a defined sub-string. The real trick of the algorithm is that most of the useless partial parses are avoided altogether.

The process of (or function for) creating items is called the *recogniser*. This creates a number of item(s) consisting of the following information fields:

- The current *state*; the state is a representation of how much of the original string has been parsed (or how much of the string remains to be parsed).
- The current *rule* being evaluated; a rule consists of one *symbol* on the left hand side, which can be decomposed into a sequence of *terminal* (literal strings and keywords) and *nonterminal* symbols; the latter are nonterminal because they refer to other rules, and other sequences of possible symbols.
- The position within the rule; this is often given as markup in a representation of the rule itself, such as:

block → "{" ◆ statements "}" (1)

where the term before the arrow represents the *symbol*, terms to the left of the ◆ character represent rule definitions which have already been processed, and terms to the right those which have yet to be processed.

- The *start state*; that is, the state that was current when the processing of the current *symbol* and *rule* began.

The initial Earley item is normally defined by the grammar (often by convention as being the first rule in the grammar); the rest of the items are generated from existing ones according to the type of symbol to the right of the ◆ character:

Completion — If there is no next symbol, the rule has been completed. If there is a parent item, they can be advanced by a symbol and added in the current state.

Prediction — If the next symbol is a nonterminal symbol, we review our set of items to see if the nonterminal has already been processed in the current state.

If it has, we don't need to add any items by processing it again: this not only improves efficiency by avoiding repetition/ replication, but avoids a possibility of an infinite recursion.

If the nonterminal has not been processed, we add the corresponding rule to the set, starting at the beginning symbol as may be expected.

Scan — If the next symbol is a terminal symbol, we check to see if it matches the corresponding yet-to-be-parsed sub-string in the input.

When a match is achieved, we can advance to the next symbol, as well as advancing the current state.

When a match is not achieved, the rule has failed, and no further items are created from the current rule.

In this way, Earley items are added to the set until there are no more symbols left to resolve, or until there are no more items to add in the *final state* (i.e. the state representing the end of the parsed string). If there exists an item in this final state that is complete and started in the initial state, we know that we have found a valid parse.

Creating the parse tree can then be achieved simply by following the trail of items from this final item to the first, discarding any which are incorrect, incomplete or which do not contribute. Terminals form the leaf nodes, and Nonterminals form the containing branch nodes.

1.6. Macro Substitution

As previously discussed, there is another desirable property that would be useful in an extensible parser: the ability to handle macro substitution (such as DTD entity resolution) as part of the parse. While the implementation of such a feature is not the goal of this paper, establishing the possibility is.

It is perhaps not immediately apparent whether or not this is even possible: the effect is that the string to parse will be altered by the act of parsing itself.

The concept of the state of the parse in the Earley algorithm is often defined using character positions in the string. Clearly, this method will not support live changes to the input string whilst parsing! However, it seems reasonable that other methods of denoting the state ought to be possible: at any given point in the parse, what the state really needs to tell you is what is (or was) coming next (for the purposes of the next *scan* operation). The only other requirement is consistency in that all references must point to the same state.

2. Methodology

The essential approach is to take a grammar expressed in an XML syntax, and a string to parse. The parser works by transforming the grammar with the string as a parameter.

The proof of concept parser is available to view on github at https://github.com/eXpertML/XMLPrague2020/blob/master/Parser/EarleyParser.xsl. This paper will restrict itself to discussing some of the more pertinent design choices.

2.1. Choosing a Grammar Language

There are a number of grammar languages in widespread use, many based on the *Backhaus-Nauer Format*. Variants can be seen in use in specifications for many XML technologies, including the W3C specifications.

For this paper, Invisible XML was chosen because of the following characteristics:

- It includes optional and repeating definitions from EBNF, which make it much easier to write than standard BNF.
- It has an XML representation (see [6]), which makes it perfect for use with XSLT without having to bootstrap with another parser.
- It includes options which define the XML representation - which symbols represent elements, which attributes, and which can be omitted from the result tree altogether.

The last point alone makes Invisible XML uniquely suitable for the task.

In practice, a proof of concept did not require every feature of Invisible XML to be implemented at this time: attribute handling, for instance, is not required to handle our example, but should be trivial to implement in the future.

2.2. Determining the Earley Objects

Creating a sequence of Earley items was not strictly necessary, but was certainly helpful in understanding the Earley algorithm, and the resulting Earley trees.

A partial set of Earley items illustrating the complete parse is included in the appendix of this paper under the Table of Earley Items.

2.3. The Earley Tree

Defining the Earley Tree transpired to be a more interactive process than was initially envisioned, as it became apparent which information was necessary to the XSLT parsing algorithm, and how that was best represented. However, the basic principle remained the same: terminal symbols become text nodes; the nonterminal symbol on the left hand side of a grammar rule becomes a containing element; the right hand side becomes a sequence of contained nodes.

Since Invisible XML grammars will result in arbitrary element names, a namespace was chosen for most nodes in the intermediary Earley tree: `xmlns:e="http://schema.expertml.com/EarleyParser"`.

Invisible XML defines the starting rule as the first in the grammar [6]; this ensures that the entire Earley Tree is contained in a single root element, thus obeying XML well formed-ness.

Originally, elements in the Earley Tree were envisioned to be the final elements returned at the end of the parse. Ultimately it became apparent that writing templates to convert the Earley Tree to successful parse results would be easier to match a single element, `e:rule`.

Attributes are used to store useful information during the parse: `@state` and `@ends` are both space separated list of states where the evaluation of the rule in question can be said to begin and finish, respectively. The XML serialisations are also preserved in an optional `@mark`. The creation and use of some of these attributes will be examined in more detail later in the paper.

Since it is a truism that the first rule will never have already been matched in the initial state, we can create the root element of our Earley Tree:

```
<e:rule name="program" state="1" ends="0" >...</e:rule>
```

A rule in the Invisible XML Grammar can contain a number of alternative formulations. For these we recycle the elements `alts` and `alt` as `e:alts` and `e:alt`, respectively. Note that `e:alt` can only ever have other `e:alt` elements as siblings; the containing `e:alts` element is used to enable this restriction where alternatives

are required within the rule definitions. The same structures can be used to capture state ambiguities, i.e. when it there is more than one viable starting state resulting from the preceding parse operations:

```
<e:alts state="3 4 5" ends="0">
  <e:alt state="3">
     <e:fail state="3" string="}"/>
  </e:alt>
  <e:alt state="4">
     <e:fail state="4" string="}"/>
  </e:alt>
  <e:alt state="5">...</e:alt>
</e:alts>
```

Where a specific `e:alts` element needs to be referred to (we'll see why later), it can be given a generated id stored in the `@gid` element.

Optional elements are handled as an alternative using an `e:empty` element; this is a leaf node of the Earley Tree (i.e. an element which is empty):

```
<e:empty state="2"/>
```

Terminal symbols that successfully match are captured in an `e:literal` element; this allows for parse metadata to be captured in the same attributes as for the nonterminals in `e:rule`, and also has the benefit of avoiding the need for mixed text processing. An attribute `@remaining` is also used in the current implementation, which stores the new unparsed string that results after the terminal symbol has been matched:

```
<e:literal state="1" ends="2" remaining="a=0}">{</e:literal>
```

Terminal symbols that do not successfully match return the `e:fail` element; these currently include diagnostic attributes `@string` and `@regex` to show the failed match:

```
<e:fail state="3" string="("/>
<e:fail state="5" regex="^([0-9]).*?$"/>
```

These should be the last sibling children of their parents, as processing should not continue after a failure.

Non terminals are checked to see whether they have already been evaluated in the current state. If they have not, a new `e:rule` element is created. If they have, and evaluation ended in a failure, then an `e:fail` element is created. If the rule has already been evaluated, a place-holder reference is created, detailing all possible end states:

```
<e:nt state="2" ends="3" name="identifier"/>
```

2.4. State References

All possible states are stored as a sequence of strings: states are then referred by the integer corresponding to their index in that sequence. The first string in the sequence is always the complete initial string; a subsequent state corresponding to the string `$required` can then be added simply whilst avoiding duplicates:

```
($states, $remaining[not($remaining = $states)])
```

Similarly, the state reference number can be retrieved using:

```
index-of($states, $remaining)
```

There is one special case: the state corresponding to the empty string, which represents a complete parse of the entire input string: this is represented by the pseudo-index 0.

Note
Although this will normally be the case, there is no requirement that the state resulting from matching a terminal or nonterminal be a substring of the previous state: this allows for the possibility of parsing macro/entity substitution at a later date.

2.5. Tracking Visited Nonterminals

As has been discussed in Section 1.5, it is essential to differentiate between nonterminals which have already been visited in the current state, and those which have not.

When we find a nonterminal that has already been visited, it is also convenient to know the corresponding end states that will result from that segment of the parse.

To do this, we require another data structure, indexed by both nonterminal and by state number. This is implemented using XSLT 3.0 maps and stored as the tunnel parameter `$visited`:

```
{
  "program": {
    "1":""
  },
  "letter": {
    "2":"3",
    "3":""
```

```
  },
  "identifier": {
    "2":""
  },
  "block": {
    "1":""
  }
  ...
}
```

The structure is a map of maps indexed by either nonterminal name or generated id (the latter being used in the case of `e:alts`). The keys of the interior maps correspond to the states where the nonterminals have been matched, and their values (if present) to the possible end states should those nonterminals complete.

These maps can be conveniently serialized (e.g. to JSON[1]) for debugging purposes.

2.6. Controlling the Process Order

By now one of the primary challenges of creating the Earley Tree becomes apparent: each node that is created depends on the data structures for the states and visited nonterminals that are calculated from the preceding node. The usual approach of applying templates passes information from parents to children, not from preceding to following node.

Circumventing this default behaviour requires a replacement mechanism for `xsl:apply-templates`. Using a named template seems the obvious choice, since it allows us to preserve the context: `e:process-children`. We'll also need to define the `$children` of the template as a parameter, defaulting to the children nodes of the context element.

We can apply templates to the first sibling child of `$children`, storing it in a variable `$first` and then returning it as the first result of the sequence. If `$first` returns `e:fail`, or if there are no subsequent nodes in `$children`, then we can stop processing. Otherwise, new state and visited parameters can be calculated and passed as the children nodes to a new instance of the template, until the last of the original sibling nodes has been processed.

2.7. Dealing with Repetition: `repeat0`

Optionally repeating elements can be handled by simply re-writing them as a choice, much as suggested in the [6]:

[1]Other serialisation options are available, and - with a good Invisible XML parser - can be treated as XML! :)

```
<xsl:variable name="GID" select="(@gid, generate-id(.))[1]"/>
<xsl:variable name="equivalent" as="element(alts)">
  <alts gid="{$GID}">
    <alt>
      <empty/>
    </alt>
    <alt>
      <xsl:sequence select="(child::*[not(self::sep)], sep)"/>
      <xsl:copy>
        <xsl:attribute name="gid" select="$GID"/>
        <xsl:copy-of select="@*, node()"/>
      </xsl:copy>
    </alt>
  </alts>
</xsl:variable>
```

It might seem that a redefinition which contains itself like this would cause infinite recursion; however, recall that we can use generated IDs in the `$visited` parameter. By using the same check that we do for nonterminals, we can ensure that the interior `repeat0` is only run in the case where the state has changed; i.e. we only repeat processing if there is a match in the initial sequence.

2.8. Dealing with Repetition: `repeat1`

Now that we have a definition for an optionally repeating element, we can use it for a similar re-write for `repeat1`:

```
<xsl:variable name="equivalent" as="element()*">
  <xsl:sequence select="(child::*[not(self::sep)], sep)"/>
  <repeat0 gid="{generate-id(.)}">
    <xsl:sequence select="*"/>
  </repeat0>
</xsl:variable>
```

2.9. Pruning the Earley Tree

Converting the Earley Tree into one (or more) parsed result trees is now relatively straightforward; any element in the Earley Tree with a zero-length value of `@ends` (or where the element is missing entirely), or which contain `e:fail` can be suppressed. Other elements are processed as follows:

`e:rule[not(@mark)]`	Each rule is replaced with the name of the symbol. Alternative children are processed as for `e:alts`.
`e:rule[@mark eq '-']`	The containing element is skipped. Alternative children are processed as for `e:alts`.

`e:alts`	Templates are applied for each of the children alternatives, but only the children of the first are returned.
`e:alt[not(e:fail)]`	Templates are applied to the children elements; if no elements are returned, nor is the containing `e:alt`.
`e:nt`	References to nonterminals in a given state are replaced with the results of pruning the corresponding `e:rule` in the Earley Tree
`e:literal`	Literal strings are replaced with their string values.

For this proof of concept, it is enough to return the first viable result. However it should be possible to return a sequence of valid results for ambiguous grammars, should this be desirable.

Similarly, in the event of no complete parse, it should be possible to either return an error, or a partial parse. This proof of concept does the latter.

2.10. Results

The parser is largely successful, being able to parse the specified string in the chosen grammar:

Example 6. Results of parsing `{a=0}`

```
<program>
  <block xmlns:e="http://schema.expertml.com/EarleyParser">{
    <statement>
      <assignment>
        <variable>
          <identifier>a</identifier>
        </variable>=<expression>
          <number>0</number>
        </expression>
      </assignment>
    </statement>
    }</block>
</program>
```

Compared to the desired output, there remains only an extraneous namespace node on `/program/block` which should be possible to remove given a little more work.

In fact, it is possible to parse other strings in the grammar:

Example 7. Results of parsing `{while a do b=5}`

```
<program>
  <block xmlns:e="http://schema.expertml.com/EarleyParser">{
```

```
      <statement>
        <while-statement>while
          <condition>
            <identifier>a</identifier>
          </condition>do
          <statement>
            <assignment>
              <variable>
                <identifier>b</identifier>
              </variable>=
              <expression>
                <number>5</number>
              </expression>
            </assignment>
          </statement>
        </while-statement>
      </statement>
      }</block>
  </program>
```

3. Conclusions

3.1. Proof of Concept

The parser works for certain strings, and proves the concept of an XSLT based Invisible XML parser.

It is not a complete Invisible XML implementation: some XML serialization options are not fully implemented, and therefore does not yet work for the general case of either the input string or the grammar.

3.2. Earley enough?

The algorithm used by the parser is inspired by the Earley parser, but it has not been shown to be equivalent.

Functions to calculate the state sequence or map of visited nonterminals mean that elements in the grammar are processed multiple times; it is not immediately clear how and to what degree this may affect performance. Other options include embedding this information in the Earley Tree itself, resulting in an intermediate data structure many times larger than the desired result.

3.3. Extensible Parsing

The use of the state sequence means that it ought to be possible to write parser extentions that change the input string as it is parsed, allowing for entity and macro expanding parsers.

3.4. No Need for Parser Generators

Since the grammar is passed in as an argument to the parsing function, and used as an input to an xslt transformation mode, there is no inherent dependency on, nor a need to generate a particular parser for any given grammar. Instead we have a general purpose transformation library that can parse using *any* grammar supplied as Invisible XML.

4. Future Work

4.1. Full Invisible XML Implementation

The first and most obvious opporunity for future work is to complete the implementation of Invisible XML. This should not be an onerous task, as the list of remaining features to implement is small and consists mainly of serialisation options.

4.2. Parsing other Grammar Languages

Once Invisible XML as XML is fully implemented using the non-xml form becomes trivial: simply parse the grammar using the XML grammar definition.

This ability of the grammar to produce one representation of itself from another is also a great test of a complete implementation.

It is equally trivial to produce an Invisible XML grammar from any other grammar language: all that is required is an Invisible XML grammar representation of the other grammar language (not the grammar itself). In this way it should be possible to extend the parser to support parsing with EBNF and BNF grammars, such as those found in W3C specifications, without writing any new code.

4.3. Quality and Performance Improvements

Automated testing can be implemented in a straightforward way using a testing framework like [8].

Viability for scaled applications, and confirmation of performance scaling will require some performance testing with a range of input strings and grammars. Performance testing should also show whether performance scales proportionately to equivalent Earley parsers for the same grammar types.

4.4. Ambiguous Parses

Currently the pruning operation on the Earley Tree returns the first valid parse; it should be possible to optionally return multiple parses for ambiguous grammars.

It might also be possible to extend the parser to try multiple grammars for ambiguous strings, allowing for general strings to be parsed according to the first preferred grammar in a list.

Bibliography

[1] Earley, Jay (1970), *An efficient context-free parsing algorithm*, Communications of the ACM 13 (2): 94-102, DOI: 10.1145/362007.362035

[2] Pemberton, Steven (2013), *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 .

[3] Pemberton, Steven (2016), *Parse Earley, Parse Often*. In Proc. XML London 2016, University College London, June 4-5, pp.120-126. DOI: 10.14337/XMLLondon16.Pemberton01

[4] Kosek, Jirka (2017) *Improving validation of structured text*. In Proc. XML London 2017, University College London, June 11–12, pp.56–67. DOI: 10.14337/XMLLondon17.Kosek01 .

[5] Sperberg-McQueen, C. M (2017). *Translating imperative algorithms into declarative, functional terms: towards Earley parsing in XSLT and XQuery.* Presented at Balisage: The Markup Conference 2017, Washington, DC, August 1 - 4, 2017. In Proceedings of Balisage: The Markup Conference 2017. Balisage Series on Markup Technologies, vol. 19. DOI: 10.4242/BalisageVol19.Sperberg-McQueen01.

[6] Pemberton, Steven (2019) *Invisible XML Specification (Draft)*, retrieved from the web on 2019-12-10 https://homepages.cwi.nl/~steven/ixml/ixml-specification.html

[7] Gunther Rademacher (2019) *REx Parser Generator*, retrieved from the web on 2019-12-10 https://www.bottlecaps.de/rex/

[8] *XSpec*, retrieved from the web on 2020-02-10 https://github.com/xspec/xspec

A. Code Listings

Example A.1. Invisible XML Grammar

```
program: block.
block: "{", S, statement*(";", S), "}", S.
```

```
statement: if-statement; while-statement; assignment; call; block; .
if-statement: "if", S, condition, "then", S, statement, else-part?.
else-part: "else", S, statement.
while-statement: "while", S, condition, "do", S, statement.
assignment: variable, "=", S, expression.
variable: identifier.
call: identifier, "(", S, parameter*(",", S), ")", S.
parameter: -expression.
identifier: letter+, S.
expression: identifier; number.
number: digit+, S.
-letter: ["a"-"z"]; ["A"-"Z"].
-digit: ["0"-"9"].
condition: identifier.
-S: " "*.
```

Example A.2. Grammar (iXML as XML format)

The grammar is available in XML format at https:// github.com/ eXpertML/ XMLPrague2020/blob/master/Parser/Program.ixml

Table A.1. Earley Items for `{a=0}`

#	Symbol	Rule	Start State	Notes
S(0) - ∧{a=0}				
1	program	→ ◆ block	0	Because the rule specifies a nonterminal, we have to *predict* the next rule, #2, for block
2	block	→ ◆ "{" statements "}"	0	Now that the next symbol is the terminal symbol { we can *scan* to see if the input's next symbol matches. On a match, we create a modified Earley item #3 in the next state set, S(1)
S(1) - {∧a=0}				
3	block	→ "{" ◆ statements "}"	0	NB the starting point is unchanged. Now we can continue to *predict* and *scan* items #4 and #5 from the nonterminal statements.
4	statements	→ ◆ empty	1	The start state for predictions is set to the current state number. This *predicts* #6
5	statements	→ ◆ statement statements	1	Because there are many choices for statement, an item is predicted for each of those choices #7-11
6	empty	→ ◆	1	This is our first *completion* (the ◆ marker is at the end of the rule). The start state is 1, so we look in **S(1)** for the rule that *predicts* 'empty' - i.e. item #4. We can then restate #4 as #12, moving the nonterminal to the left-hand side.
7	statement	→ ◆ if_statement	1	*predicts* **if_statement**
9	statement	→ ◆ assignment	1	*predicts* **assignment**
10	statement	→ ◆ call	1	*predicts* **call**

#	Symbol	Rule	Start State	Notes
12	statements	→ empty ◆	1	a *completion* of #4 resulting in #13
13	block	→ "{" statements ◆ "}"	0	a *scan* of the next character, 'a' will fail to match '{', so no further items are created from this parse branch
14	if_statement	→ ◆ "if" condition "then" statement else-option	1	a *scan* of the next token fails; no further actions
16	assignment	→ ◆ variable "=" expression	1	*predicts* **variable**
17	call	→ ◆ identifier "(" parameters ")"	1	*predicts* **identifier**
19	variable	→ ◆ identifier	1	*predicts* **identifier** - note that this is the same *prediction* that results from #17, so we don't need to run this twice...
20	identifier	→ ◆ [abxy]	1	A *scan* of the next character ('a') succeeds - we can proceed to the first item of state S(2)
S(2) - {a∧=0}				
21	identifier	→ [abxy] ◆	1	A *completion* of #20 resulting in #22 and #23
22	variable	→ identifier ◆	1	A *completion* of #19 resulting in #24
23	call	→ identifier ◆ "(" parameters ")"	1	a *scan* of the next token fails; no further actions
24	assignment	→ variable ◆ "=" expression	1	a *scan* of the next token matches, so we can create a new item and increment the start state
S(3) - {a=∧0}				
25	assignment	→ variable "=" ◆ expression	1	*predicts* expression
26	expression	→ ◆ number	3	(other potential nonterminal matches for expression are skipped here for brevity)
27	number	→ ◆ [0-9]	3	An example of how to cope with '+' - it's equivalent to a choice between a single instance...
28	number	→ ◆ [0-9] number	3	... or a single instance followed by the same nonterminal.
S(4) - {a=0∧}				
29	number	→ [0-9] ◆	3	A *completion* of #27 resulting in 31
30	number	→ [0-9] ◆ number	3	*predicts* **number** (#32 and #33)
31	expression	→ number ◆	3	A completion of #26 resulting in 34
32	number	→ ◆ [0-9]	4	a scan of the next token fails; no further actions
33	number	→ ◆ [0-9] number	4	a scan of the next token fails; no further actions
34	assignment	→ variable "=" expression ◆	1	A *completion* of #25 resulting in #35
35	statement	→ assignment ◆	1	A *completion* of #9 resulting in #36
36	statements	→ statement ◆ statements	1	A *completion* of #5 resulting in #37 and #38 being a *prediction* for **statements**
37	statements	→ ◆ empty	4	*predicts* **empty** #39
38	statements	→ ◆ statement statements	4	We're going to skip the list of nonterminals here for brevity; it is left as an exercise for the reader to show that none of them will complete satisfactorily!
39	empty	→ ◆	4	*completes* itself

#	Symbol	Rule	Start State	Notes
40	state-ments	→ empty ◆	4	*completes* #37
41	state-ments	→ statement statements ◆	1	*completes* #36 giving #42
42	block	→ "{" statements ◆ "}"	0	
S(5) - {a=0}∧				
43	block	→ "{" statements "}" ◆	0	Now we have a *completion* of the entire string, ending at the final state **S(5)** and beginning with the initial state **S(0)** - but we aren't quite finished because it doesn't match the start symbol...
44	program	→ block ◆	0	Parse Success!

Jiří Kosek (ed.)

XML Prague 2020
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 2020

ISBN 978-80-906259-8-3 (pdf)
ISBN 978-80-906259-9-0 (ePub)